UNIVERSITY OF CALIFORNIA,
IRVINE

Interactive Worst-case Execution Time Analysis of Hard Real-time Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Trevor Wade Harmon

Dissertation Committee:
Professor K.H. (Kane) Kim, Chair
Professor Kwei-Jay (K.J.) Lin
Professor Jean-Luc Gaudiot

2009

DEDICATION

To my wife, Florence, for her never-ending love and support

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

A dissertation is often thought of as a solitary endeavor, but it is very much a team effort. Its success depends on exposure to others' ideas and feedback from one's peers. No researcher, no matter how talented, can win the dissertation game by playing alone.

Without a doubt, the head coach of this dissertation's team is Raymond Klefstad, my advisor. He took a chance by recruiting me as a rookie graduate student, providing me with the funding and training I needed to stay in the game. Without his support, this dissertation would not have been possible.

I am also grateful for the entire coaching staff: Lichun Bao, Kane Kim, Falko Küster, and Jörg Meyer served on my qualifying examination committee. Kane Kim chaired the defense committee with the assistance of K.J. Lin and Jean-Luc Gaudiot. Their guidance was invaluable.

My teammates at the Distributed Object Computing laboratory deserve many thanks, as well: Juan Andrés Colmenares Diaz, Shruti Gorappa, Jie Hu, Hojjat Jafarpour, Jinhwan Lee, Mark Panahi, Krishna Raman, Gunar Schirner, Chia-Yen Shih, and Yue Zhang. When not helping me directly with coding or writing, they cheered me on from the sidelines, providing the moral support I needed to get out of the occasional slump.

Surprisingly, this dissertation received some major-league assistance from intellectual athletes all over the globe, some of whom I could never meet and had to collaborate only via email. Martin Schöberl deserves special thanks for creating the Java Optimized Processor (JOP), a device that allowed me to validate my claims with a physical implementation. He also assisted me with research ideas, funding, and even JOP technical support. Jochen Hönicke, creator of the Java Optimize and Decompile Environment (JODE), provided crucial advice that allowed me to modify JODE for annotated control flow analysis. Paulo Abadie Guedes, Raimund Kirner, and Rasmus Pedersen helped me gain valuable new insight into the finer points of worst-case execution time analysis.

Successful sports teams are often a result of generous funding, and doctoral dissertations are no different. The financial support of the National Science Foundation and its Graduate Research Fellowship Program helped me obtain vital equipment and extra time to focus on my research. I also thank the Institute of Transportation Studies at UCI for providing funding through a graduate research assistantship.

Finally, I thank my parents, Patrick and Sharon Harmon, who have supported me from the beginning.

CURRICULUM VITAE

Trevor Wade Harmon

EDUCATION

Ph.D. in Electrical and Computer Engineering **2009**
University of California, Irvine *Irvine, California*

M.S. in Electrical and Computer Engineering **2005**
University of California, Irvine *Irvine, California*

Bachelor of Science in Computer Engineering **1998**
Washington University *Saint Louis, Missouri*

Associate of Arts in Liberal Arts **1995**
Johnson County Community College *Overland Park, Kansas*

RESEARCH EXPERIENCE

Graduate Research Assistant **2003–2005**
University of California, Irvine *Irvine, California*

Research Assistant **1998–1999**
Institute for Biomedical Computing *Saint Louis, Missouri*

TEACHING EXPERIENCE

Teacher Trainee **2005–2006**
California Community College Internship Program *Santa Ana, California*

English Teacher **2001–2002**
Nova Group *Tarumi, Japan*

Math and Physics Teacher **1999–2001**
Peace Corps *Tumu, Ghana*

SELECTED HONORS AND AWARDS

Graduate Research Fellowship **2005–2008**
National Science Foundation

Engineers' Class of 1991 Scholarship **1995–1998**
Washington University

REFEREED JOURNAL PUBLICATIONS

Design, implementation, and test of a wireless peer-to-peer network for roadway incident exchange **2009**
International Journal of Vehicle Information and Communication Systems, pg. 288–305
T. Harmon, J. Marca, R. Klefstad, and P. Martini

REFEREED CONFERENCE PUBLICATIONS

Toward Libraries for Real-Time Java **May 2008**
Symposium on Object Oriented Real-Time Distributed Computing, pg. 458–462
T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad

A Modular Worst-case Execution Time Analysis Tool for Java Processors **Apr. 2008**
Real-Time and Embedded Technology and Applications Symposium, pg. 47–57
T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad

Interactive Back-annotation of Worst-case Execution Time Analysis for Java Microprocessors **Aug. 2007**
Embedded and Real-Time Computing Systems and Applications, pg. 209–216
T. Harmon and R. Klefstad

Identification and Removal of Program Slice Criteria for Code Size Reduction in Embedded Systems **May 2007**
International Embedded Systems Symposium, pg. 269–278
M. Panahi, T. Harmon, J. Colmenares, S. Gorappa, and R. Klefstad

Toward a Unified Standard for Worst-Case Execution Time Annotations in Real-Time Java **Mar 2007**
Parallel and Distributed Real-Time Systems
T. Harmon and R. Klefstad

A Survey of Worst-Case Execution Time Analysis for Real-Time Java **Mar 2007**
Java and Components for Parallelism, Distribution and Concurrency
T. Harmon and R. Klefstad

Automatic Performance Visualization of Distributed Real-Time Systems **Apr. 2006**
Symposium on Object-Oriented Real-Time Distributed Computing, pg. 531–538
T. Harmon and R. Klefstad

RTZen: Highly Predictable, Real-time Java Middleware for Distributed and Embedded Systems **Nov. 2005**
Middleware, pg. 225–248
K. Raman, Y. Zhang, M. Panahi, J. Colmenares, R. Klefstad, and T. Harmon

VADRE: A Visual Approach to Performance Analysis of Distributed, Real-time Systems **Jun. 2005**
Modeling, Simulation and Visualization Methods, pg. 121–126
T. Harmon and R. Klefstad

Late Demarshalling: A Technique for Efficient Multi-language Middleware for Embedded Systems **Oct. 2004**
Distributed Objects and Applications, pg. 1155–1172

G. Schirner, T. Harmon, and R. Klefstad

Adaptive Techniques for Minimizing Middleware Memory Footprint for Distributed, Real-Time, Embedded Systems **Oct. 2003**
Computer Communications, pg. 54-58
M. Panahi, T. Harmon, and R. Klefstad

BOOKS

Web Developer's Guide To Visual J++ And ActiveX **1996**
Coriolis Group Books

SELECTED TECHNICAL PUBLICATIONS

CORBA is dead! Long live CORBA! **Mar. 2004**
Linux Magazine

Portability and the ARM Processor **Sep. 2003**
Dr. Dobb's Journal

Linux and the iPAQ, Arm in Arm **May 2003**
The C/C++ Users Journal

SOFTWARE

Volta `volta.sourceforge.net`
Development tools for distributed, hard real-time Java software

HR-XSL `hr-xsl.sourceforge.net`
XML transformations for a curriculum vitae or résumé

INVITED TALKS

The Volta Project **Nov. 2007**
Orange County Embedded Java Users' Group *Irvine, California*

Real-time Java **Feb. 2007**
Orange County Embedded Java Users' Group *Huntington Beach, California*

Static Timing Analysis for Multi-Robot Systems **Nov. 2006**
Space and Naval Warfare Systems Center *San Diego, California*

PROFESSIONAL MEMBERSHIPS

Institute of Electrical and Electronics Engineers (IEEE)

ABSTRACT OF THE DISSERTATION

Interactive Worst-case Execution Time Analysis of Hard Real-time Systems

By

Trevor Wade Harmon

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2009

Professor K.H. (Kane) Kim, Chair

Hard real-time systems are already common in aerospace, automobiles, and industrial robotics, and they will become even more prevalent with the emergence of new domains such as computer-assisted surgery. Perpetual concerns over safety and overall mission success require a guarantee that these increasingly complex systems perform as designed.

One technique involves a static analysis to place an upper bound on the worst-case execution time (WCET) of hard real-time software. Without knowledge of WCET, the timing behavior of a program cannot be guaranteed. Although substantial research has been applied to the problem of WCET analysis, virtually all prior work has focused on increasing accuracy without regard to speed. The resulting implementations are often too slow to be integrated into the development cycle, requiring WCET analysis to be postponed until a final validation phase. Fixing timing errors after the code has been written is expensive, time-consuming, and may necessitate a redesign of the system.

A new approach, *interactive WCET analysis*, would prevent such problems. Its novelty lies in making timeliness a fundamental concern of system design from the moment the first line of code is written. Rather than waiting until an implementation

is complete before starting the analysis, it incorporates knowledge of worst-case time into the development cycle continuously and dynamically, allowing early detection and removal of timing errors.

This dissertation proposes a variety of methods for making WCET analysis interactive. It begins with a simplification of the problem by relying on Java-based microprocessors to eliminate many sources of unpredictability. It then presents contributions in a bottom-up fashion, starting with a technique for annotating control flow graphs with decompiled source code. These graphs are used to map the results of WCET analysis to source code as it is written, a process known as interactive back-annotation. Additional contributions include compiler support for type-safe WCET annotations, statically analyzable collection classes, and a fast tree-based analysis algorithm with method cache support. Each contribution is implemented within a suite of extensible, modular, open-source tools that demonstrate how interactive WCET analysis can be integrated into a traditional software development environment.

Chapter 1

Introduction

Ask three different software engineers what a real-time system is, and you will likely get three different answers. The video compression specialist will define real time as a system that encodes movies as fast as they can be recorded. A software developer at a sports news network might claim that his web site delivers basketball scores over the Internet in real time. And a 3D animation expert might rave about graphics hardware that generates complex computer images in the blink of an eye, without delay—that is, in real time.

None of these systems are truly real-time. In fact, they are more accurately described as *pseudo*-real-time. Many online stock market systems, for example, claim to provide real-time quotes, but there is no guarantee that the data they broadcast reflects current market conditions. External factors, such as traffic congestion on the Internet, could inject unpredictable delays that prevent price changes from appearing as soon as they are posted.

Examples like these show that the adoption of the phrase "real time" into modern English has clouded its meaning. Indeed, "real time" has in some circles become

synonymous with "real fast." Consider, for example, rendering software, which creates a picture or an animation based on a geometric model. The duration of this process depends on the complexity of the scene—a forest with trees and thousands of leaves will take much longer to render than a simple cube.

This mathematically intensive process is also heavily dependent on the speed of the hardware doing the calculations. When Pixar's *Toy Story* was first released, the animators relied on 117 computers working in parallel; each frame of the movie took about two hours of processor time [1]. Today, Pixar's render farms are more than 300 times faster, but the rendering of a *Toy Story* scene would still take around four minutes. There is a substantial delay between the time an artist changes a model and the time that the model appears within the fully rendered scene. However, as computers grow ever faster, cheaper, and more parallel, this delay will drop. Eventually, it could disappear entirely, resulting in "real-time" rendering.[1]

This relationship between computer power and processing latency has led to a misperception that real-time simply means being fast enough to eliminate delays. But how fast is "fast enough," and what happens if the system sometimes fails to prevent a delay? Even if a render farm had enough computing resources to generate scenes in real time, stochastic events such as virtual memory paging or an excess of job requests could easily reduce its performance. The system would suddenly and unexpectedly cease to be real-time.

Fortunately, the consequence of losing real-time behavior in rendering software is ordinarily benign. A temporary decrease in responsiveness would merely cause frustration. If the delays become too long or frequent, the artist could simply take a

[1] In reality, render time is actually *growing* due to an appetite for detail and realism in animation that tends to grow faster than computing resources can supply. As a result, an average frame in Pixar's *Cars*, which was released ten years after *Toy Story*, took fifteen hours to render. The goal of real-time rendering, however, remains the same.

Figure 1.1: As potatoes roll through a sorting machine, a real-time system attached to an array of digital cameras scans them for abnormal bumps and discolorations. If detected, the system triggers a corresponding set of pneumatic fingers to redirect the spud into a separate bin. (Photograph by R J Herbert Engineering Limited.)

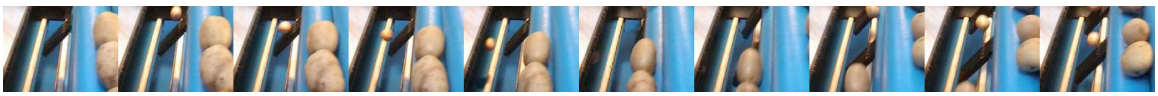

Figure 1.2: These video stills illustrate the real-time behavior of the potato sorter in Figure 1.1. In the first five stills, a camera has detected a discoloration, triggering a set of pneumatic fingers to redirect the potato into a defect bin as it falls. In the remaining stills, just milliseconds later, the camera triggers a different set of fingers for a different set of potatoes. (Video by R J Herbert Engineering Limited.)

coffee break until real-time performance returns.

In other real-time systems, however, the repercussions of lateness are more dire. Even seemingly mundane settings such as potato chip factories now rely on real-time systems to keep food sanitary. As potatoes zoom by on a conveyor belt, an array of digital cameras scans them for defects, as illustrated in Figures 1.1 and 1.2. If any dark spots are detected, a set of pneumatic fingers are triggered to redirect the defective potato off of the belt and into a separate bin. Any delay in this process, even on the order of milliseconds, could allow rotten chips to reach the consumer.

In more elaborate systems, the result of improper timing can be disastrous: a space-ship crash-lands in a desert, a tanker spills ten million gallons of oil onto a beach,

or a car skids out of control because its antilock brakes responded too slowly. Such systems simply cannot tolerate late results in a computation.

1.1 What Is a True Real-time System?

This extreme variation in the consequence of lateness raises a question: How does one separate the pseudo-real-time systems, such as rendering software, from the "real" real-time systems? The distinction is important, as it can mean the difference between an artist going on a coffee break or a catastrophe of fire, twisted metal, and possibly the loss of human life. It mandates an objective method of distinguishing the various types of real-time systems.

One popular classification system borrows a concept from economic theory known as *utility*, which describes how valuable something is. The utility of a commodity, for example, is a function of how much you have of it. Applying this idea of utility curves to real-time systems was first proposed by Jensen [2] as a paradigm for scheduling tasks. It has since developed into a means of defining the three basic types of real-time systems: non-real-time, soft real-time, and hard real-time.

Figure 1.3 and Table 1.1 illustrate this point. The utility curves indicate the relative importance of a result before and after some deadline at which the result is needed. In the case of non-real-time systems, the deadline is an ill-defined gray area. Web browsers, for instance, have a pseudo-deadline of approximately two seconds when loading pages [3]. Most users notice delays that last longer than this period, but of course a longer wait does not cause serious problems, only minor annoyances. Loading a web page after the deadline still has some utility, but the value gradually declines as the user gets more and more frustrated.

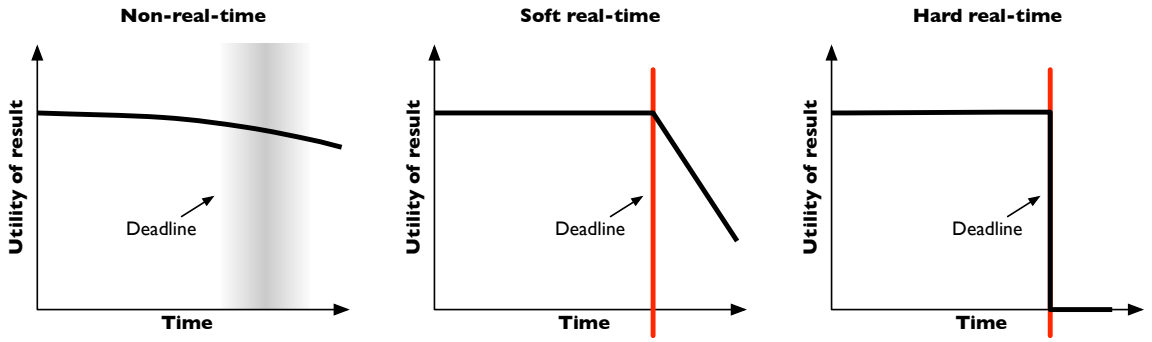

Figure 1.3: The utility curve paradigm shows that hard real-time systems place the highest emphasis on obtaining results at the proper time. If a deadline is missed, the value of the result is zero.

In contrast, a soft real-time system has stricter deadlines. A DVD player, which decodes video and audio streams in real time, is a typical example of this type of system. Unlike a web browser, its deadlines are more tightly defined, and the penalty for missing those deadlines is more apparent. In the case of lip synchronization, for instance, experimental results have shown that the sound of an actor's voice can be no more than 160 milliseconds out of sync with the corresponding picture [5], otherwise the mismatch becomes noticeable and distracting to the viewer. The utility of the result therefore drops sharply after this deadline. It does not immediately drop to zero, however, because the result may have some utility even if the deadline has passed. (The assumption here is that displaying an out-of-sync movie is better than no movie at all.) The deadline is very important but not absolutely crucial.

Finally, at the end of the spectrum of utility curves, the penalty for lateness is the most severe. These systems are known as *hard* real-time systems because failure to meet a deadline is a fatal fault. The utility of the result drops to zero instantly, as shown in the graph. Safety-critical systems, such as avionics, are a typical example of this case—after the airplane has crashed, there is no value in finishing the computation—but even when human lives are not at risk, hard constraints can be found in many real-time systems. The control of an unmanned Mars rover, for instance, requires

Table 1.1: In *Real-Time Java Platform Programming* [4], Peter Dibble offers these real-life examples of real-time deadlines.

Type	**Real-life example**
Non-real-time	Mowing the lawn. Getting it done earlier is better, but generally there is no point at which it suddenly becomes urgent.
Soft real-time	Fixing lunch for your children on a relaxed summer day. The children know the time they should be fed, and if you are late, they get fussy. The level of hungry complaints gradually increases after the deadline, but nothing really bad will happen.
Hard real-time	A child flushing a T-shirt down the toilet. The deadline for action is the moment before the child flushes. Missing the deadline probably means that the shirt is ruined and a plumber must be called. (A related example would be grabbing a child before she runs into traffic. This variation is known as a *safety-critical* hard real-time deadline because the safety of human life is at risk.)

timeliness in all calculations to prevent the vehicle from tumbling into a ravine and ruining the mission.

With this utility curve classification as a guide, the distinction between rendering software and potato chip machines is now clear: The former is a non-real-time system, while the latter is a hard real-time system. Of course, these are generalizations, and often there is overlap between the categories. A system may have a mixture of hard and soft real-time deadlines. For example, the control system for the potato chip machine might have a graphical user interface with soft deadlines while its air hose control system must keep to hard deadlines.

Distinct among the three cases, however, is the hard real-time version of the utility curve. It presents major challenges because it transforms timeliness into a correctness criterion. The overall correctness of the system depends on both functional correctness and temporal correctness. In other words, the right answer delivered too late is just as bad as the wrong answer.

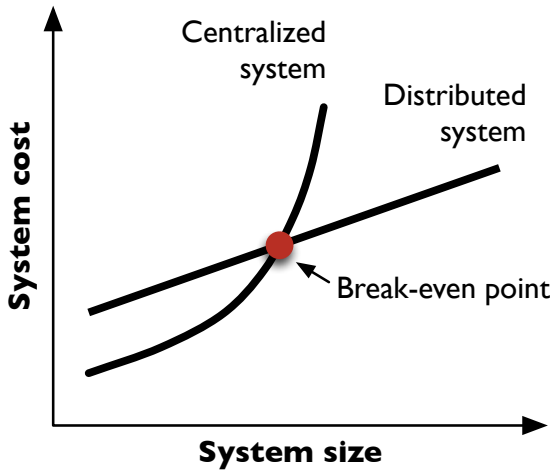

Figure 1.4: Kopetz observes that a centralized system with a single powerful CPU starts out with a lower cost, but as the system grows, there is a break-even point beyond which the hardware for a distributed system is cheaper [6].

1.2 The Rise of Distributed Real-time Systems

While hard real-time deadlines complicate every aspect of system design and construction, a new type of system gaining popularity in recent years promises to add yet another layer of complexity: the *distributed* real-time system. Such systems split computation across multiple processors, usually separated into physically independent nodes that do not share a memory or a clock. Each node works together toward a common goal and communicates with the other nodes over some network to share results and ensure synchronization. This modular approach increases performance by executing tasks in parallel, and it lowers costs, as shown in Figure 1.4, by sharing expensive resources among multiple nodes. It also improves reliability and maintenance because nodes can be made redundant and added or removed easily, at least compared to a centralized, monolithic system.

At the same time, a distributed architecture magnifies any existing complexities in the system. It adds new complications such as:

Clock synchronization In a distributed system, there exists no systemwide common clock. Even if all nodes in the system are initially synchronized to each other, the quartz crystals that regulate clock ticks in a computer are not perfect and will drift from the true time value. Some mechanism of synchronizing the clocks, such as the Precision Time Protocol [7], must be incorporated into the system.

Heterogeneous architectures Because nodes in a distributed system are independent, they may have different implementations. The node controlling a user interface, for example, may consist of a high-throughput Intel processor while sensor nodes at the edge of the network may be implemented as low-power ARM-based microcontrollers. The potential for a wide variety of implementations may not affect the high-level design and modeling of the system, but it can greatly hinder the development process because the same workflow and code base cannot be used for all elements of the system.

Dynamic task allocation Imagine a group of robots designed to perform a cooperative task. Individual robots may be assigned sub-tasks of the system-wide goal, but as the robots' environment changes, the sub-tasks may need to be continuously adjusted. For example, a robot needing to perform a lengthy computation but busy with other work could offload the task to an idle robot elsewhere in the system. This dynamic task allocation balances the utilization of resources and improves overall performance. In a distributed environment, however, there may be no central coordinator available to make task assignments, and no robot has a complete view of the system, thereby making the problem of task allocation much more difficult.

Blocking A distributed system, by its nature, depends on communication among nodes. A fundamental problem arises when one node sends a message to an-

other, but the receiver node is busy with some other task. The individual bits of the message could also be corrupted, perhaps due to interference in a wireless link, requiring the message to be re-sent. These events could cause the sender node to block while waiting for a response. Even if the message passing is asynchronous and no acknowledgment is required, the question remains of what the sender should do in the meantime. The precise scheduling of the real-time distributed system would be disrupted.

These issues add new and challenging dimensions to designing and building real-time systems. Making matters worse, they must all be addressed without neglecting the end-to-end timing constraints of the system. Despite these concerns, a distributed architecture is the preferred implementation of a real-time system [6], especially if the break-even point of Figure 1.4 is reached. The primary argument is composability.

In the context of real-time systems, composability means constructing larger systems by integrating well-specified and well-tested subsystems while maintaining timeliness throughout. The idea is that different components can be mixed and matched and will work correctly without having to redesign or retest those that have already been validated. Dependability arguments also advocate the distributed approach because it achieves fault-tolerance by replicating nodes.

As an example of composability, consider the first down line that appears to television viewers of American football games. This virtual, movable marker does not actually exist but is superimposed onto the screen, giving the illusion of a yellow line painted on the field. The illusion is enhanced by removing key parts of the image so that players appear to run over the line, as shown in Figure 1.5.

This on-screen magic is created by a distributed real-time system called 1^{st} and Ten from Sportvision. It consists of up to five separate broadcast cameras outfitted with

Figure 1.5: 1st and Ten from Sportvision is an example of a composable, distributed real-time system. It consists of four high-end workstations for video processing, three embedded systems for data acquisition of camera movements, and a standard personal computer for the human operator's user interface. (Photograph and frame capture by Sportvision, Inc.)

custom sensors and encoders to capture the pan, tilt, and zoom of the cameras. The data enables the virtual line to follow suit, staying in perspective and getting larger or smaller as needed. This feature places very rigid timing constraints on the system; it must check camera positions, map the position of the line, and superimpose the image onto the video feed thirty times every second.

Because 1st and Ten's architecture is distributed, its components are *composable*. Any of the computing nodes in the system can be swapped in and out, as long as they provide the necessary thirty-frames-per-second timeliness. New and improved sensors and actuators can be tested separately, before system integration, to make sure that they meet this requirement. In addition, the redundancy of multiple nodes means that if one camera fails, the system as a whole remains functional. This pluggable, distributed approach to real-time systems helps reduce overall cost and increase fault-tolerance.

1st and Ten is certainly not the only distributed real-time system to see commercial

success. These types of systems are becoming increasingly popular in order to take advantage of the improved scalability, flexibility, and performance that distributed designs offer. A variety of cutting-edge applications in this area have begun to appear, such as multi-robot systems [8]. To facilitate the construction of such systems, the industry has adopted various standards for real-time network communication:

- The Controller Area Network (CAN), a protocol and bus standard, allows microcontrollers and similar devices to communicate with each other. The protocol can act as a priority-based global dispatcher to support hard real-time communication under certain fault assumptions [9].

- The Time-Triggered Protocol (TTP), designed for hard real-time safety-critical applications, is a convention for fault-tolerant communication [10]. It uses a Time Division Multiple Access (TDMA) protocol where all nodes share a common time base. Collisions and overloads are guaranteed not to exist. A competing standard, known as FlexRay, shares many of the same goals. Both have been adopted by major automobile manufacturers to implement drive-by-wire, brake-by-wire, and other "X-by-wire" systems [11].

- The Common Object Request Broker Architecture (CORBA) [12], a specification for a type of software known as *middleware*, resides between applications and the underlying operating system. It decreases the effort required to develop distributed systems by allowing them to be composed from reusable components, rather than building the software from scratch. The Real-Time CORBA specification [13] extends the standard to support end-to-end predictability for remote operations. Real-Time CORBA has been used in a range of distributed real-time systems, including avionics missions, high-energy physics experiments, and image processing (see Figure 1.6) [14].

Figure 1.6: The Linatronic system from Krones is a network of ten cameras that examines glass bottles for defects. It relies on Real-Time CORBA middleware to process the images within fifty milliseconds, providing a guarantee on throughput of twenty bottles per second. (Photograph by Krones AG.)

- Like Real-Time CORBA, the Data Distribution Service (DDS) is a middleware specification for real-time systems [15]. Unlike CORBA's synchronous, tightly-coupled messaging paradigm, DDS takes a publish/subscribe approach where the sender of a message does not necessarily know who its receivers are. This decoupling of publishers from subscribers allows for increased scalability and more flexible network topologies. DDS has been used for military simulations, ship subsystem control, and unmanned underwater vehicles.

Despite the inherent advantages in these standards, the practice of building distributed real-time systems remains difficult. Some observers have even suggested that it is the most ambitious type of software engineering project that one could undertake. Eric S. Raymond, writing in *The Art of UNIX Programming* [16], noted the dangers of this new horizon:

> *The combination of threads, remote-procedure-call interfaces, and heavy-weight object-oriented design is especially dangerous. Used sparingly and tastefully, any of these techniques can be valuable—but if you are ever invited onto a project that is supposed to feature all three, fleeing in terror might well be an appropriate reaction.*

In a presentation on information timeliness [17], Thomas F. Lawrence was even more lugubrious, claiming:

> *If a software development project can be reasonably classified as distributed real-time, it has an 80% chance of failure.*

Indeed, the field of distributed real-time systems is relatively immature, and the tools and techniques to aid their development are still low-level and somewhat basic. The development process, when targeting non-trivial distributed designs, tends to be cumbersome and tedious. When combined with the hard real-time aspect, which demands increased testing and validation, it becomes an extremely arduous task.

Tom Van Vleck summed up the current situation nicely [18]:

> *We know about as much about software quality problems as they knew about the Black Plague in the 1600s. We've seen the victims' agonies and helped burn the corpses. We don't know what causes it; we don't really know if there is only one disease. We just suffer—and keep pouring our sewage into our water supply.*

Chapter 2

Interactive Analysis

While the bubonic plague revealed inadequacies of medical science, it also led to a revolution in the way diseases and the human body are dealt with. Following the Black Death of the 1300s, more emphasis was placed on anatomical investigation. Doctors began to understand the inner workings of the body rather than simply treating symptoms from the outside.

Today, software engineering of real-time systems is at a similar crossroads. These systems are often so complex that they sometimes seem as mysterious as the *Yersinia pestis* bacterium was to doctors of the medieval era. Of course, the mere inscrutability of software is unlikely to cause a deadly pandemic on the scale of the Black Death, yet we are becoming increasingly dependent on real-time systems in our everyday life. No longer restricted to esoteric space and military applications, they are now responsible for keeping us alive. When we drive, they regulate the engine and brakes. When we fly, they maintain the aircraft's flight path and help it take off and land. When we are sick, they regulate our blood pressure and heartbeat.

Unfortunately, the revolution in medicine that followed the bubonic plague seems

overdue in the real-time software field. The functionality provided by such software is becoming ever more complex, especially for the distributed variety, and the job of testing for correct timing behavior becomes ever more difficult. The current situation manifests itself in ominous failure statistics. In the automotive industry, for instance, 30% of electronics system breakdowns can be traced back to software timing problems [19].

Compared to the remarkable instruments available to modern doctors—X-ray machines, CAT scans, electrocardiograms—the tools for analyzing the wellness of real-time systems is considerably limited. Instead of understanding the timing properties of a system at a fundamental level, engineers typically derive these properties through coarse external observation. Research papers on real-time middleware, for instance, use empirical evidence to make arguments about predictability. Figure 2.1 is a typical case; it shows timing measurements obtained from two middleware implementations [20].

The figure is essentially a histogram of latencies. It demonstrates that, for example, the latency of method invocations in the middleware is almost always 2,000 microseconds for 32-byte messages in a particular test environment. In rare instances, the latency jumps to over 2,300 microseconds but never reaches 2,400. The middleware authors rely upon these measurements to show that the software has firm upper bounds on latency and is thus able to meet real-time constraints.

This evidence is circumstantial, however. Basing any claim of predictability on these measurements is specious for two reasons:

Inferences vs. Guarantees Measurements *infer* predictability but do not *guarantee* it. They apply only to a particular set of inputs in a particular environment. Such inferences may provide suitable confidence for soft real-time systems, but

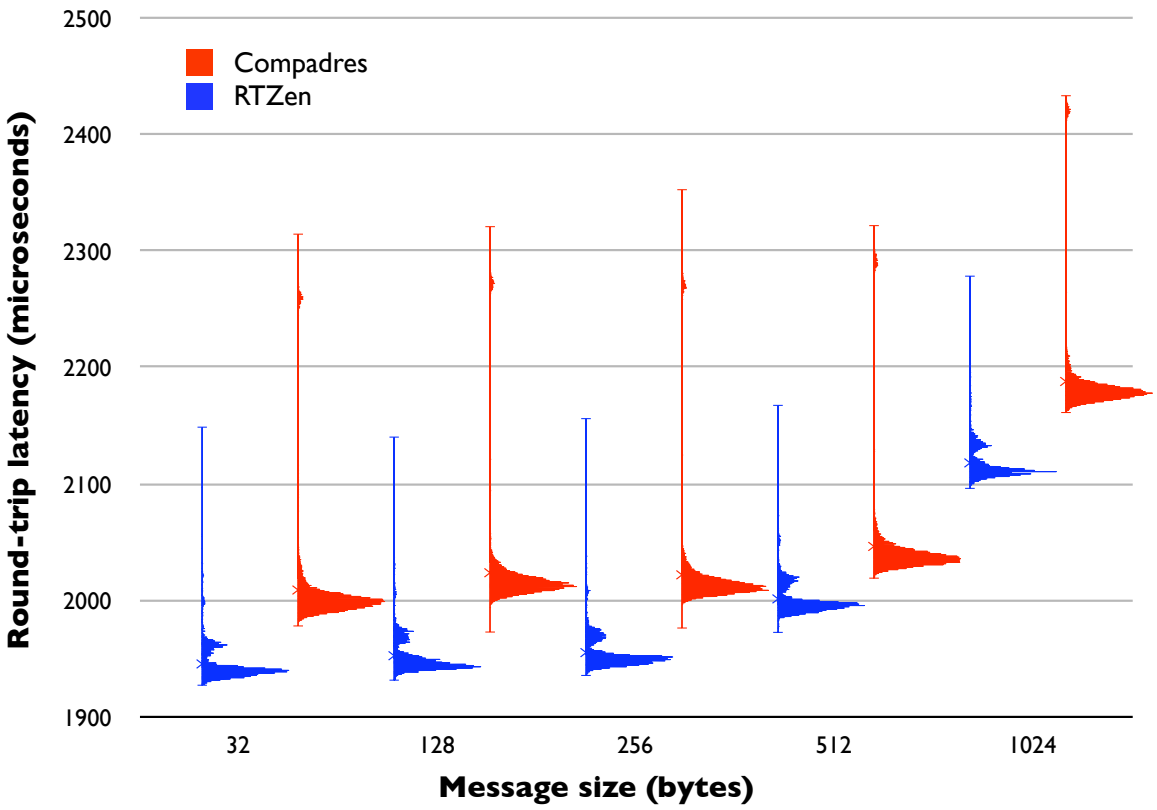

Figure 2.1: A series of tests under controlled conditions, such as these measurements of method invocation latency in two middleware implementations, is often cited as evidence of predictability in real-time software.

relying on them is dangerous for hard real-time and especially safety-critical systems. An unexpected piece of input data could cause the system to react more slowly than was measured during testing, as illustrated in Figure 2.2. Attempting to work around this problem by iterating through all possible inputs would only lead to a state explosion that would be infeasible to measure for any non-trivial program.

The Heisenbug Principle In 1927, Werner Heisenberg theorized that measuring the momentum of a subatomic particle makes its position uncertain, while locating its position makes its momentum uncertain. This Heisenberg Uncertainty Principle is analogous to the "observer effect," in which the act of observation changes the phenomenon being observed. In the case of a real-time system, the

> very act of measuring its performance can disturb the system's real-time characteristics in a way that may cause missed deadlines and scheduling conflicts that would not occur under normal execution. This uncertainty has led some researchers to dub the condition a "heisenbug."[1] In addition to making performance problems exceptionally difficult to understand and repair, bugs of this kind also cast doubt on the validity of measurements such as those in Figure 2.1. For example, the study's authors were unable to explain the jitter that occurs near the upper end of each histogram for the Compadres data set. Whether this anomaly will continue to occur after system deployment or if it is simply a peculiar artifact of the testbed is unknown.

There is simply too much uncertainty in measurement. Without careful planning and extensive analysis during design and implementation—not just measurements taken during the final test phase—the system has no guarantee on response time and degrades to a best-effort approach. In practice, building a real-time system thus becomes more like an art than a science. A common tactic in the safety-critical aerospace industry, for instance, is to over-design systems so that processor utilization is extremely low—around 1%—in the hope that unplanned behavior does not exceed CPU resources, resulting in missed deadlines and critical failure.

This over-provisioning of the processor is wasteful. It demands CPU resources two orders of magnitude greater than what is actually required (1% vs. 100%). Worse, it unmasks a more fundamental problem: Despite decades of research, practitioners still cannot trust modern tools and techniques to produce a real-time system that

[1] The heisenbug pun has a storied etymology. Originally coined by Bruce Lindsay [21], its first appearance in print came from Lindsay's colleague Jim Gray in 1985 [22]. Since then, the term has spawned a variety of similar monikers for unusual software bugs: the bohrbug, which occurs reliably under a well-defined set of conditions; the mandelbug, whose causes are so complex that its behavior appears chaotic; and the schrödinbug, which manifests only when someone observes that the program never should have worked in the first place, at which point it promptly stops working for everyone until fixed. An excellent treatise on the history of the heisenbug and its offspring was compiled by Grottke and Trivedi [23].

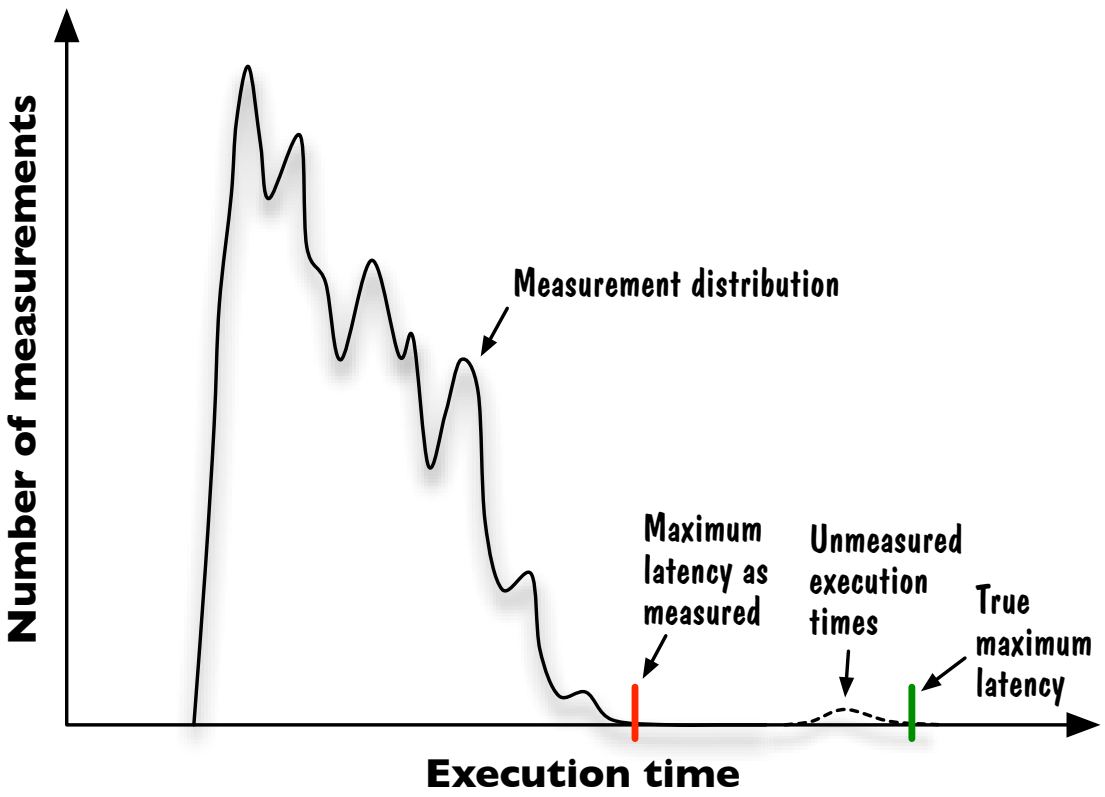

Figure 2.2: This histogram of execution time for a fictional real-time task illustrates the weakness of measurement when making claims of predictability. Performance testing may produce a data distribution like the shown one here, leading the developer to underestimate the maximum latency of the task.

performs as expected. There is clearly not enough confidence in the predictability of software. For hard real-time systems, a deeper, stronger guarantee is necessary.

2.1 The Importance of Worst-Case Execution Time

This is not an immutable situation. In 1986, while developing a real-time variant of the programming language Euclid, Kligerman and Stoyenko put forth a concept known as *worst-case execution time*, or WCET [24].[2] WCET places an upper bound

[2]Over the years, WCET has been known by other names, such as *maximum execution time*, or MAXT. These terms have since been deprecated in favor of WCET.

on the execution time of a given software task, where "execution time" is simply the time a particular processor takes to execute that task. The idea behind WCET is to make timeliness a property that can be formally analyzed rather than simply measured. It yields a provably correct bound rather than an educated guess.

This concept of worst-case execution time is a fundamental departure from *average-case* execution time, a performance metric with which most software developers are concerned. Developers of hard real-time systems, on the other hand, must be intimately familiar with worst-case performance. Without knowing the WCET for each time-dependent task in the system, no guarantee can be made that the system will meet its deadlines, possibly leading to critical failure or even injury and loss of life.

In spite of this fact, worst-case execution time analysis has received little attention—relative to the real-time field in general—from the research community. In universities, too, courses on real-time systems focus primarily on scheduling algorithms, normally omitting WCET from the syllabus. And in industry, the situation is similar: WCET analysis is seldom performed or even ignored entirely.

Part of this apathy toward WCET comes from the ostensible maturity of the field. Remarkable advancements in real-time research, such as priority inversion avoidance, real-time garbage collection, and operating systems designed for real-time behavior, give the impression that all of the requisite elements for building a time-predictable system are readily available. There is even an entire hierarchy of task scheduling algorithms, shown in Figure 2.3, that are able to order the release time of tasks such that each one is guaranteed to meet its deadline (assuming that such an order exists).

An essential element is missing, however. All of these scheduling algorithms *require* knowledge of worst-case execution time. Without this key piece of input, any resulting schedule is invalid. For example:

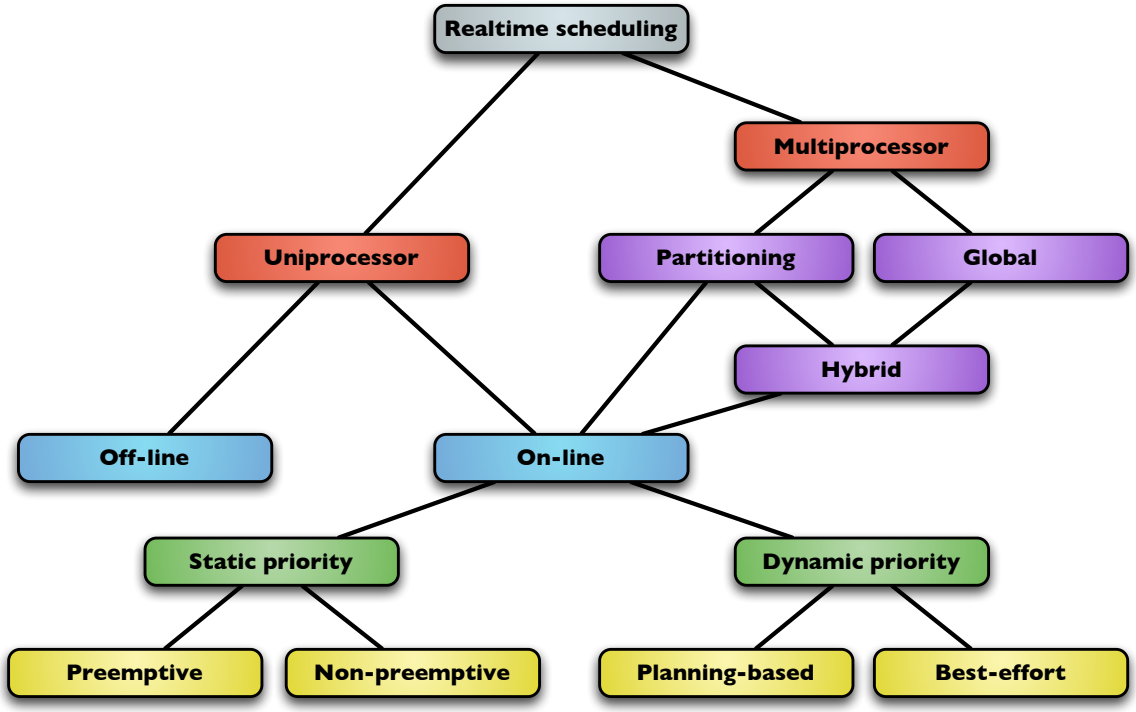

Figure 2.3: Real-time scheduling algorithms devise a feasible schedule according to various strict constraints, such as whether a task can be pre-empted. This figure, based on work by Mohammadi and Akl [25], shows the basic taxonomy of these algorithms.

Table 2.1: Real-time scheduling algorithms require worst-case execution time (the C variables) to determine whether deadlines can be met.

Rate-monotonic	$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \left(\sqrt[n]{2} - 1 \right)$
Earliest deadline first	$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$
Least slack time	$slack = d - t - c$
Time-triggered	$period = \sum_{i=1}^{n} C_i$

Rate-monotonic This algorithm assumes that no task has a WCET longer than its deadline. Furthermore, its feasibility analysis requires WCET as input: $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(\sqrt[n]{2} - 1\right)$, where U is processor utilization, n is the number of processes, T_i is the release period, and C_i is the WCET.

Earliest Deadline First (EDF) This scheduler does not require WCET knowledge at run-time. However, EDF is unstable (one late job causes many other jobs to be late), so a guarantee against overloading is vital. The only way to make this guarantee is to perform an acceptance test: $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$, where the variables have the same meanings as above. Again, note that WCET is part of the equation.

Least Slack Time (LST) The slack (also called "laxity") of a real-time task is defined as $d - t - c$, where d is the deadline, t is the current time, and c is time required to complete the remaining portion of the job. Determining c requires knowledge of the WCET.

Time-triggered Also known as *clock-driven* or *time-driven*, this approach depends on a static schedule computed prior to execution. The static schedule dictates an exact moment in time for each task's release. In such a scheme, task n cannot be scheduled until task $n - 1$ completes; otherwise, two tasks could be released at the same time. Preventing this failure requires knowledge of each task's WCET.

Note that *all* of the above algorithms depend on WCET, as shown in Table 2.1. For this reason, even if the underlying operating system implements a solid, carefully crafted real-time scheduling algorithm, making any guarantee about the predictable behavior of a task is impossible unless its WCET is known. For example, the rate-monotonic scheduler is provably optimal in the sense that if a feasible schedule exists,

the algorithm will find it [26]. It is a fallacy to presume, however, that the mere virtue of having a rate-monotonic scheduler incorporated into a system makes that system optimally real-time. If the WCET cannot be guaranteed, then the rate-monotonic algorithm—or any other known scheduler—also provides no guarantee.

Given the importance of WCET, the natural question is how to obtain it. While most real-time practitioners are aware of its significance, the job of calculating the WCET is often perfunctory or even neglected. In many cases, a value for the WCET is simply assumed to exist with little or no explanation. Consider, for instance, a white paper from TimeSys Corporation describing a real-time video decoder system [27]:

> *By experimentation, it has been found that the longest frame decode time on a given processor is six milliseconds.*

In other words, the WCET was determined through *ad hoc* measurement. If any portion of the hardware or software were to change, the experiment must be conducted again.

Similar examples abound. In the Time-triggered Message-triggered Object (TMO) programming scheme [28], WCET is fundamental to the underlying event model and must be specified explicitly, yet the mechanism for determining this WCET is left undefined. Likewise, the Real-Time Specification for Java [29] exposes API methods, such as ReleaseParameters.setCost(*RelativeTime*), under the assumption that the WCET parameter will somehow be obtained but without specifying how.

2.2 Current Methods for Obtaining WCET

Because of the lack of emphasis on finding WCET, methods for obtaining it are somewhat primitive. A common tactic in industry is to perform a series of tests under varying conditions and attempt to extrapolate the actual WCET from the resulting measurements. While this approach works well for *average* response time, *worst-case* response time is not amenable to measurement. An unexpected set of input data could cause the system to react more slowly than was measured during testing, as indicated in Figure 2.2. An upper bound on the WCET cannot be guaranteed through measurement alone.

A more dependable and systematic approach to finding the WCET involves a *static* analysis. Given the executable code for a task and the processor on which it will run, static analysis provides a guaranteed upper bound on the time taken to execute the task. Consider, for example, the following high-level expression:

```
a = b * c
```

Compiled into low-level executable code, the expression may look something like this:

```
movl    -16(%ebp), %eax
imull   -12(%ebp), %eax
movl    %eax, -20(%ebp)
```

In theory, finding the WCET of this statement statically is simply a matter of summing the number of CPU cycles that each of these three instructions requires.

In reality, however, static analysis is never so simple. Large pipelines, branch prediction, multi-level caching, and other sophisticated features of today's processors cause

huge variances in execution time. While such features greatly improve average performance, they come at the expense of worst-case performance. In the listing above, for example, the movl instruction might take just one cycle if the requested data is already in the cache, or many hundreds if the processor detects a cache miss and must load the data from off-chip memory. The static WCET analysis must account for this lengthy cache servicing time, even though its occurrence may be rare, resulting in hugely overestimated WCET values. Although prior work has focused on modeling the data flow through the processor to predict its state and thereby place a tighter bound on WCET [30, 31, 32, 33], the techniques remain limited and extremely challenging.

Modern languages are another enemy of tight WCET analysis. Just-in-time compilation, a common technique for improving average performance in Java virtual machines, causes the first few executions of a task to be slow while subsequent executions are fast. Static analysis must account for this variance, leading to very large estimates of WCET. Developers are faced with unsettling choices: Turning off just-in-time compilation would merely slow down every execution, while reducing the WCET to the expected (average) running time would be unsafe and defeat the very purpose of analysis.

Even without these complexities, applying static analysis to find the worst-case execution time can be a tedious and time-consuming manual effort. A typical case can be seen in Figure 2.4. In this assembly code listing for a microcontroller, the programmer has determined the WCET of the loops by looking up the specification of each instruction, inserting its execution time as a source code comment, and then manually calculating the total delay. Not only is this process inefficient and error-prone, it is completely dependent on the target processor. If the processor changes, the entire manual analysis must be redone. Programmer productivity suffers, and the

```
;ATonTn11.ASM: 440 Hz Standard Music "A" tone generator
.include "TN11def.inc"      ; Port definitions here
.cseg
.org 0
         sbi     DDRB, 1     ; config DDRB bit-1 as output
         sbi     DDRB, 0     ; config DDRB bit-0 as output

         cbi     PORTB, 0    ; LED=ON

AGAIN:
         cbi     PORTB, 1    ; PB1=LOW; 2us
         rcall   DL1132us    ;
         NOP                 ; 1us
         NOP                 ; 1us
                             ;                1136
                             ;               _______         __
         sbi     PORTB, 1    ; PB1=High     |       |       |
         rcall   DL1132us    ;             ___|       |_______|
                             ;                          1136
         rjmp    AGAIN       ; 2us
;-------------------------------------------------------------------------
;delay 1132us at 1 MHz

DL1132us:
         LDI     R20, 0           ; 1us
LOOP1:
         DEC     R20              ; 1us \ 3*256 = 768 us
         brne    LOOP1            ; 2us /

         LDI     R21, 119         ; 1us
LOOP2:
         DEC     R21              ; 1us \ 3*119 = 357 us
         brne    LOOP2            ; 2us /

         RET                      ; 4us
;-------------------------------------------------------------------------
;rcall is 3us;  Total delay = 1132 us
```

Figure 2.4: This assembly code listing from the February 2006 issue of *Nuts and Volts* magazine is an example of the tedious and error-prone process that WCET analysis sometimes requires. Here, the source code comments reveal how the programmer had to compute the delay of each individual instruction and calculate the final WCET value manually.

development of real-time systems becomes lengthy and expensive.

To combat the obvious disadvantages of the manual approach, several tools for automated WCET analysis have emerged. They perform essentially the same steps as a human but are able to cope with much larger programs and compute the results far more quickly. Error rates are also substantially reduced, assuming of course

that the tool has been thoroughly tested and debugged. Perhaps the most powerful and sophisticated of these tools is aiT [34, 35], which won a competition testing the precision of both commercial and academic static analyzers [36]. However, even a state-of-the-art tool like aiT has serious weaknesses. Virtually all such tools focus on obtaining tight bounds; the speed of computing the WCET is secondary. Developers have to interrupt their testing efforts while waiting for the tool to finish its work.

Another problem is that current tools operate at a very low level. They perform analysis of the assembly language and then simply stop. While the tool may provide a visualization of the results, as shown in Figure 2.5, there is no mapping from this assembly code back to the original source code constructs of the analyzed program. In order to understand the visualization, the user must also understand the assembly language of the target processor. If the target processor changes, the user may have to learn yet another assembly language. In addition, these tools are self-contained and exist separately from the user's source code editor, forcing a periodic switch between source code and assembly code that hampers productivity. There is little or no interaction between the WCET tool and the natural development environment.

These existing approaches to finding the WCET of a task are insufficient. Measurement is unsafe; manual analysis is time-consuming and error-prone; and automated analyzers are unduly low-level and lack integration with traditional programming tools. Real-time systems are thus notorious for requiring multiple cycles of refinement. After the initial design and implementation phases, there often comes a lengthy and expensive cycle of testing, refinement, re-testing, and so on until performance is adequate. The assumption is that with sufficient testing, the system will be both correct and predictable, but this assumption is misleading. In the words of Steve McConnell [37]:

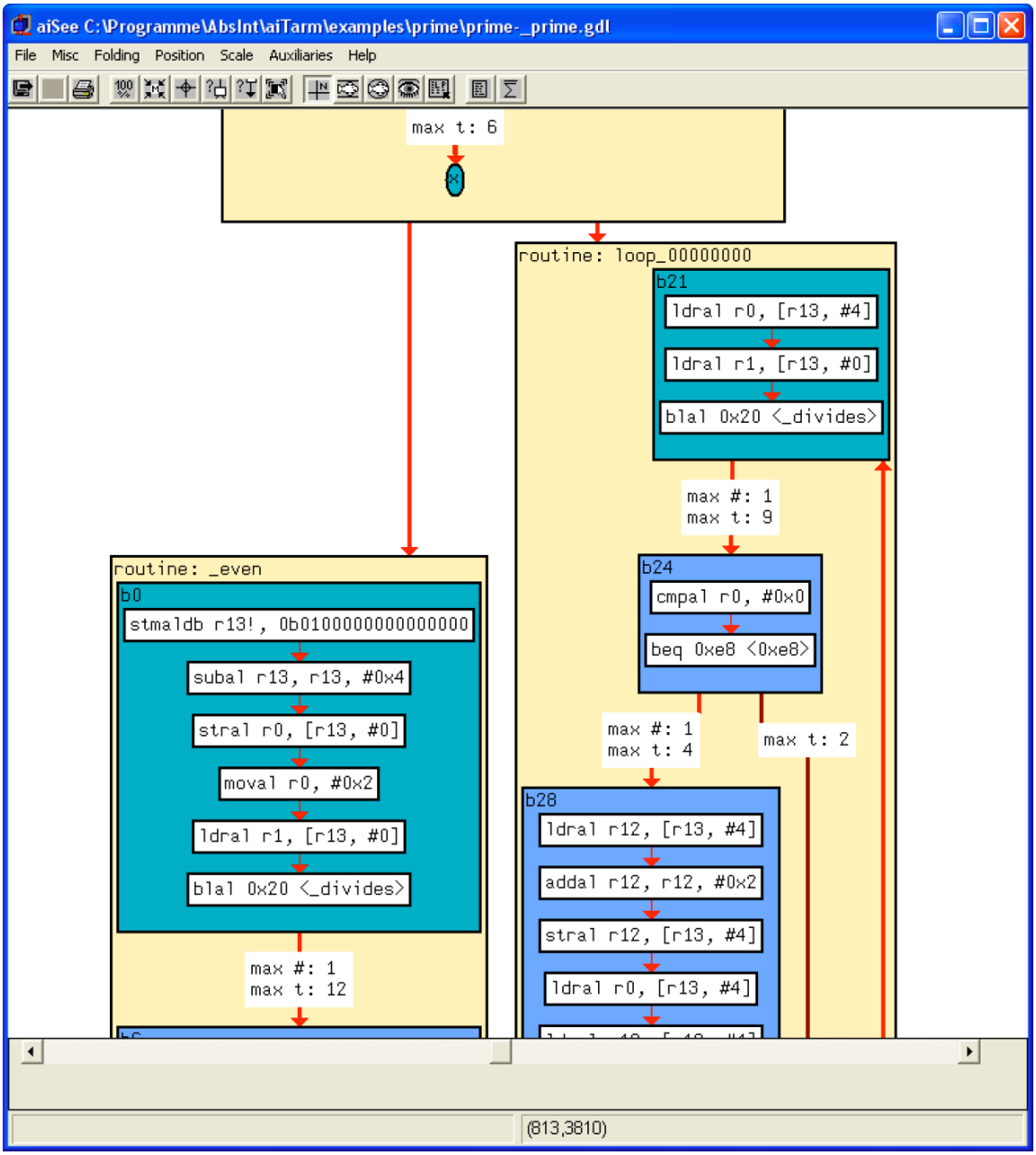

Figure 2.5: One of the most advanced tools for automated WCET analysis is aiT. Despite its sophistication, aiT exposes too much implementation detail, such as the source code disassembly shown here. (Screenshot by AbsInt Angewandte Informatik GmbH.)

Testing by itself does not improve software quality. Test results are an indicator of quality, but in and of themselves, they don't improve it. Trying to improve software quality by increasing the amount of testing is like

trying to lose weight by weighing yourself more often. What you eat before you step onto the scale determines how much you will weigh, and the software development techniques you use determine how many errors testing will find. If you want to lose weight, don't buy a new scale; change your diet. If you want to improve your software, don't test more; develop better.

2.3 What Is Interactive Analysis?

To develop better requires a fundamental change in the way real-time systems are constructed. Thus far, virtually all approaches to real-time system development have been backward. The tendency is to take existing, well-understood paradigms from the non-real-time domain, then gradually tighten and restrict them until real-time constraints are satisfied. For example, much effort has been expended trying to shoehorn traditional event-based programming models, which were intended for flexibility, not predictability, into the real-time environment. The end result is a system nearly impossible to analyze and lacking any guarantees on timeliness. The task is somewhat like trying to fit a square peg into a round hole.

For hard real-time systems, the opposite approach is more appropriate. Software design methods should begin with the principle that timeliness is the prime quality. If a system fails to respond to external events in a timely fashion, then everything else about it—cost, functional correctness, or overall performance—matters little. Therefore, the system should begin as a simple, predictable, completely analyzable design and only then may it be expanded and the timing constraints relaxed in order to meet cost and performance requirements.

The intent is not only to design systems that work correctly and responsively but also to show that they indeed do, given the serious consequences of malfunctioning hard real-time systems. Therefore, the major emphasis should be on techniques for validation, demonstrating that the system in question has the proper timing behavior. The end goal is to prove that it meets its temporal objectives in all circumstances.

Undoubtedly, the concept of validation is nothing new. Hard real-time systems, especially the safety-critical variety, undergo extensive testing and analysis before deployment. Industry standard guidelines, such as the DO-178B for avionics software [38], include rigorous assessment procedures to ensure that a system behaves according to its requirements. Despite these efforts, timing analysis routinely occurs only after implementation is complete. Potential design flaws related to timing, such as an overly high WCET, may not be revealed until well into the development cycle.

A new approach, encapsulated by the phrase *interactive analysis*, is a way of preventing such problems. The novelty of this approach is in making timeliness a fundamental aspect of system design from its very inception. Instead of waiting until implementation is complete before starting timing analysis, the intent is to incorporate knowledge of worst-case time into every facet of software development. WCET analysis must be performed *continuously* and *interactively* from the moment the first line of code is written.

In this sense, the interactive analysis approach is analogous to the Extreme Programming (XP) method [39], which advocates continuous testing and integration as opposed to a "Big Design Up Front" or a waterfall approach. The key difference is that XP focuses on functional correctness, whereas interactive analysis centers on temporal correctness. The premise is that temporal correctness is often harder to achieve and verify, and therefore it should be established at the very beginning of development and maintained throughout.

This philosophy can be boiled down into three basic tenets of interactive analysis:

Bug prevention over bug detection The conventional software development process is 1) design, 2) implement, 3) test. This life cycle divorces the validation process from the implementation phase, making it an exercise in bug *detection*. In contrast, interactive analysis marries the implementation and timing analysis into a single stage, leading to a kind of bug *prevention*. By incorporating static WCET analysis into the coding process, timing bugs caused by missed deadlines are guaranteed not to occur. (In other words, heisenbugs are transformed into bohrbugs.) This practice of correctness-by-construction has been championed by many researchers as a means of developing safer, cheaper, and more reliable real-time systems through early detection and removal of timing errors [40]. As a case in point, an avionics project reported a four-fold increase in productivity and a ten-fold improvement in quality by adopting unambiguous programming languages that focus on preventing bugs early rather than detecting them in a subsequent validation step [41].

Interactivity The typical strategy for real-time system development is inherently an open-loop control flow. There is no feedback during the implementation phase to indicate whether the code is meeting its timing constraints. Recent studies have criticized this state of affairs, noting that current WCET tools demand too much manual intervention [42]. By integrating automatic WCET analysis into the implementation process, the loop closes and development of real-time systems becomes *interactive*, as shown in Figure 2.6 The intent is to give the programmer knowledge of the worst-case time *as the code is written* and thereby help to identify timing problems the moment they are created. Furthermore, if the implementation must later change to fix a bug or add new behavior, the impact of the change on worst-time time can be seen instantly, without having

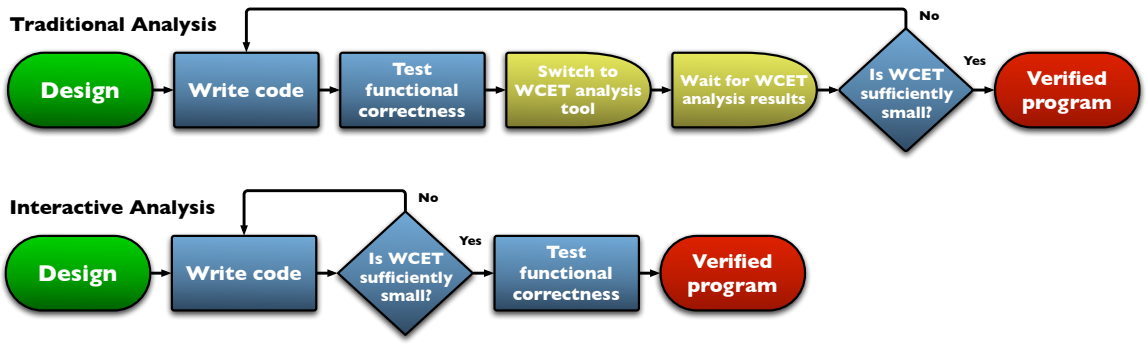

Figure 2.6: In the traditional approach to WCET analysis, the developer's feedback loop is large due to the switching to and from a separate tool that may require minutes to finish its job. Interactive analysis tightens this loop and shortens development time by making the WCET tool much faster and integrating it into the development environment.

to perform an additional and separate validation step.

High-level languages The same studies that noted the lack of a "one-click analysis" approach in current WCET tools also noted that these tools require too much detailed knowledge of the analyzed code. As illustrated in Figure 2.5, even the most advanced WCET analyzers do little to shield the developer from the minutiae of the underlying hardware. Interactive analysis requires that real-time software be developed entirely in modern, high-level, mainstream languages. The current practice of implementing real-time systems in low-level languages, such as assembly code and C, is inefficient and unscalable. Newer languages, such as Java and C#, were designed for programmer productivity while also offering safety features, most notably stricter type checking, that simplify validation of real-time systems. These languages are more suitable for the goals of interactive analysis.

These ideas are essentially a synthesis of prior research in the real-time computing arena. Specifically, their genesis was inspired by the work of four individuals:

K.H. (Kane) Kim The TMO programming scheme, which Kim invented, shares

many of the goals of interactive analysis. In particular, TMO relies on knowing WCET for event triggering, and it makes ease of analysis a prime consideration. In addition, it addresses design-time guarantees of timeliness, corresponding to the interactive analysis mantra of bug prevention over bug detection. It is also object-oriented and based primarily on C++, which is in line with the high-level language requirement in interactive analysis. In fact, the greatest motivation for interactive analysis came from one of Kim's aphorisms: "The difference between an amateur and a professional real-time programmer is this: The amateur says, 'I'm convinced my system is real-time.' The professional says, 'I *guarantee* my system is real-time.'"

Hermann Kopetz As the creator of the Time-Triggered Architecture (TTA) [43], Kopetz is an advocate of guaranteed timeliness in real-time systems by means of precise specification. The TTA approach requires that all interactions between distributed components be fully specified, not only in the functional domain but in the temporal domain as well. This concept, which makes the notion of time a crucial ingredient of the system, is tantamount to the idea of interactive analysis. In addition, Kopetz's book, *Real-Time Systems: Design Principles for Distributed Embedded Applications* [6], stressed the importance of global time, time-triggered protocols, and similar concepts that guided the formation of this work.

Edward A. Lee A strange reality in computing is that the underlying electronic hardware of digital logic delivers precise timing accuracy, while the overlaying software abstractions discard it. The Alan Turing model, which forms the basis of nearly all computers and programming languages, makes time irrelevant. It does not exist in the semantics of programs, and one must step outside the language to specify timing. Timing then becomes a consequence of implemen-

tation and not a property of design. The resulting systems are brittle; small changes have big consequences; and porting real-time code to new platforms requires a redesign. In making these observations, Lee suggests that the fundamental model of computing needs to be reinvented [44]. For example, the formal definition of computing is currently stated as:

$f : \{0,1\}^* \rightarrow \{0,1\}^*$

In other words, computing is in essence a function that transforms a sequence of bits into some other sequence of bits. Lee proposes changing this formula to the following:

$f : [T \rightarrow \{0,1\}^*]^P \rightarrow [T \rightarrow \{0,1\}^*]^P$

Here, the function is still transforming a sequence of bits, but it is doing so within a specific period of time. Lee notes that altering the model impacts nearly every layer of computing, but such a revolution is necessary when computation must be absolutely, positively on time. The notion of interactive analysis, which shares the same goal of elevating time to the level of the programmer's model, is one of many ideas that may help make this transformation possible.

Peter Puschner Perhaps the most prolific author on the problem of worst-case execution time, Puschner has a long history in WCET research. His contributions include not only scientific advancements, such as linear programming techniques [45], but also position papers that help debunk myths [46] and explain why the industry has been slow to adopt WCET analysis [47]. More recently, Puschner has been promoting *WCET-oriented programming*, a general technique for rewriting an algorithm to improve its worst-case running time [48]. For example, the standard bubblesort algorithm can be rewritten to avoid input data dependencies and always execute the full number of loop iterations. Although the resulting algorithm will look quite unconventional and likely will

not perform as well overall, it will keep the WCET tight.

Interactive analysis is both a refinement and extension of this idea. Whereas WCET-oriented programming is a manual strategy for algorithm design, interactive analysis offers an automated, systematic approach for achieving similar goals. Both techniques recognize the inherent weaknesses in static analysis, such as overly pessimistic estimates of WCET, but they differ in how to handle them. Interactive analysis puts the human operator "in the loop" so that potential WCET problems can be identified and acted upon, while WCET-oriented programming seeks to avoid the problems entirely.

To be sure, the goal of interactive analysis is both challenging and ambitious. With static analysis, the predicted WCET can be thousands of times larger than actual worst-case time. It computes only the upper bounds for WCET, not necessarily the exact WCET, and thus it can be overly conservative. Yet this drawback only underlines the need to integrate time into the development cycle. Just as object-oriented programming demanded a sacrifice in performance in exchange for easier maintenance, interactive analysis also impacts average-case performance, but in return it provides strong guarantees that a system will behave as expected. This should be a welcome change for real-time programmers who traditionally have no peace of mind as they worry whether their code will meet its deadlines:

> *When I worked with microcontrollers (fairly hefty ones), in actual practice I never lost any sleep over pointer correctness. However, I did sweat bullets over real-time response in my nested interrupt handlers.*[3]

[3]Excerpted from an online discussion of the tradeoffs involved in programming language design: `http://it.slashdot.org/comments.pl?sid=209924&cid=17110228`

2.4 Research Objectives and Contributions

Research papers describing computer systems and tools often exhibit a disconnect between what is proposed and what has been implemented. This is unfortunate, given that an implementation provides compelling evidence that the design of a complex tool or system is valid. Some papers do not even claim an implementation at all, and those that do may not provide source code for study and enhancement by other researchers.

Conversely, all of the ideas and techniques proposed in this dissertation have been implemented, at least at an experimental or prototype level. The intent is to show the viability of the interactive analysis approach. Specifically, the source code for this implementation has been pooled into a software project called Volta.[4] Every line of code in Volta is published under an open-source license, inviting critical comparison and making the reported experiments repeatable.[5] The suite of tools in Volta are built in a modular, extensible fashion to facilitate future research.

Volta is a substantial contribution because it demonstrates a novel end-to-end technique for constructing analyzable real-time systems with guaranteed predictability. Its techniques have the potential to become a standard part of the real-time developer's toolbox. Volta can also serve as an educational tool due to its ability to operate entirely within the implementation language, shielding students from low-level details such as assembly code. It thus lowers the barrier of entry for novice real-time developers, many of whom are unaware of the need for static analysis. As others have noted [49], academia should integrate tools like Volta into the curriculum so that students of today—and ultimately the real-time system engineers of tomorrow—understand the benefits of guaranteed worst-case execution time.

[4] Named after the lake in Ghana, not the physicist from Italy.

[5] Volta is available online at `http://volta.sourceforge.net/`

Most research projects must make certain simplifying assumptions in order to achieve their objectives, and the Volta tools are no different. To meet the goals of interactive analysis, the following statements about real-time developers are assumed to be true:

- They are more concerned with guarantees on worst-case execution time than with the average performance of the system. In other words, they would rather choose a factor of ten slowdown in all operations than a factor of one hundred slowdown that happens very rarely but at unpredictable moments.
- They are willing to deploy the system exclusively on specialized hardware.
- They are willing to implement the entire system in a single language.
- They are unconcerned with code size, power consumption, and other non-temporal aspects of the system.

While future enhancements to the Volta suite may relax these requirements, in their current form they may seem exceedingly strict, particularly the need for special hardware. In the closed environment of a tightly controlled real-time system, however, such assumptions are realistic. They provide a reasonable tradeoff between analysis results and analysis efforts. By making these assumptions, Volta helps developers *guarantee* the predictability of real-time programs, which is at least as important as helping them run quickly.

The following chapters explain the elements of Volta and how it enables interactive analysis. This is a tricky task, given that the architecture of Volta is somewhat complex because it touches so many areas of computing: processor instructions, control flow analysis, scheduling, shared libraries, and more. In contrast, traditional research in real-time systems (see Figure 2.7) tends to focus on one particular aspect with little or no attempt at integration with other research. Volta is different in that

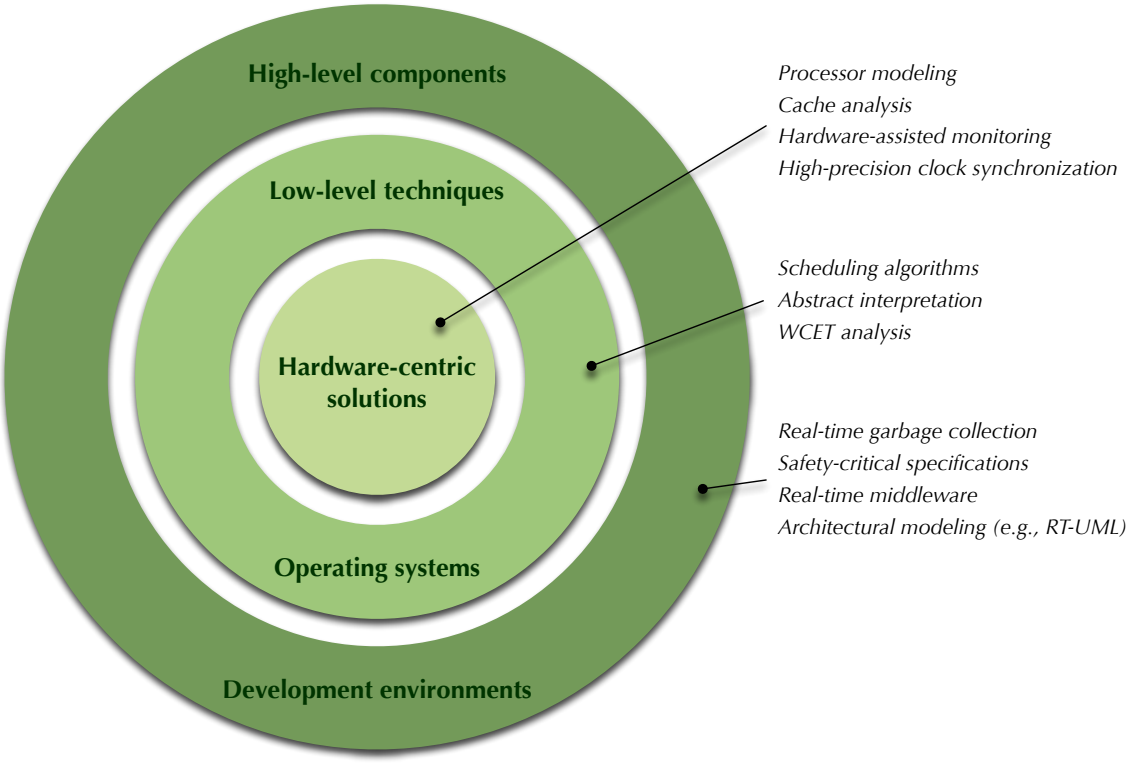

Figure 2.7: Research in real-time systems tends toward stratification, as indicated by the rings of this diagram. (Larger rings indicate higher-level techniques.) Solutions are often proposed, implemented, and verified in isolation with little or no regard to integration with other solutions. In contrast, the Volta project fills the gaps between the separate layers, merging them into an integrated platform for interactive analysis, as illustrated in Figure 2.8.

it integrates a vertical stack of abstractions tied together by the common theme of interactive analysis. Each layer builds upon the services provided by the one underneath, as shown in Figure 2.8. The job of each layer is to simplify the design and construction of the layer above it without sacrificing hard real-time predictability.

To explain Volta more clearly, each chapter is dedicated to one specific layer of the Volta stack. Their order proceeds in a bottom-up fashion, starting with the fundamental hardware and software requirements in Chapter 3. Chapter 4 focuses on control flow analysis, and Chapters 5 and 6 show how the control flow is traversed to compute worst-case execution time. Chapter 7 continues the journey upward along the Volta stack by showing how shared libraries can be made interactively analyzable

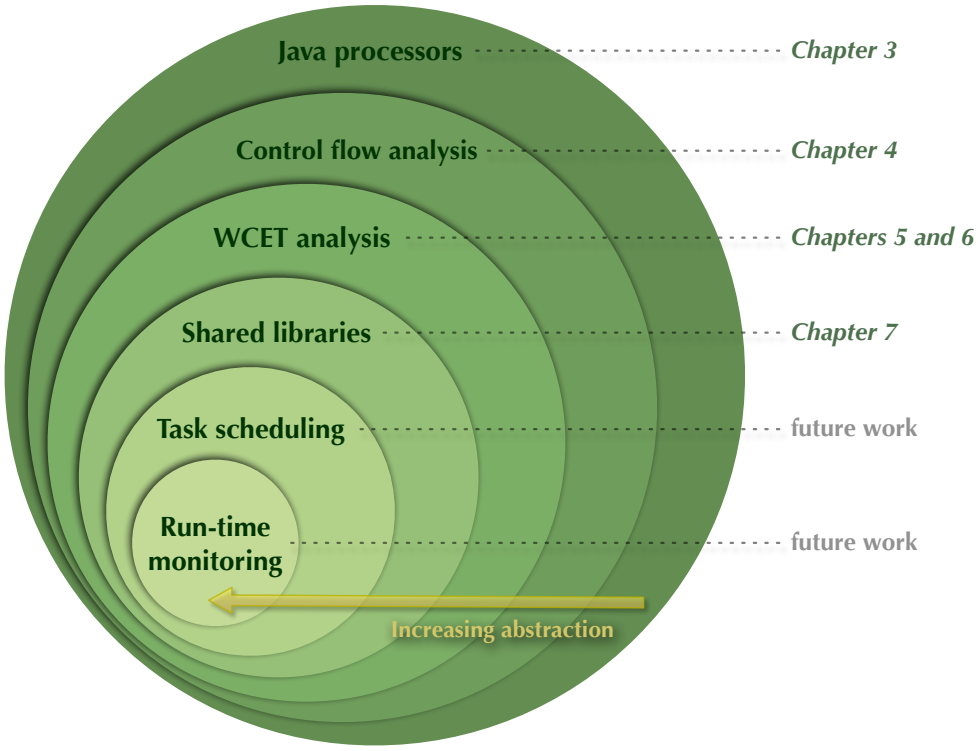

Figure 2.8: The remaining chapters explain the Volta project in a bottom-up fashion, starting with the low-level hardware assumptions and ending with a real-world example of interactive analysis.

for WCET.

In future work that is planned but not complete, Volta will include an interactive WCET-aware static scheduler, as well as facilities for run-time observation of real-time systems in a non-intrusive way.

Chapter 3

Hardware and Software Requirements for Interactive Analysis

Given the potential benefits of interactive analysis, the essential question is why this paradigm is not already commonplace despite two decades of research in worst-case execution time. One explanation comes from Kirner and Puschner [46], who argue that industrial-strength WCET tools are simply too difficult to implement, largely due to the increasing complexity of modern processors. For example, branch prediction hardware normally results in a fast and efficient processor pipeline, but when a prediction misses, the pipeline stalls. The result is a huge variance between optimal and worst-case performance.

Handling this kind of variance demands one of two approaches: an extremely complicated WCET analysis tool (including data flow analysis and a complete model of the pipeline) or a very conservative WCET bound that cripples the performance of the

system. Both options are unattractive to industry, leading to a neglect of the proper WCET analysis that is necessary for safe and reliable real-time systems.

As a compromise, current WCET tools expose the underlying complexities of a given hardware architecture. They routinely ignore high-level source code altogether, operating only at the machine code level. Mapping this machine code back to the original source code—a fundamental requirement of interactive analysis—is typically a cumbersome and unintuitive manual process. The aiT tool [35], for instance, displays its analysis results in assembly language, leaving the programmer to mentally map the jumble of mnemonics and hexadecimal numbers back to the source code language of choice.

While these problems may ultimately be unavoidable given the intricacies of modern hardware, this does not mean interactive analysis can never become a reality. To overcome the current limitations and bring about a new generation of interactive analysis tools, certain restrictions in the hardware and software foundations of a real-time system can be adopted. Making a few realistic, carefully chosen assumptions about the system's implementation can transform interactive analysis into a tractable goal.

3.1 The Trouble with C

The first step is to abandon the conventional wisdom that real-time systems must be programmed in C. Today, this language is still the predominant choice for hard real-time systems [50], not only in legacy code but in new projects as well. For example, the designers of Boeing's Unmanned Little Bird, a nascent research project for developing autonomous unmanned helicopter control, chose C for all of the project's in-the-air

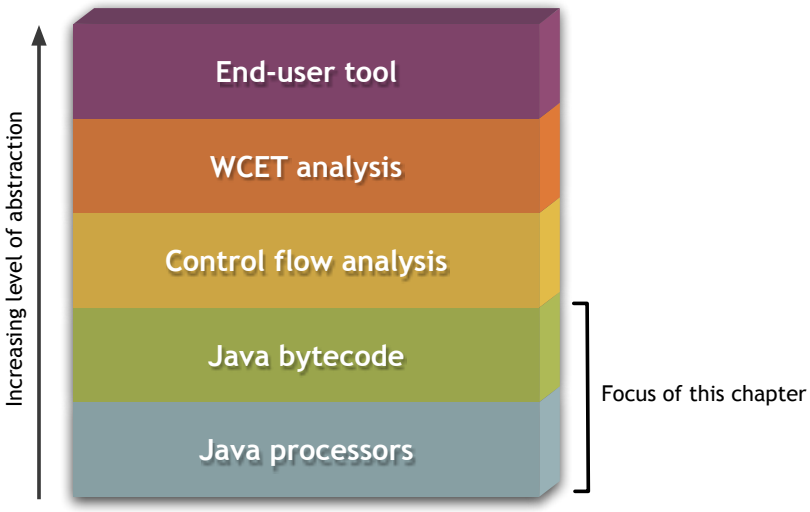

Figure 3.1: To reduce complexity to a manageable level, interactive analysis requires breaking down WCET analysis into the layers of abstraction shown here. At the foundation lies the simplifying assumption of a Java processor running Java bytecode, which is the focus of this chapter.

code [51]. Boeing is not alone; 68% of embedded systems developers are also using C [52].

At first glance, the reasons for choosing C are clear. It offers several desirable characteristics for real-time applications:

- Code portability and efficiency
- Ability to access specific hardware addresses
- Low runtime demand on system resources

These features give C a degree of control over the performance and timeliness of the system that is unmatched by almost any other language. In exchange for this control, however, C sacrifices a vital criterion for interactivity: ease of analysis. C is simply a poor choice when it comes to ensuring correctness.

The problem can be traced back to the birth of C in 1972. As the popularity of

the language grew, so too did the number of C compilers, each of which parsed C constructs in slightly different—and sometimes contradictory—ways. Subsequent efforts to standardize the language focused more on allowing compatibility across these compilers than on limiting the number of ways to interpret a given C expression. As a result, even the most recent C standards documents often surrender to the phrase "undefined behavior," a reconciliation in deciding how to interpret the numerous corner cases of the language. While this approach helped cement the popularity of C, it also hindered the ability of static analysis to verify the language's semantics. As Dennis Ritchie, co-inventor of C, would later remark: "C is quirky, flawed, and an enormous success." [53]

This is not to say that writing time-critical code in C is impossible. With good habits and due diligence, experienced programmers can produce high-quality real-time systems in C. The trouble arises when trying to *verify* the timeliness of these systems. (Even the most experienced programmers make mistakes.) Yet many of C's fundamental features, such as pointer aliasing and global variables, make static analysis excessively difficult. In addition, certain ancillary features of the language—pre-compiled headers, pre-processor directives, inline assembly code, and so on—complicate the verification process even further. Vendors of static analysis tools have been known to swap war stories of battles against the sheer number of unusual extensions and flavors of the various C compilers [54]. A quote attributed to Bill Joy, co-founder of Sun Microsystems, sums it up best:

> *You can't prove anything about a program written in C. It's really just Peek and Poke with some syntactic sugar.*

The loose specification of the C language is not the only obstacle. The lack of a standard intermediate representation (IR) is another reason for the absence of inter-

active analysis in modern tools. Because C compilers vary, there is no common IR for static analysis tools to target. GCC's Register Transfer Language, for example, is incompatible with the Intel compiler's IR. Furthermore, these IRs may change across compiler versions; they may suffer from limited documentation (if any); and they are subject to change with each new compiler version.

This changing nature of IR magnifies the difficulty of building analysis tools. Because there is no clean, consistent separation between high-level source code and low-level machine code, the tools must be able to perform a complete top-to-bottom analysis. This includes parsing source code, constructing a control flow graph, mapping the basic blocks to machine code, analyzing each basic block according to a model of the target processor, and so on.

In short, while there is no formal proof that C is inadequate for implementing hard real-time systems, there is an abundance of circumstantial and anecdotal evidence. One of the more telling clues is the remarkable proportion to which legal C code can be obfuscated. Of course, code in any language can be obfuscated, but the severity and ease at which C can be twisted is unique, as proven each year by Binghamton University's Underhanded C Contest and the International Obfuscated C Code Contest. Even practical constructs such as Duff's Device are disturbing when considering their possible presence in safety-critical systems. Verifying the correctness and timeliness of a language that allows such expressions requires colossal effort and sometimes may not even be possible. Indeed, some experts claim with a mixture of sincerity and levity that C is not a "human-readable" language at all. As noted by Bertrand Meyer [55]:

> *The belief is still widespread, in the computing community, that C and its derivatives are programming languages—languages intended for people to*

write programs in. This is a regrettable misunderstanding, as anyone who has looked at the syntax of C will testify. C use by humans is problematic at any speed.

One of the unnamed derivatives to which Meyer refers is C++, a direct descendant of C and an increasingly popular choice for programming real-time systems. For instance, C++ is the basis of real-time middleware research projects such as TMO [28] and TAO [56]. But because C++ is a superset of C, it shares many of its parent's problems. In fact, the additional features of C++, including exception handling, polymorphism, and operator overloading, make analysis and verification of this language even more challenging. In the words of its creator, Bjarne Stroustrup [57]:

C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.

3.2 Java as a Catalyst

All of these complications, at both the hardware and software levels, combine to make WCET analysis tools complex, non-portable, and difficult to implement and to use. It is no wonder, then, that the industry has been slow to adopt the idea of WCET analysis. This situation has led developers, as well as researchers in the WCET field, to seek a better platform on which to build real-time systems and tools.

The platform that an increasing number of developers and researchers are turning to is Java. This much newer language offers direct benefits over C and C++: Compilers for Java catch many errors that C compilers miss; the language definition specifically addresses safety and security issues; and the high-level nature of Java makes it more

productive, maintainable, and portable than C [58].

Java of course is not magical, but it has the virtue of about twenty-five years of research and technology advancements. When C first appeared, networking hardly existed. Object-oriented programming was unknown. Systems with a few dozen kilobytes of memory were considered large. All this changed by the time Java emerged. Overall, Java is a simpler language than C, and simplicity, as the renowned computer scientist C.A.R. Hoare would say, is a prerequisite for reliability [59].

In particular, Java offers the following advantages over C and, to a lesser extent, C++:

Productivity Real-time systems are becoming increasingly complex and widespread. To keep up with the growing demand for these sophisticated systems, developers need to become more productive, and Java is recognized as a more productive language than C. A recent experiment by Nortel Networks found that programmer productivity doubled after switching to real-time Java [60]. In another case, Intel used Java to develop a fault-tolerant real-time distributed computing demonstration, claiming that the same project would have taken three months had C been used instead [61]. Java's object-oriented underpinnings also make it more maintainable and reusable than the flat procedural paradigm of C, especially in the context of large systems.

Training Technical aspects are not the only consideration when choosing Java for real-time systems. The state of the workforce should also be taken into account. With Java now the dominant language in university computer science curricula, most graduating software engineers are experts in Java, not C. This change of skills in the workforce is one reason why Java eclipsed C in 2001 and C++ in 2004 as the most popular language for desktop and enterprise software [62].

In the real-time and embedded space, developers are more conservative and adopt new languages more slowly, but there is an historical trend of desktop technologies migrating to embedded systems within five to seven years [50]. Unless Java can be incorporated into the development process for real-time systems, the industry will have tremendous difficulty leveraging the skills of the coming generation of software engineers.

Portability The innate portability of Java, which allows applications to run on a variety of hardware platforms with little or no change, is often cited as a primary advantage for general-purpose desktop software, but it is even more beneficial for real-time systems. Such systems are often deployed onto a diverse assortment of embedded devices with widely varying processors and instruction sets. The ability of Java to adapt readily to newer and more powerful processors, as well as to smaller processors that are less expensive and more efficient, is a distinct advantage. Many of the portability headaches from lower-level languages, such as special-purpose cross compilers and incompatible binary interfaces, are absent in Java. In addition, portability allows rapid prototyping on standard desktop workstations—even before the real-time system's hardware has been chosen—while still providing a clear path to deployment on the target device. This portability is often cited as the reason embedded real-time developers claim Java is roughly twice as productive as C++ during development of new functionality and five times less costly for maintenance and integration efforts [61].

Tool and library support One of the criticisms leveled at C is its profusion of non-standard add-on packages for serial and network I/O, forcing developers to re-learn different APIs for identical needs. In contrast, Java has had a consistent set of APIs for networking, as well as many other common tasks, since day one. It also includes built-in support for source code annotations, which are a key

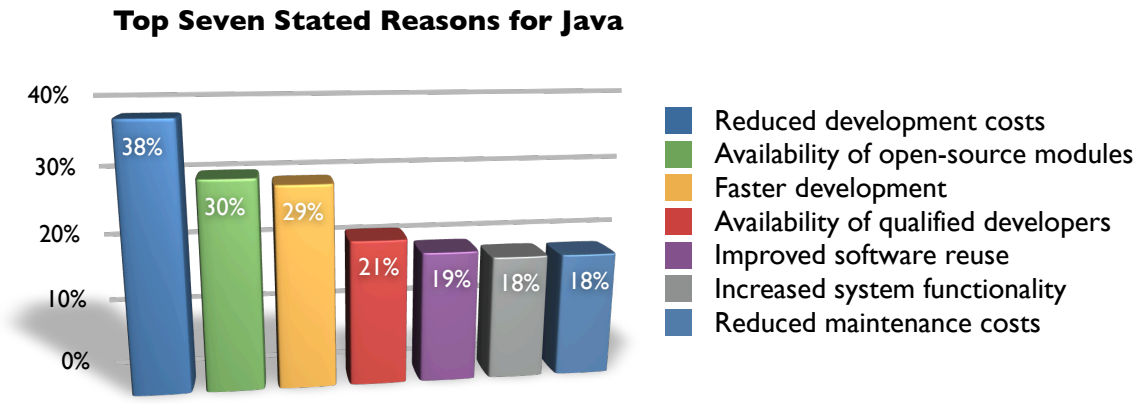

Figure 3.2: A 2005 survey asked 108 companies why they were moving embedded systems projects to Java [61]. Faster and cheaper development, as well as the availability of open-source code, were the most highly cited factors.

component of static analysis for interactivity, as described in Section 6.4.2. As a result of this stable API, there is now a thriving community of open-source and free Java software, most of which will compile and run on any platform without modification. A recent survey, illustrated in Figure 3.2, indicated that this availability of open-source tools and libraries was one of the top reasons embedded developers are switching to Java.

Data encapsulation Decades after their invention, object-oriented techniques still have a reputation among embedded and real-time system engineers as being synonymous with code bloat and sluggish performance. In recent years, however, these suspicions are being torn down by case studies that reveal less tangible benefits of object-oriented programming (OOP). An experiment in real-time flight instrument software, for instance, concluded that OOP greatly simplified communication between system and hardware engineers [63]. Both groups tend not to think in terms of function call hierarchies but in system components and their interactions: features naturally described using OOP techniques. The data encapsulation provided by OOP also encourages hiding platform details within platform-independent interfaces, which in turn enables easier debugging

and testing. Furthermore, the data hiding enforced by Java makes the job of WCET analysis easier, particularly for data flow analysis, because the number of live objects in a given state is substantially reduced.

Modularity Java's native instruction set, known as bytecode, provides an inherent modularization of static analysis tasks, as illustrated in Figure 3.3. For example, high-level WCET tools for Java can ignore any timing aspects below the bytecode level. Separate low-level tools, perhaps written by an entirely different team of programmers, can then complete the analysis once the target architecture is known. This clean separation between high- and low-level analysis helps solve the software complexity problem raised in Chapter 2. Bytecode also acts as a common, well-specified intermediate representation that does not vary with different compiler versions and vendors, unlike the situation with C. WCET analysis tools can standardize on this common IR, allowing for their interoperability [64].

In spite of these advantages, arguing that Java is better than C may be like arguing that grasshoppers taste better than tree bark.[1] Bruce Boyes, founder of embedded Java provider Systronix, notes that Java is certainly not a panacea for all programming problems [65]:

> *Sloppy programmers can circumvent many of Java's safety features and write bad code in Java. The programmer is still the most important link in the software development chain (and it's unlikely that any computer language or development tool will ever completely replace the human brain).*

Indeed, Java appears at first glance to be a terrible match for real-time systems. Its

[1] This delicious analogy comes from a post by Thant Tessman to the comp.lang.lisp Usenet group on June 15, 2000.

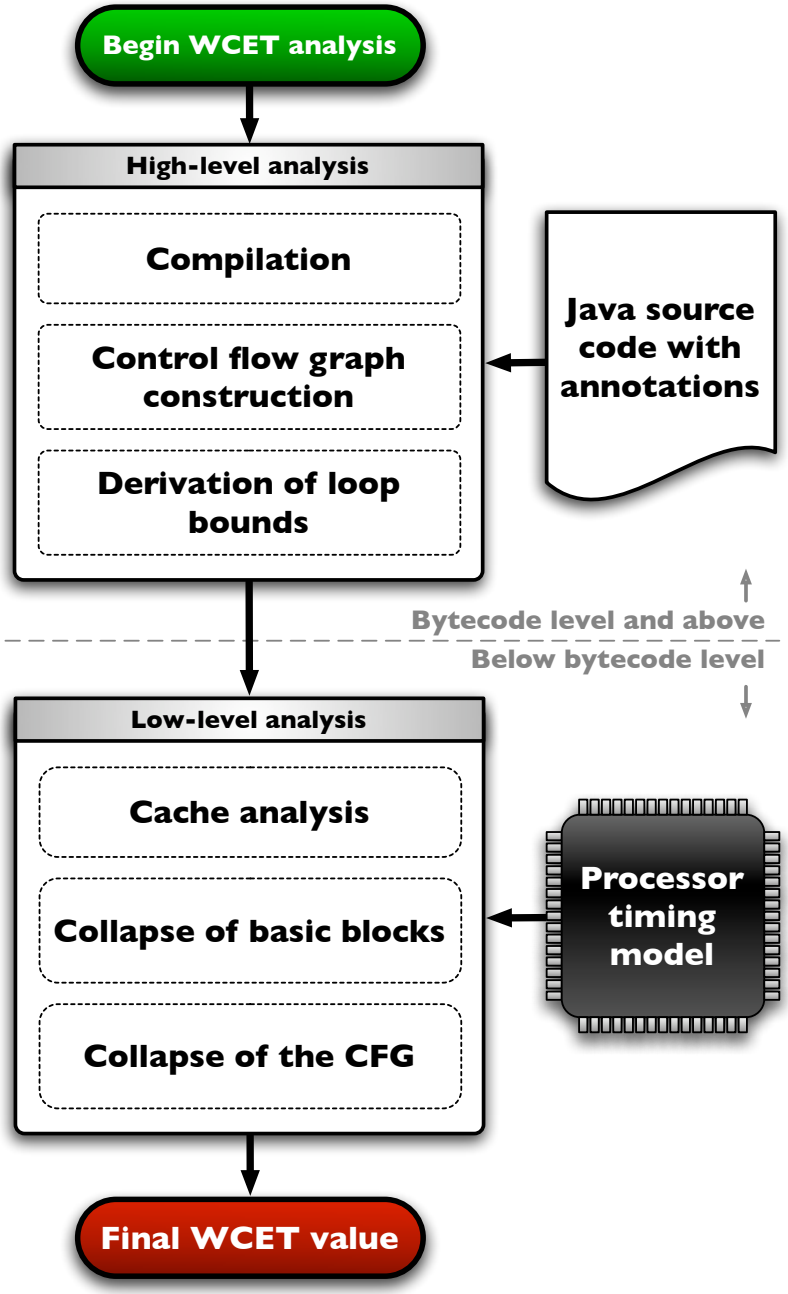

Figure 3.3: This sketch of the WCET analysis process shows the clean separation between high- and low-level analysis that Java bytecode provides. With bytecode as a common intermediate representation, the two levels become independent: High-level analysis can ignore the CPU timing model, while low-level analysis need not construct a CFG or parse annotations. This separation of tasks makes WCET analysis tools simpler, more modular, and more interoperable.

combination of automatic garbage collection, underspecified threading semantics, and the complexities of object-orientation are all impediments to building time-predictable software. Certain run-time characteristics of Java, such as a high frequency of method invocation and dynamic loading, make Java more difficult than other languages for conducting WCET analysis. In particular, dynamic dispatch—the mechanism upon which OOP is based—is still very much an open problem. Although prior work has confronted the issue [66, 67], it remains unsolved for the case of arbitrary object-oriented code.

Java can also be a voracious consumer of memory compared to C, and it is often criticized as being slow and inefficient. Java does indeed take more CPU cycles to execute the average line of code than C, but the extra overhead can be justified by other benefits that Java offers. As a real-time system grows in size and complexity, raw throughput tends to be less important than overall productivity and maintenance issues, both of which are Java's strengths. One must also consider that C has none of Java's built-in safety features, such as exception handling, array bounds checking, and automatic memory management. If C incorporated these features, much of its speed advantage would evaporate.

Furthermore, Java compilers, interpreters, and virtual machines have closed the performance gap in recent years. Well-written Java code running on an adaptive optimizing engine such as HotSpot [68] can now outperform C in some instances. When comparing C++, the performance difference virtually disappears, especially for large, complex applications. Garbage collection preemption latencies have been reduced to 100 microseconds or less [61], closing the gap even further. While there are still cases where memory and execution constraints may prevent the use of Java, such as very small embedded systems, this is the exception rather than the rule.

To be sure, hard real-time and safety-critical applications may not yet be ready to

jump to Java. For that to happen, Java must prove itself in rigorous implementations that meet the demands of mission-critical applications. Much research still remains, and new analysis techniques must be devised. Existing analysis tools will have to be rewritten for Java, as many are currently grounded in C.

Despite these obstacles, Java has the potential to provide safer, cleaner, and higher-quality real-time software. It is not merely a newer language; it is an enabler. It enables new ways of developing and analyzing code that were not possible before. The common theme here is that Java offers a higher level of abstraction, a crucial ingredient for reducing the complexity of real-time systems to a point where temporal analysis becomes viable and effective. Java is therefore a catalyst in making interactive analysis possible.

3.3 Java in Real-Time Systems

For a language that offers no temporal guarantees and is designed to be dynamic, positioning Java as a solution to real-time programming problems is counter-intuitive. Its many sources of unpredictability—garbage collection, dynamic class loading, just-in-time compilation, and more—inhabit the worst nightmares of real-time developers. The phrase "real-time Java" may even sound like an oxymoron, yet it continues to gain traction among researchers looking for new solutions to old real-time problems.

The renewed interest in Java for real-time systems is less surprising when examining its history. When it was first released by Sun Microsystems in 1995, it had already undergone considerable evolution as an embedded programming language, originally targeted toward set-top boxes for interactive television [69]. By the time of its release, the exploding popularity of the Internet prompted Sun to reposition Java as a tech-

nology for interactive web sites. Its relevance to embedded and real-time applications was momentarily lost.

Not long after Java's release, however, computer science professor Kelvin Nilsen published two influential papers outlining Java's potential benefits to real-time programming [70, 71]. The positive response to these publications revealed pent-up demand for the capabilities described in the papers [69], prompting Sun to launch an initiative in 1998 to make Java suitable for real-time systems. This effort resulted in a draft of the Real-Time Specification for Java (RTSJ) [72], a formal document that proposed dozens of new interfaces and behavioral specifications to make Java more time-predictable.

Then, in the year 2000, three near-simultaneous events seemed to open the floodgates for real-time Java: In May, the final draft of the RTSJ was released. In June, the first paper on applying Java to the problem of WCET analysis was published [64]. And in November, the first processor designed specifically for embedded and real-time Java, the aJ-100 [73], became commercially available.

In the years since these innovations, Java has become a viable platform for real-time systems [74]. Commercial implementations of the RTSJ are available from aicas [75] and Sun [76], while open-source projects such as jRate [77] and OVM [78] have shown promise. Real-time garbage collectors, such as Metronome [79], are also gaining steam. Java is currently an immature but increasingly credible platform in the real-time arena, as indicated by the growing number of prototype applications and new development projects. Examples include:

- Researchers at Lund University created a real-time control system in Java for the FlexPicker (see Figure 3.4), a three-armed industrial robot [80]. Java handles the feedback control loop of reading the state of the motors, calculating

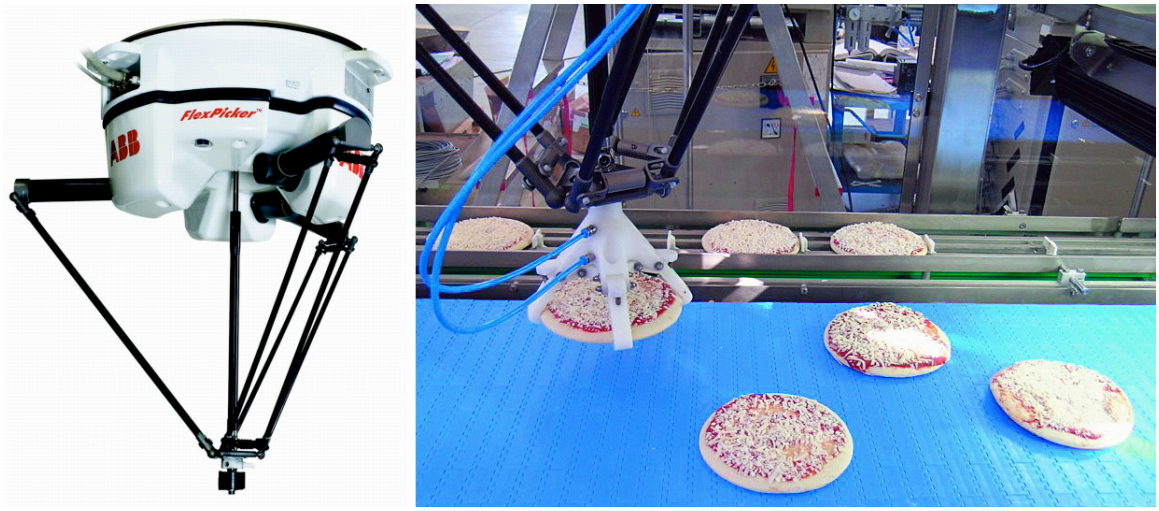

Figure 3.4: The FlexPicker, a three-armed industrial robot for pick-and-place operations, served as an experimental platform for real-time Java threads. (Photograph by The ABB Group.)

new positions, and sending new position commands one thousand times each second. Because FlexPicker is capable of accelerating at 10 g, the system is truly hard real-time: A timing error could spew carbon fiber everywhere, damaging equipment and possibly harming the operator.

- The United States Navy has been one of the earliest adopters of real-time Java technology. Lockheed Martin, for example, chose Java to handle the predictable performance aspects of the Aegis Weapons System, a ship-to-ship network for tracking and destroying enemy targets [50]. The U.S. Navy is also incorporating real-time Java into DDG 1000, a suite of middleware and infrastructure services for powering the next generation of battleships [81].

- The telecommunications industry is embracing Java for real-time applications. Nortel Networks has one million lines of Java code operating SONET fiber switch line cards, the protocols for which have strict 40-millisecond timing constraints [61]. In another case, L-3 Communications selected Java for a real-time data acquisition system [82].

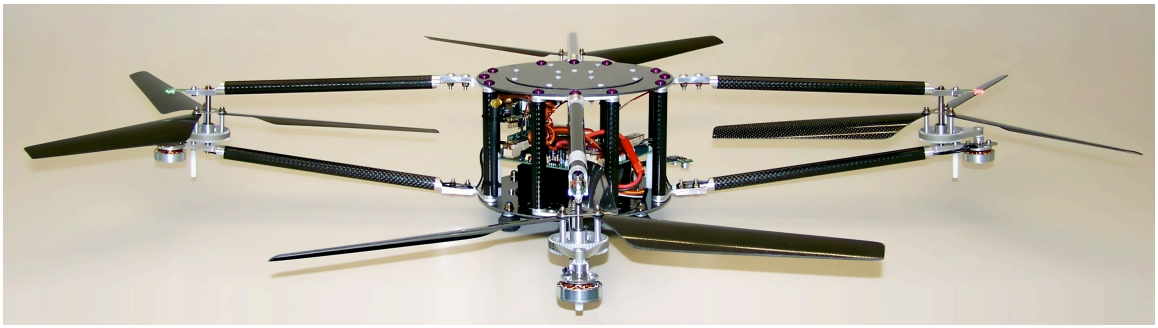

Figure 3.5: This quad-rotor helicopter was custom-built for experimenting with real-time garbage collection in Java. (Photograph by the Computational Systems Group at the University of Salzburg.)

- Java is also a popular choice for powering unmanned aerial vehicle prototypes. In the JAviator project, a real-time garbage collector for Java ensures that the gyroscopes, accelerometers, and ultrasonic sensors on a quad-rotor autonomous helicopter (see Figure 3.5) are polled at just the right frequencies to keep it in the air [83]. In a similar project, Perrone Robotics created a Java-powered helicopter that can fly autonomously and map the terrain below in 3D using a laser rangefinder [84]. Boeing is also using real-time Java to build semi-autonomous drones for the military, such as the ScanEagle airplane (see Figure 3.6) for battle damage assessment [78].

- Java is not only helping unmanned vehicles fly through the air; it also guides them through the sea. SONIA, shown in Figure 3.7, is a Java-powered autonomous underwater vehicle (AUV) designed for inspecting ocean pipelines, detecting mines, and performing other aquatic tasks [85].[2]

- General-purpose robotics is another industry where Java uptake is increasing. National Oilwell Varco is using a Java-based real-time control system to manage

[2]The student group that created SONIA entered their creation in an annual AUV competition and finished in last place in each of their first four years. After switching from C to Java for SONIA's controller implementation, the group finished in either second or third place out of about twenty other contestants for the next four years. The group credits the move to Java as a key factor in their turnaround.

Figure 3.6: For the ScanEagle, a long-endurance UAV designed for ground surveillance, Boeing installed real-time Java software on the plane to perform autonomous route planning and navigation. (Left photograph by Insitu Inc.; right photograph by Airman First Class Jonathan Snyder of the United States Air Force.)

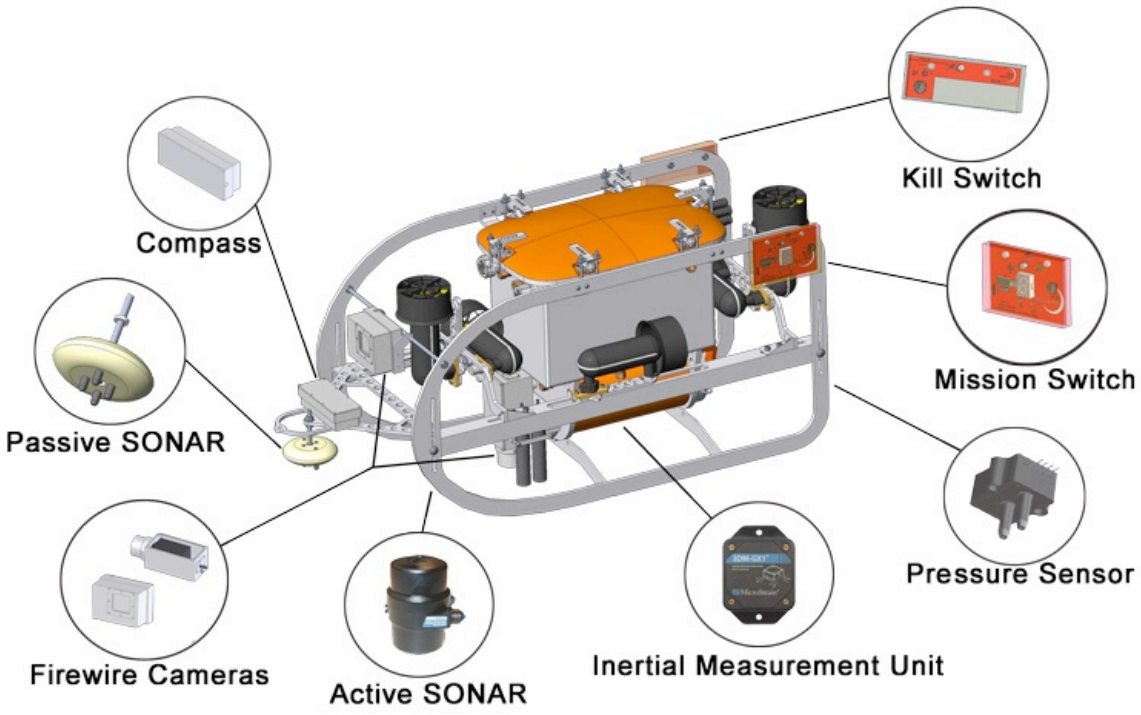

Figure 3.7: With the exception of the vision system, every component in the SONIA AUV is written in Java. (Diagram provided by the University of Quebec's École de technologie supérieure.)

automated robotic drilling [86]. The grass-roots robotics community is embracing Java, too, as evidenced by the many open-source projects available for motor control, haptics interfaces, image recognition, and so on.[3]

- In a seemingly mundane but nonetheless vital application, Java is powering traffic lights across Europe. Signalbau Huber, a vendor of city-wide traffic management systems, is porting its flagship controller software to real-time Java. A safety-critical control program specific to each intersection will enable engineers to program light behavior in Java [87].

Some of these projects rely on implementations of the RTSJ, while others use Aonix PERC, a proprietary subset of the RTSJ that extracts its real-time essence and adds some additional features geared toward safety certification requirements. Like the RTSJ, it provides ways of avoiding high-latency garbage collection, tightens the threading model to support real-time scheduling, and adds features commonly needed for embedded devices, such as direct access to memory. Both approaches sacrifice some degree of portability in exchange for these features, downgrading Java's "Write Once, Run Anywhere" mantra to "Write Once *Carefully*, Run Anywhere *Conditionally*." (This phrase was coined by Paul Bowman at a 1999 meeting of the Real-Time for Java Expert Group.)

Nonetheless, the Real-Time Specification for Java is itself a remarkable achievement, taken as a whole. Arguably the most successful attempt so far to adapt a modern high-level language to current and future real-time issues, it has encouraged many developers to consider Java for systems previously built only with C/C++, Ada, or assembly language. The general consensus among academic researchers and industry practitioners alike is that current RTSJ implementations are mature enough to be

[3]See `http://community.java.net/projects/community/robotics` for a collection of community-supported robotics source code written in Java.

used in production systems [88], at least for non-safety-critical scenarios like stock trading and investment banking. In these soft real-time applications, most problems arise solely due to the lack of real-time support [89], considering that an unplanned two-second stop for garbage collection can cause a loss of tens of thousands of dollars.

For hard real-time applications, however, the RTSJ is only a partial solution. It leaves many important issues unaddressed, and developers of hard real-time systems who are banking on the RTSJ to meet their needs will face serious obstacles. Specifically, it offers virtually no support for computing the worst-case execution time of a task, and without knowing the WCET, no guarantees can be made on the timeliness of the system. The RTSJ's only provision is in its scheduler interface—the ReleaseParameters class—which takes as input the WCET of a schedulable object. But even this basic support may be useless in practice, as evidenced by the documentation note accompanying the class:

> *Cost measurement and enforcement is an optional facility for implementations of the RTSJ.*

Because this facility is optional, many implementations choose to ignore it, thereby discarding an essential requirement of hard real-time and safety-critical systems. Thus, in spite of the deterministic scheduling, priority inversion avoidance, and predictable memory allocation of the RTSJ, it is simply not a hard real-time specification and was never intended for safety-critical applications.

Furthermore, the RTSJ is rather broad in scope, attempting to anticipate a variety of real-time scenarios and meet the needs of the largest possible cross-section of developers. As a result, implementations of the RTSJ tend to be very large and resource-hungry. The current version of the Sun Java Real-Time System [90], for instance, a popular commercial implementation of the RTSJ, requires at minimum an

UltraSparc III processor and 512 MB of RAM. For embedded real-time applications where processing power and memory are at a premium, the RTSJ is simply infeasible.

3.4 Java Microprocessors

A primary reason for the lack of a WCET analysis facility in the RTSJ, as well as other real-time frameworks, is the modern microprocessor. Architectural advancements in processor design—long pipelines, branch prediction, and complex multi-level caches—have focused on making the average case as fast as possible. Unfortunately, the shrinking of this average has not come without cost. While average execution time may be low, its standard deviation has grown large, resulting in large worst-case times. In the best case, everything proceeds smoothly: Caches are hit, operands are ready, functional units are free, branches are correctly predicted. But in the worst case, everything goes wrong: Memory loads miss the cache, functional units are busy, operands are still percolating through the pipeline, and branches are mispredicted. The span between these two cases may be several hundred cycles or, for a deep-level cache miss, many thousand. Even older and simpler processor designs, such as the 16-bit 80188 with its prefetch queue and two-stage pipeline, can cause major complications for WCET analysis [91].

Theoretically, a highly sophisticated WCET analyzer may be able to follow the flow of data through the CPU pipeline and predict when the worst-case occurs and when it does not, but this is a formidable task that is still an open area of research and virtually impossible in practice. Instead, static WCET analysis generally assumes that the worst case could occur on any given instruction, leading to hugely pessimistic worst-case time estimates. While some attempts have been made to mitigate this problem through a hybrid of static and dynamic analysis [92, 93], these approaches

are statistical in nature and provide no guarantee that the worst case will ever be tested.

In recent years, an alternative solution for dealing with over-estimation in WCET analysis has emerged. Rather than fight the increasingly hard-to-predict behavior of real-time operating systems, Java virtual machines, and modern superscalar processors, this new strategy simply eliminates them entirely. The approach relies on specialized processors that understand Java bytecode as their native instruction set.[4] These processors offer several genuine advantages for hard real-time systems:

Easier analysis By far the most important benefit of Java processors is their predictability. Running bytecode directly on the processor eliminates the need for virtual machines and just-in-time compilation, making execution time far less variable. Java's stack addressing scheme also helps reduce variability. The restrictions of the stack allow the CISC-style bytecodes of Java to be translated into RISC-style microcode that executes in a short pipeline (often just three or four stages), mitigating the need for branch prediction and the uncertainty it would introduce. Most bytecodes also have a best-case execution time that is identical to their worst-case execution time, without any pipeline dependencies between them, making low-level analysis of basic blocks as easy as summing the WCET of each bytecode. Cache analysis is simpler, as well. Jump instructions in Java are guaranteed never to target beyond the address range of the declaring method; therefore, a method-based cache [95] can ensure that every non-invocation and non-return instruction is a cache hit, vastly simplifying the timing analysis. Taken together, these characteristics yield a much tighter

[4]Strictly speaking, this approach is not limited to Java. It could be applied to C# or similar high-level, bytecode-based languages, assuming that the corresponding hardware—a "C# microprocessor" for example—is available. Some progress has already been made in this direction [94]. Without loss of generality, however, this work considers only Java-based processors, which are relatively mature and widely available.

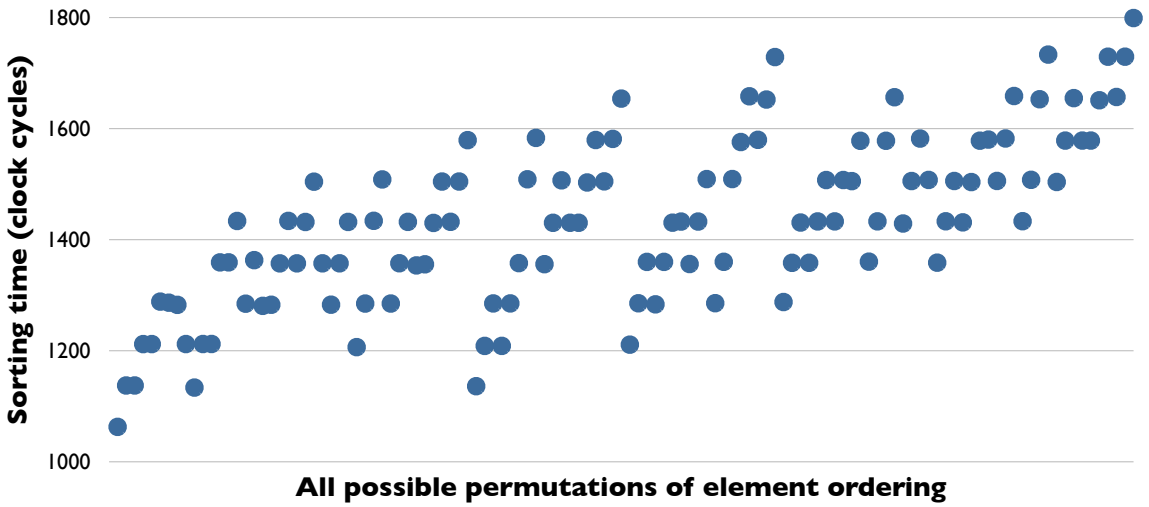

Figure 3.8: In an experiment by Schoeberl on a Java processor, the results of which are replicated in this figure, the worst-case execution times of the canonical bubble-sort algorithm were measured for each possible permutation of five elements [96]. A manual WCET analysis of the algorithm had predicted a worst-case time of 1,799 cycles, and the measured result, as shown in the graph above, was exactly the same—an optimal pessimism ratio of 0%. Analysis with general-purpose CPUs rarely achieves such a low ratio.

bound when performing static WCET analysis, as illustrated in Figure 3.8.

Speed and efficiency Recent surveys indicate that perceptions persist of Java being too big and too slow to meet real-time constraints [50]. The overhead introduced by the operating system and virtual machine not only hinders performance but also devours precious memory. Java processors, on the other hand, eliminate the need for an OS and VM, greatly reducing the memory requirements of the system. For example, current versions of the Sun Java Real-Time System require 512 MB of RAM, while a Java processor implemented on an FPGA requires 4 MB—even less if the processor is fabricated as an ASIC. Native execution of Java on these specialized processors is also extremely fast. Benchmarks have shown that a Java processor can be 500 times faster than a comparable processor running an interpreting virtual machine [97]. A side-effect of this efficiency is that Java processors can achieve deterministic real-time

performance while consuming very little power. The JStamp board, which integrates a Java processor from aJile Systems, can run on a 9-volt battery for over 40 hours [65].

Easier certification A subtler benefit of Java processors is important when considering safety- and mission-critical real-time systems. These systems must obtain certification, such as the DO-178B standard for avionics software [38], to ensure traceability from system requirements to source code. In a traditional real-time Java application, multiple layers must be traced: the operating system, the virtual machine, and the application code. When the application runs on a Java processor, however, the OS and VM layers disappear, removing tens of thousands of lines of code and making the certification process faster and cheaper. Only the Java processor and the application itself need to be certified.

The Java language Moving a real-time system from C to a Java processor brings advantages due to the nature of the Java language: strong type safety, easy portability across processors, and other benefits as outlined in Section 3.2. The processors allow the entire system to be written in 100% Java, disposing of the need for special-purpose real-time languages like wcetC [98].

These qualities make Java processors an attractive platform for hard real-time systems. Although moving to such a novel and unique architecture may seem drastic, developers have a tradition of adopting new platforms when special needs arise, as evidenced by the popularity of ARM and PowerPC architectures for embedded devices. Supporting this assumption is a questionnaire distributed to WCET tool users in 2003; it revealed that 75% of respondents would adopt a processor with more predictability even if it meant a loss in average performance [99]. Certainly, the most important feature of a processor for hard real-time systems is not how fast it can go but how much it can be slowed down by a series of unfortunate events.

Recognizing the potential of Java-specific processors, research groups and commercial vendors have created a number of different designs over the last decade. The first was Sun's picoJava [100], an experimental project in making a chip that would accelerate Java bytecode in much the same way that graphics co-processors accelerate drawing operations. Although the picoJava was never actually manufactured, it was influential as a reference platform for subsequent generations of Java processors.

Since the picoJava, a number of Java chips have come and gone. Some are not pure Java-native processors but rather hybrids—conventional processors combined with some form of Java acceleration. For example, they may have a bytecode interpreter built-in to the hardware, or they may have a customizable instruction set that allows them to understand Java bytecode. Jazelle [101, 102], Sun SPOT [103], TINI [104], Lightfoot [105], and the "asynchronous Java accelerator" [106] are instances of this hybrid approach. Such processors add additional layers of complexity that make WCET analysis difficult and therefore are not very suitable for hard real-time applications.

Other Java processors appear to be dead projects (or at least in suspended animation). Moon [107], Komodo [108, 109, 110], and Femtojava [111, 112] are apparently no longer actively developed, although each has had an impact on the evolution of Java processors. Conversely, some Java processors that have migrated from the lab to the commercial sector are clearly a success and remain quite popular. In particular, the aJile chip [73, 113] is readily available in development kits such as the JStamp from Systronix (see Figure 3.9). aJile's microprogramming includes most of the functionality specified in the RTSJ, including a priority-preemptive scheduler, a ceiling semaphore protocol, periodic threads, and a non-garbage-collected heap. More recent designs for Java processors, such as the Cjip [114, 115], LavaCORE [116], SHAP [117], jHISC [118], and BlueJEP [119], continue to shatter myths about what Java can and

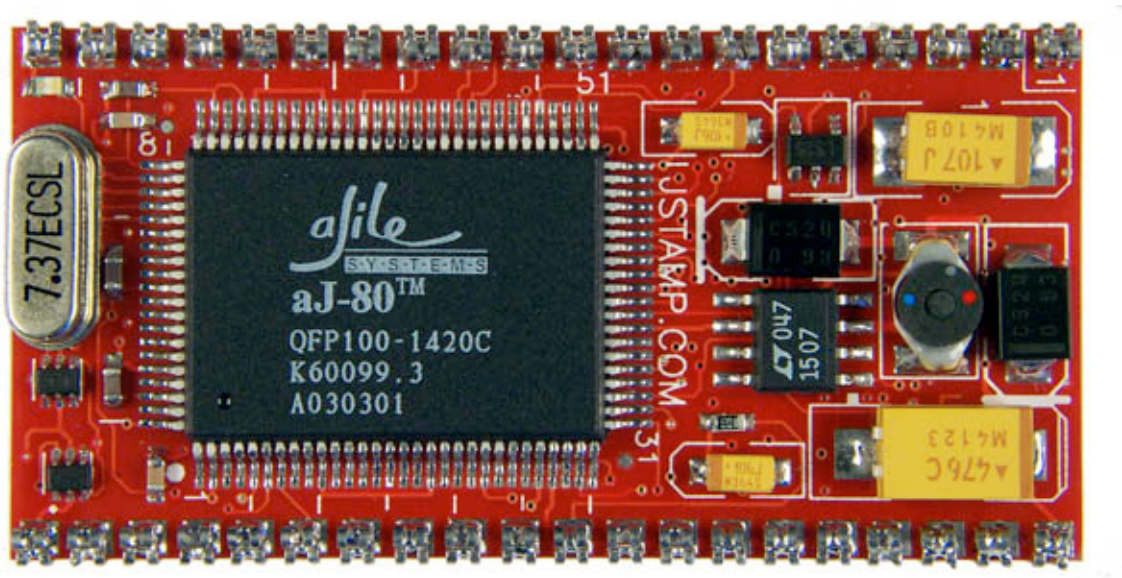

Figure 3.9: The JStamp, a Java-based analog to the popular BASIC Stamp, executes Java bytecode natively using the aJile processor. It includes built-in components often needed for real-time and embedded applications, such as serial interfaces, timers and counters, pulse-width modulation output, and so on. (Photograph by Systronix Inc.)

cannot do for real-time systems.[5]

Of all these processors, one in particular stands out, especially when considering the requirements of interactive analysis. The Java Optimized Processor, or JOP [120, 97], was designed from the ground up with predicability and ease of analysis as the primary objective. While other processors focus on making the average case fast, JOP follows a different mantra: "Minimize the worst case." It strives to enable simple and accurate WCET analysis, even at the cost of overall performance. Remarkably, this design constraint was met without resorting to unreasonable sacrifices in speed. Benchmarks show that JOP performs about as well as other Java processors when normalized for logic cell count and memory block usage.

One of the principal contributions of JOP is that it demonstrates how the unique nature of Java enables the processor's functional units to be free of time dependencies.

[5]For a treatise on the history of Java processors, including architectural details and performance comparisons, refer to Schoeberl's doctoral dissertation [120].

For example, JOP takes advantage of the fact that Java's getfield and putfield instructions never overlap with an invoke or return instruction, ruling out the possibility of bus contention between the data cache and JOP's method cache. Consequently, the large worst-case time that would otherwise exist is eliminated, and WCET analysis is much less complicated. Additional WCET-friendly features of the JOP architecture include:

- Translation from Java bytecode to JOP microcode takes exactly one cycle in all cases and is therefore predictable.
- The execution pipeline stage is kept simple; only the two topmost stack elements are available. This prevents pipeline bubbles and ensures a constant and predictable execution time for all microcode instructions.
- The data cache for local variables and the operand stack is predictable in that access to local variables is a guaranteed hit and no pipeline stalls can occur.
- Even simple processors may include an instruction prefetch buffer, but JOP's method cache and translation unit eliminate the need for such a feature, along with the variable latency it would introduce. JOP is immune to "cache chaos" [121].

The end result is a processor that is substantially easier to analyze. The cycle timings for every bytecode implemented in the hardware are listed in its documentation, so the WCET analysis of a basic block—that is, code without branches or method invocations—can be computed simply by summing the cycles.

Another advantage of JOP is that it is entirely open-source. All of the design resources, including the raw VHDL, build files, and auxiliary source code, can be downloaded for free from the JOP web site.[6] This openness helps encourage fur-

[6] `http://jopdesign.com/`

ther development, both of the JOP itself and of projects based on the JOP. For example, a loose-knit research community has been forming around the processor; it has contributed add-on projects such as a Bluetooth API [122], a speech recognition library, a VGA output module, a multiprocessor architecture, and more. The community is also working to alleviate certain weaknesses in the JOP, most notably its lack of a proper ASIC fabrication of the design, which means that any deployment of the processor must rely on expensive and relatively slow FPGAs. Currently, there are ongoing efforts to port JOP to cheaper FPGA boards, and preliminary work has begun on synthesizing JOP as an ASIC. Combined with the inevitable progress of computer technology, which has proven capable of squeezing the large workstations of today into tomorrow's mobile phones, the current speed and cost limitations of the JOP should soon dissipate. Even now, the processor is fast enough for most embedded real-time applications, as it has already been used successfully in various industrial projects: a railway communication device, a remote data logging system, and other real-world settings.

Given all of these advantages, JOP makes an ideal platform for demonstrating the capabilities of interactive analysis. All subsequent discussion will therefore assume that JOP is the processor in use. Theoretically, almost any other Java processor could also support interactive analysis; however, the required data for timing analysis is virtually non-existent. The documentation for the aJile chip, for instance, goes into great detail about the processor schematics, pinouts, signaling, and the runtime Java classes, but it lacks information on the worst-case cycle time of each bytecode instruction. Only JOP explicitly defines these crucial details.

Chapter 4

Annotating Control Flow for Interactive Analysis

Chapter Summary

Context Analyzing a program for worst-case execution time begins with knowledge of control flow—the order of execution of the program's instructions. This knowledge is typically captured as a graph data structure that corresponds to the connections between basic blocks, loops, and branches of the underlying code. Representing the code in this manner adds a higher level of abstraction that facilitates more complex analysis, not only for WCET, but also for compiler optimization, reverse engineering, static bug detection, and so on.

Prior Work Countless control flow graph construction algorithms and tools have been created, usually as part of larger projects such as compiler infrastructures. In the context of WCET analysis for Java microprocessors, the most notable examples include Soot, aiCall, and Avrora. They are able to generate a graph of control flow but cannot create a tree-based representation that some WCET algorithms require. Most do not even guarantee that the control flow data matches the original program, since the focus is typically on compiler optimization, where correctness is fundamental but precise timing is not.

Problems Because control flow graphs are a representation of the underlying machine code, any visualization of the data is difficult to comprehend because it contains only the low-level instructions and often lacks any obvious relationship to the original source code. Existing algorithms make no attempt to bridge this gap by mapping their control flow data structures to the source. In addition, a graph-based structure is not suitable for tree-based analysis algorithms that would make fast, interactive WCET analysis possible.

New Claims This chapter presents the first generic, stand-alone tool for control flow analysis of Java bytecode. Called Cascade, it preserves all bytecode so that it may be used safely for WCET analysis. It also demonstrates a novel approach of decompiling the program and mapping the results to the control flow data structures. Not only does this technique make the data easier to understand, it is also an important prerequisite for the back-annotation feature described in Section 5.3.1. Furthermore, Cascade can optionally generate a tree-based variant of the control flow data that is much faster to analyze and visualize.

Results The architecture of the Cascade tool is presented with a special emphasis on its support for the tree-based representation. Empirical evidence shows that this tree structure makes control flow creation and drawing much faster. Anecdotal evidence also suggests that trees produce superior visualizations of control flow because they more closely match the structure of the original source code. Finally, an example is shown of integrating Cascade into a traditional development environment.

The computer scientist Alan Perlis once said that writing an incorrect program is easier than understanding a correct one [123]. As real-time systems continue to grow in size and complexity, this axiom is taking on new significance. The ability of developers to understand their real-time code is eroding, especially when relying on older technologies such as C. Despite its status as the most popular language in real-time computing, it is relatively low-level, error-prone, and sometimes makes incorrect programs easy to write.

Chapter 3 outlined a solution to this complexity problem. The basic approach is to construct real-time systems with a cleaner, higher-level language that makes software easier to create, understand, and analyze. The analysis factor in particular helps meet the broader goal of interactive analysis, which depends heavily on WCET knowledge to prevent timing bugs.

As for the remainder of the Perlis epigram—understanding correct programs—the object-oriented features of high-level languages hide unnecessary details and keep interfaces separate from their implementations. These capabilities aid in understanding the functional correctness of a program. For *temporal* correctness, however, modern high-level languages fall short. The analysis model of Chapter 2 promises to fill this void through interactive tools that help developers understand the timing properties of their software—properties that would otherwise be nearly impossible to comprehend.

With the assumption of Java code running on Java processors, the foundation for constructing these tools is already in place. The next step is to move up to a higher level of abstraction. As shown in Figure 4.1, the layer above Java bytecode is control flow analysis, which produces a data structure expressing the order of execution of individual statements. This data structure is then used as a basis for WCET computation and other elements of interactive analysis.

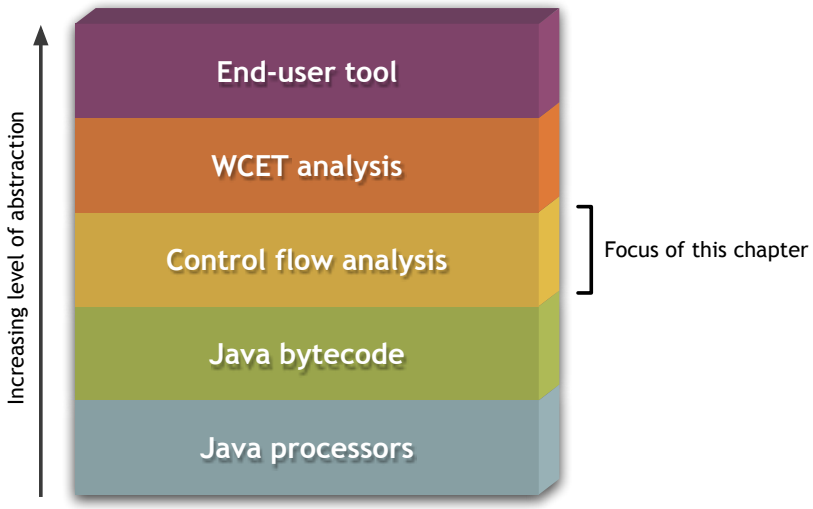

Figure 4.1: Tools for interactive analysis must operate across the multiple layers of abstraction shown in this diagram. At the core lies the simplifying assumption of a Java processor, followed by Java bytecode that runs on the processor. Control flow analysis—the focus of this chapter—is then built on top of the Java bytecode.

Control flow analysis necessarily begins with knowledge of the exact code that will run on the target device. The analysis must then examine this code to build up a model of the flow. The process is simplified somewhat in that all code relating to data flow can be ignored; only the control flow is relevant. For example, a series of instructions to calculate an arithmetic expression may push and pop the stack, read and write memory, and so on, but as long as the sequence invokes no methods and takes no branches, it can be reduced to a single entity known as a *basic block*. By definition, basic blocks are sequences of instructions without any transfers of control. If one instruction in a block is executed, all are.

To accomplish the goal of grouping code into these basic blocks and specifying the flow between them, two general tactics have been employed:

1. Write a custom tool that parses Java bytecode[1] and, by analyzing branch targets, groups the bytecode into its constituent basic blocks, loops, if statements,

[1] Although this chapter assumes Java bytecode, the general techniques apply to other bytecode-based languages such as C#.

and so on.

2. Modify a Java compiler so that the abstract syntax tree (AST) it produces can be saved to a file for later analysis. Additional hooks must be added so that the code generation phase of the compiler can link the bytecode to the AST. (This step is necessary for WCET analyzers and other higher-level tools.)

Both techniques have advantages and drawbacks that are mutually exclusive. For instance, a custom tool designed only for inspecting bytecode and extracting control flow information is easier, in general, than adapting a compiler's innards for WCET-oriented control flow analysis. Most compilers were never intended for such a task and tend toward monolithic designs that are not amenable to customization. On the other hand, a compiler approach can be more powerful. The data flow analysis built-in to most compilers may be combined with the control flow information for more sophisticated analysis. A loop bound detector, for example, could be derived from the loop unrolling mechanism available in optimizing compilers. As a result, instances of both techniques can be found in prior work for control flow analysis of Java bytecode.

4.1 Related Work

Control flow information has long been important for many types of program analysis. The availability of control flow is a key factor in the precision of pointer analysis algorithms, for instance [124]. These algorithms are able to detect certain cases of null pointer dereferencing and other invalid pointer operations without actually running the program. Compilers also rely on control flow analysis as the starting point for a large number of optimizations, while program transformation tools use it to

convert source code from one language to another. More recently, control flow analysis has been applied to the enforcement of security models in high-level languages, helping ensure that mobile phones, for example, are protected from malicious code downloaded from the Internet [125].

For Java, a variety of control flow analysis tools are available, though none is designed with WCET analysis in mind. The vast majority are hidden within some sort of compiler infrastructure to support AST generation and similar internal chores. For instance, the program transformation framework Spoon [126] includes a partial evaluation engine that calculates the control flow of a program for identification and removal of dead code.

Optimizing compilers for Java also depend heavily on control flow analysis. The bytecode optimization framework Soot [127] generates control flow graphs for subsequent data flow analysis and conversion to static single assignment (SSA) form. Soot provides several different control flow abstractions via its DirectedGraph interface, as well as an interactive tool (see Figure 4.2) that integrates control flow with data flow analysis for debugging purposes [128].

FLEX, another compiler infrastructure for Java, is designed for code generation as well as bytecode optimization [129]. It can transform Java source code into StrongARM instructions, MIPS instructions, or portable C. As part of this process, it converts Java code into an intermediate representation whose control flow graph can be queried programmatically via a public API.

A less common but equally valuable application of control flow analysis is test coverage. The eXVantage system is one such example; it includes a static analysis phase during which it parses Java class files and constructs a corresponding control flow graph. The system then examines the graph and, through dominator analysis [130],

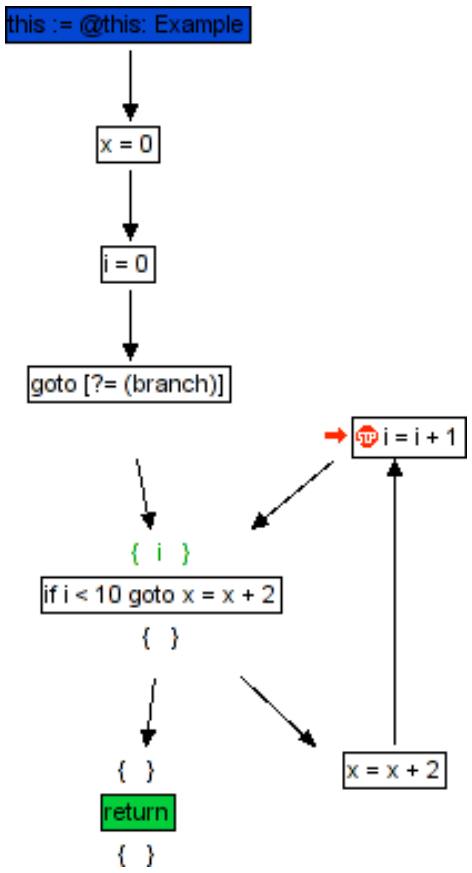

Figure 4.2: Soot, a framework for bytecode optimization, includes an interactive control flow graph tool for debugging intraprocedural analyses and for teaching students about control flow and data flow. (Screenshot by Jennifer Elizabeth Shaw [128].)

assigns an appropriate weight to each vertex and edge. The end result of this effort is a system that ensures all exceptions in a given program are handled properly.

Various other tools for static analysis in Java also incorporate control flow in some way. BAT_2XML performs control flow analysis in order to produce a high-level XML-based representation of Java source code [131]. SableCC, a compiler compiler, generates visitor classes for walking the nodes of an abstract syntax tree [132]. JRefactory [133], a semi-automatic source code refactoring tool, and Barat [134], a Java-based compiler front end, also produce ASTs for representing the control flow of an input program.

Thus far, no existing control flow analysis tool has been applied to the problem of

WCET analysis in Java. Instead, WCET analyzers generate control flow information on their own. The Javelin tool [64], for example, derives control flow graphs using a Java bytecode parser written from scratch in Ada. The WCA tool [135] likewise builds a model of control flow on its own, although it takes advantage of BCEL [136] to simplify parsing of bytecode. ASM [137], a bytecode manipulation framework like BCEL, can also be used for bytecode parsing and includes limited support for extracting control flow information from a single method.

4.2 Source-Annotated Control Flow Analysis

These prior efforts in control flow analysis suffer from two weaknesses. First, most tools are intended for compiler research and therefore focus on bytecode optimization and code transformation. Any control flow structure generated from the initial pass may not resemble the actual bytecode produced in the final pass due to optimizations such as loop unrolling. This factor is especially significant for WCET analysis, which requires a control flow analyzer with a precise mapping from bytecode instructions to control flow for accurate timing predictions. Modifying existing compilers for this purpose, which are already inherently complex, is a formidable task and may require extensive interleaving of custom code.

Second, control flow analyzers tend to be cryptic. They typically display only assembly code or an intermediate representation when visualizing the control flow structure. The aiCall control flow grapher [138], for example, shows assembly code in its output window (see Figure 4.3), requiring the user to understand the intricate details of a program even if it was written in a high-level language. Other control flow tools, such as the one provided with the simulation and analysis framework Avrora [139], are just as inscrutable (see Figure 4.4). Even tools designed for the needs of WCET analysis,

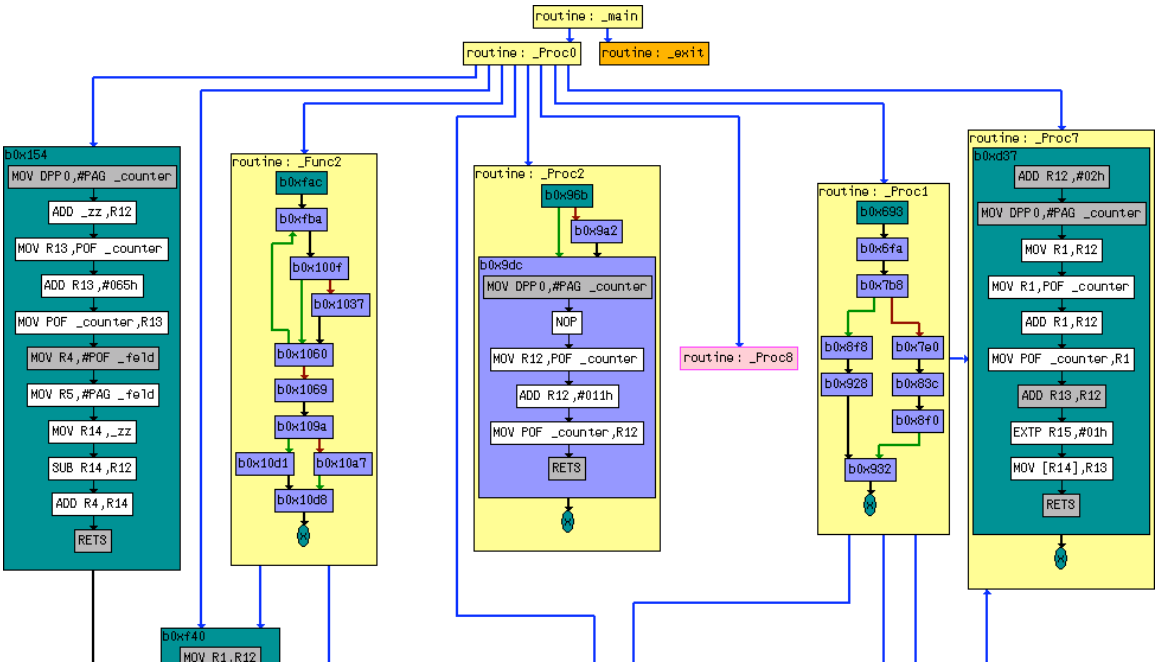

Figure 4.3: Many control flow graph visualizations, such as the aiCall tool shown here, require the user to develop in a high-level language but analyze in low-level assembly code. Understanding the behavior of the code is impossible without a deep understanding of the processor's instruction set. (Screenshot by AbsInt Angewandte Informatik GmbH.)

such as WCA [135], are equally obfuscated in their visualizations (see Figure 4.5). This exposure of low-level assembly code places an undue burden on the programmer and conflicts with the goals of interactive analysis.

Avoiding the first weakness is normally handled by writing a control flow analyzer from the ground up, but the second weakness is much more difficult to circumvent. Ideally, a control flow analysis tool would be able to reverse engineer executable instructions back into human-readable source code. The tool could then combine the control flow visualization with this source code for easier data digestion by the user. In practice, though, executable machine instructions bear little resemblance to the original source code due to compiler optimizations, user-defined data types, idiomatic expressions (such as bit-shifting instead of multiplication), and so on. Despite prior research in this direction [140, 141], decompilation of C programs is still limited and

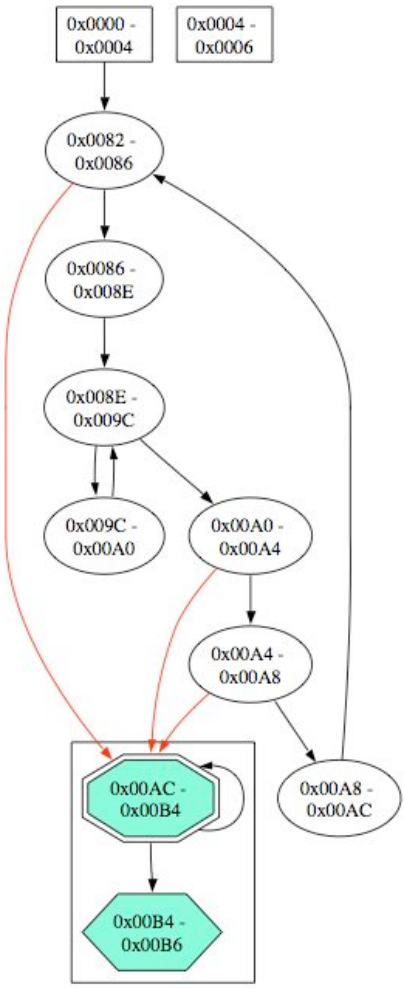

Figure 4.4: Avrora, a software simulation and analysis framework, includes a tool for control flow visualization. Because its visualizations lack high-level source code information, the semantics of the programs under analysis are obscured. (Screenshot by the UCLA Compilers Group.)

only semi-automatic.

In Java, however, these conditions no longer hold. Java class files contain more information than is available in compiled C executables, and the high-level nature of Java bytecode allows a near-perfect reconstruction of the original source code, assuming that the bytecode contains standard debugging symbols and has not been obfuscated. The susceptibility of Java to decompilation presents a unique opportunity for control flow analysis: Every control flow element (e.g., basic block) can be mapped to

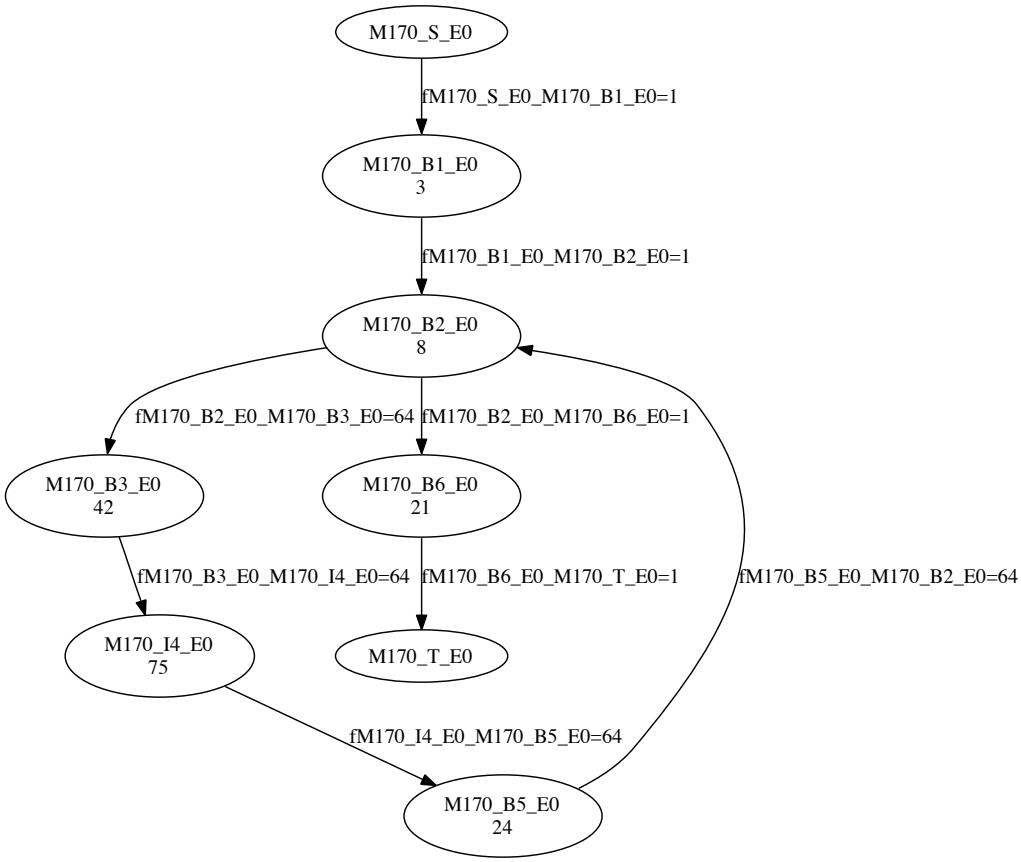

Figure 4.5: WCA, a prototype tool for WCET analysis on the JOP, can produce a graph of control flow. The graph shown here is a visualization of the program in Figure 4.6, but the relationship is nearly invisible because the graph retains none of the source code from the original program.

a representative expression in the Java language. By exploiting this feature, analysis tools can automatically *annotate*[2] control flow data structures with source code information, making them much easier to comprehend.

For example, a control flow graph of the computeVelocity method from Figure 4.6 would show only bytecode instructions if generated by conventional tools:

[2]An *annotation* in this context refers to a source code expression attached to the corresponding element of a control flow data structure. The term should not be confused with the metadata facility in Java—also known as an *annotation* facility—that allows source code elements to be marked as having a particular attribute.

```
class SpeedSensor
{
    private final static int VELOCITY_SIZE = 64;

    private int computeVelocity(int startVelocity,
                                int acceleration,
                                int deltaTime)
    {
        return startVelocity + acceleration * deltaTime;
    }

    public void getVelocityData(int[] v, int[] u, int[] a, int[] dt)
    {
        @LoopBound(max=VELOCITY_SIZE)
        for (int i = 0; i < VELOCITY_SIZE; i++)
        {
            v[i] = computeVelocity(u[i], a[i], dt[i]);
        }
    }
}
```

Figure 4.6: This example program, which computes a simple velocity function, forms the basis of Figures 4.8 and 5.3. The purpose of the code is unimportant; it exists only to illustrate control flow and WCET analysis techniques. (The @LoopBound statement is a custom annotation for communicating loop bounds to a WCET analyzer.)

```
iload_1
iload_2
iload_3
imul
iadd
```

With decompiler support, the tool can match this bytecode sequence to its high-level source code construct. It can also reproduce the original variable names by reading debugging symbols from the class file, resulting in the following expression:

```
startVelocity + acceleration * deltaTime
```

Another noteworthy benefit of decompilers is their ability to identify Boolean expressions hidden within the control flow. Consider, for instance, the following source code:

```
if (i > 10 && !b)
    i += 6;
```

From this snippet, a typical Java compiler might generate the following bytecode:

```
 0:   iload_0
 1:   bipush    10
 3:   if_icmple 13
 6:   iload_1
 7:   ifne      13
10:   iinc      0, 6
13:   ...
```

Traditional control flow analyzers would map the above branching instructions directly to nodes in the control flow, as shown in Figure 4.7. This one-to-one relationship results in a disconnect between the source code structure and the control flow structure: There is one branch in the source code but two branches in the control flow. Failing to capture the high-level structure of the original source code can lead to confusion for the user studying the control flow.

By comparison, a control flow analyzer with decompiler support collapses the branching bytecodes into a single entity (shown in Figure 4.7 as a gray box) by applying Ramshaw's algorithm [142] or a similar technique. In other words, the decompiler identifies the high-level Boolean expression implemented by the low-level bytecode, allowing the analyzer to group the vertices of the graph accordingly. This approach

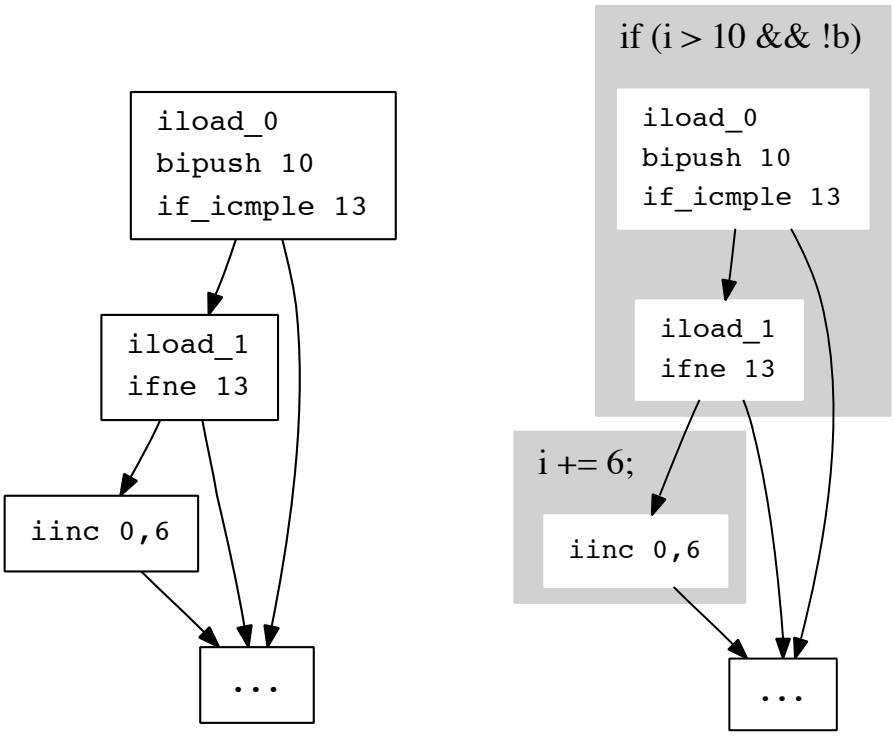

Figure 4.7: This simple example of the expression if (i > 10 && !b) i += 6 shows the difference between traditional and annotated control flow information. The unannotated graph on the left appears to contain two if statements, while the annotated graph on the right not only reveals high-level source code but also more accurately represents the original expression's structure.

produces a structure that bears a closer resemblance to the original program flow while also incorporating source code statements into each entity. (Variations on the visualization could collapse the edges so that they link the entities instead of the vertices, making the graph match the source code structure even more closely.)

While the difference may seem small in the simple example of Figure 4.7, it becomes significant as code complexity grows and control flow readability dwindles. Furthermore, the annotation and grouping remains fully automatic, no matter how large and complex the program becomes.

Such features clearly enhance the readability of control flow data, yet they would be difficult to implement properly in lower-level languages such as C. In Figure 4.8, for

example, a typical C-based control flow analyzer would show only the processor instructions of each basic block, but with decompilation support, the Java-based control flow analyzer was able to annotate the vertices of the graph with the Java statements they represent. Note that this particular visualization, which was produced by the Cascade tool described in Section 4.4, also groups the control flow entities according to the method in which they belong to further enhance readability. The process is entirely automatic and does not require manual intervention of any kind.

4.3 Strengths and Limitations of Decompilation

Understanding programs is indeed difficult, as Alan Perlis said, but annotated control flow analysis is a step toward mitigating this problem. The source code visualization enabled by decompilation helps guide developers through the process of program analysis. The decompilation aspect also offers an intriguing side-effect: It plays a major role in enabling *back-annotation*, a key component of interactive analysis that will be explored in detail in Chapter 5.

Decompilers for Java also serve as a practical foundation for writing WCET analysis tools because they perform essentially the same task—that is, parsing bytecode and building up control flow information—that traditional WCET tools have always done. Integrating a decompiler into the analysis process is, however, a novel and unconventional tactic. It has not previously been attempted, most likely because the very idea of Java as a hard real-time programming language is so new.

The general idea of decompilation is, on the other hand, not new at all. Decompilers emerged in the 1960s, just a few years after the invention of the first compilers [143]. They were intended to recreate lost source code, translate binary programs from one

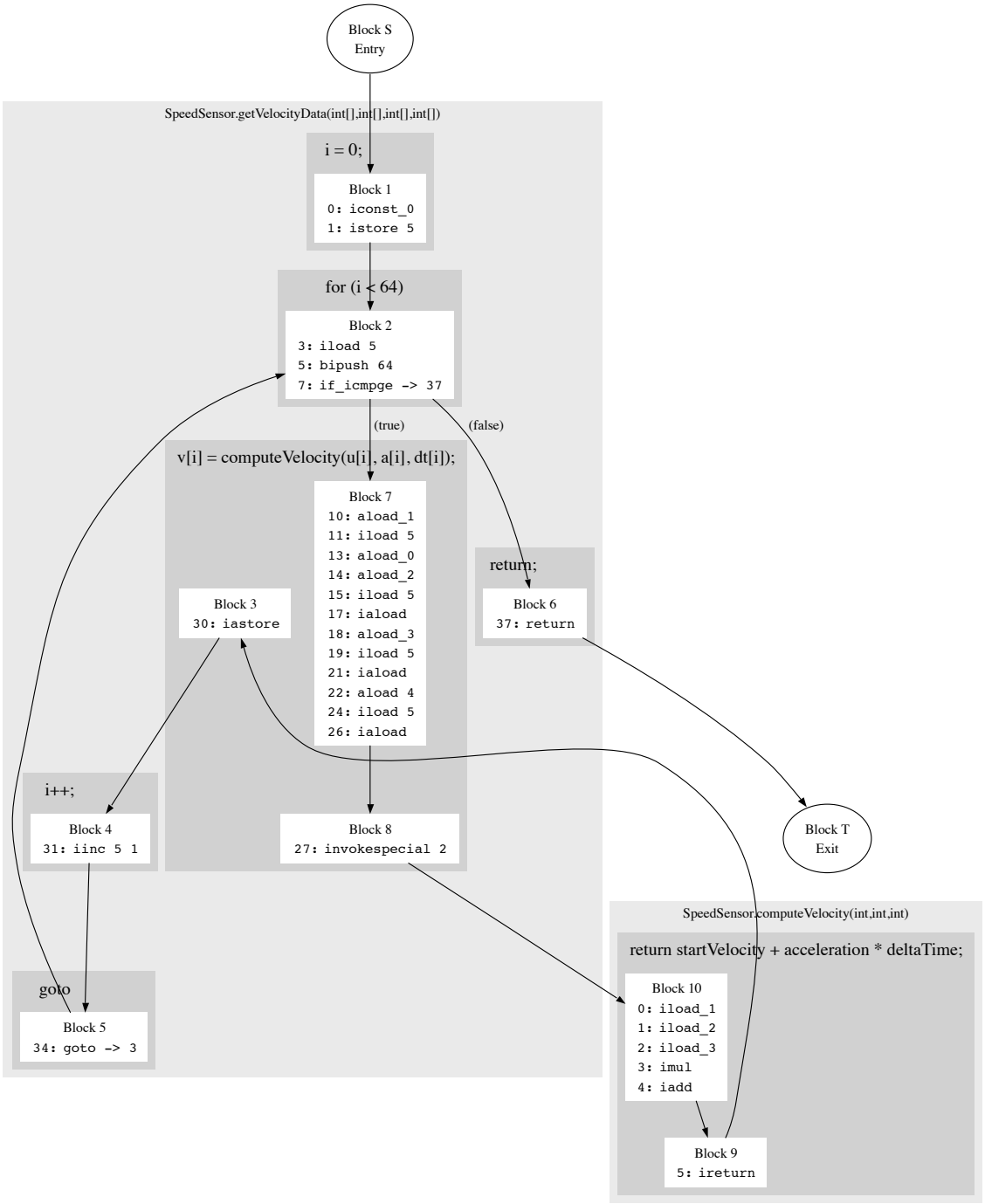

Figure 4.8: An annotated control flow graph, such as the one shown here of the code in Figure 4.6, contains more human-readable information than the examples in Figures 4.3, 4.4, and 4.5. The user does not even require the original source code listing to understand the semantics of the control flow.

machine to another, debug compiler code, and perform other such tasks. These first-generation decompilers were only about 90% accurate, leaving the programmer with the chore of decompiling the remaining 10% by hand. Even the decompilers of today have not improved upon this ratio, due in large part to the optimization tactics of modern compilers. As a result, reproducing the original C source code of an arbitrary binary program is virtually impossible. Decompilers tend to output scattered chunks of raw assembly code in places where they cannot determine the equivalent construct in C.

In Java, however, compilers typically perform no optimizations whatsoever (with a few notable exceptions, such as constant folding). The compiler authors usually rely instead on the performance-boosting power of Java virtual machines, whose just-in-time compilers translate bytecode into fast-running native code at runtime. The relatively simple compilation of Java source code gives decompilation a much greater chance of success. Indeed, only one year passed between Java's introduction in 1995 and the release of Mocha [144], the first Java decompiler, which caused a minor panic to erupt in the Java community. Suddenly, binary code was no longer hidden from prying eyes and could instead be reverse engineered into a nearly perfect recreation of the original source code, revealing programming tricks and other trade secrets that would have been well-hidden within a C binary. The uproar prompted the author to momentarily withdraw Mocha from public distribution.

Since then, the angst surrounding Java decompilation has largely subsided. Most developers seem to have realized that Java decompilers have legitimate purposes, just like the original decompilers of the 1960s, and are not inherently wrong. Eric Smith, the curator of Mocha, notes that "attempting to ban tools like Mocha to prevent reverse engineering of software is like trying to ban socket sets to prevent reverse engineering of automobiles." [144]

```
static void FillPowerMatrix(Digit matrix[][], Digit x[]) {
    int n = matrix[0].length;

    for (int i = 0; i < n; i++) {
        matrix[i][0] = new Digit(1);

        for (int j = 1; j < n; j++) {
            matrix[i][j] = matrix[i][j-1].mult(x[i]);
        }
    }
}
```

Figure 4.9: This listing is a simple matrix multiplication example to illustrate source code before obfuscation. The result of obfuscating this code can be seen in Figure 4.10.

Regardless of how developers may feel about the ethics of reverse engineering, protests against Java decompilers have abated mostly because of the simple realities of the Java industry. The vast majority of Java code is now deployed not as end-user desktop applications but as web services, where bytecode is not available for inspection. Even if it were, the best decompilers on Earth are unable to recover source code comments and other documentation that may be necessary to make use of the code. Furthermore, a variety of obfuscation techniques [145, 146] are now available in tools such as JHide [147]. These techniques, which include identifier scrambling, control reordering, and method merging, change the bytecode in ways that thwart decompilation without breaking the semantic correctness of the program. Figures 4.9 and 4.10 provide a before-and-after example of the effects of obfuscation.

These obfuscation techniques would seem to foil the idea of annotated control flow analysis if not for one simple fact: The developer analyzing the code is very likely the same person who wrote the code. There are few scenarios in which analysis would take place without the developer having access to the original source code. Even if the analysis and the coding are performed by two different individuals, they would almost certainly be part of the same team of developers. Tools for annotated control flow

```
static void FillPowerMatrix(Digit[][] r0, Digit[] r1) {
    long l0;
    int i2, i3;

    for (i2 = r0[0].length, i3 = 0; i3 < i2; i3++) {
        r0[i3][0] = new Digit(1);

        for (l0 = (long) 1 & 4294967295L ^ l0 & -4294967296L;
        (int) (l0 & 4294967295L) < i2;
        l0 = (long) (1 + (int) (l0 & 4294967295L)) ^ l0 & -4294967296L) {
            r0[i3][(int) (l0 & 4294967295L)] =
            r0[i3][(int) (l0 & 4294967295L) - 1].mult(r1[i3]);
        }
    }
}
```

Figure 4.10: This listing shows the effect of an obfuscation technique that packs local variables into bitfields [145]. The code performs the same computation as Figure 4.9 but is much more difficult to read and understand.

analysis can therefore assume that obfuscation is disabled, since there is no reason to obfuscate code from oneself.

The next step, then, is to select a decompiler best suited for the needs of annotated control flow analysis. On the surface, the choice may seem arbitrary because decompilers for Java operate on the same basic principles [148]. Starting with Java bytecode as input, they identify expressions and type information to build up a control flow graph. A sequencer then compresses the graph by converting particular branching patterns into Boolean expressions using Ramshaw's algorithm [142]. The result is a legal, though possibly convoluted, abstract syntax tree. Next, the decompiler transforms this AST to resemble a more natural program by converting anomalous break and continue statements into equivalent loops and if statements. Finally, it translates the normalized AST into Java source code. Figure 4.11 illustrates this process.

While all decompilers tend to follow the same general strategy, they are not all alike. Some, like Mocha, are relatively basic and are only able to invert known compilation

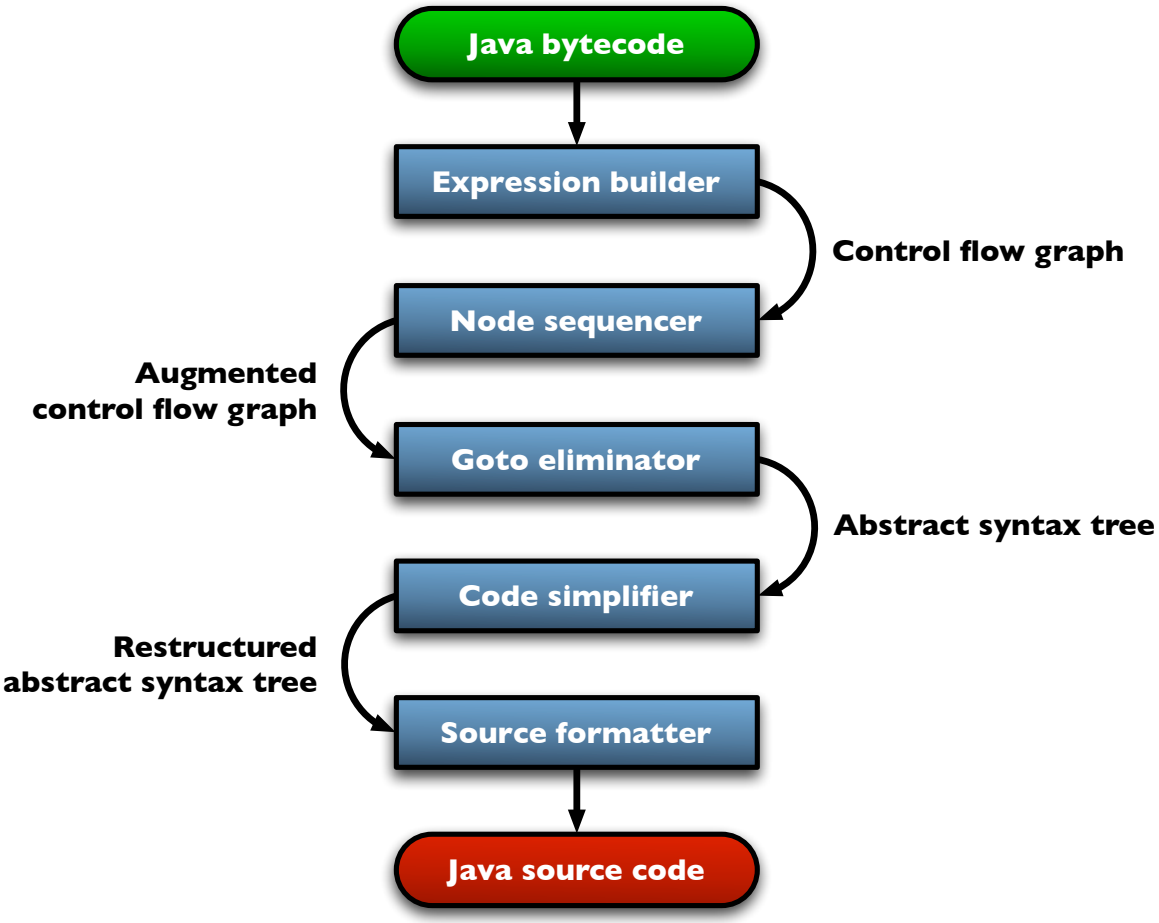

Figure 4.11: Decompilers for Java tend to follow the same general process: They build up a control flow graph, simplify and normalize the graph, generate an abstract syntax tree from the graph, and then translate the AST into nicely formatted Java source code.

strategies, limiting their usefulness to a specific compiler. Others, like Dava [149, 150], are more adept at decompiling arbitrary code patterns and can cope with bytecode optimizers, non-Java compilers (e.g., Fortran-to-bytecode compilers), and even certain types of obfuscations. For example, weaker decompilers presented with the following bytecode (shown in simplified form):

```
compute cond
ifeq label1
stmt1
```

```
        return
label1: stmt2
        return
```

would transform it into the following source code (shown in pseudocode):

```
void f() {
  if (cond)
    stmt1;
    return;
  else
    stmt2;
    return;
}
```

instead of this more accurate version:

```
void f() {
  if (cond)
    stmt1;
  else
    stmt2;
}
```

In addition to avoiding such weaknesses, a Java decompiler must also consist of a callable library in order to be suitable as the foundation of an annotated control flow analysis tool. Many of the currently available decompilers[3] exist only as a stand-alone executable or as a graphical front-end to an executable. Even those that are avail-

[3]For a comprehensive list of Java decompilers, refer to the work by Hou et al. [146] in testing decompiler resistance to obfuscation.

able in library form may be commercial products that cannot be modified, a factor that becomes an obstacle especially at higher levels of the tool stack. For example, a WCET analyzer requires that the control flow contains the exact bytecode of the original Java classes—not one instruction can be lost or altered—and most decompilers must be modified to support such bytecode preservation. The ideal decompiler for annotated control flow analysis is therefore an open-source decompiler library, of which there are only three:

- Dava [149, 150]
- Java Optimize and Decompile Environment (JODE) [151]
- JReversePro [152]

Currently, Dava is still an experimental prototype, while JODE and JReversePro are more mature projects that continue to be maintained. The latter two would both fit the needs of annotated control flow analysis, but a closer investigation indicates that the control flow information exposed by JODE is arguably more versatile and complete, providing the best mix of sophisticated decompilation techniques and extensible programmatic interfaces.

4.4 Cascade: A Control Flow Analysis Tool

With a strong decompiler as the foundation, a prototype tool can be constructed to demonstrate the practicality of the ideas presented in Section 4.2. The Volta project, introduced in Section 2.4, includes such a prototype as part of its tool suite. Called Cascade, it is the first tool to support the concept of annotated control flow analysis.

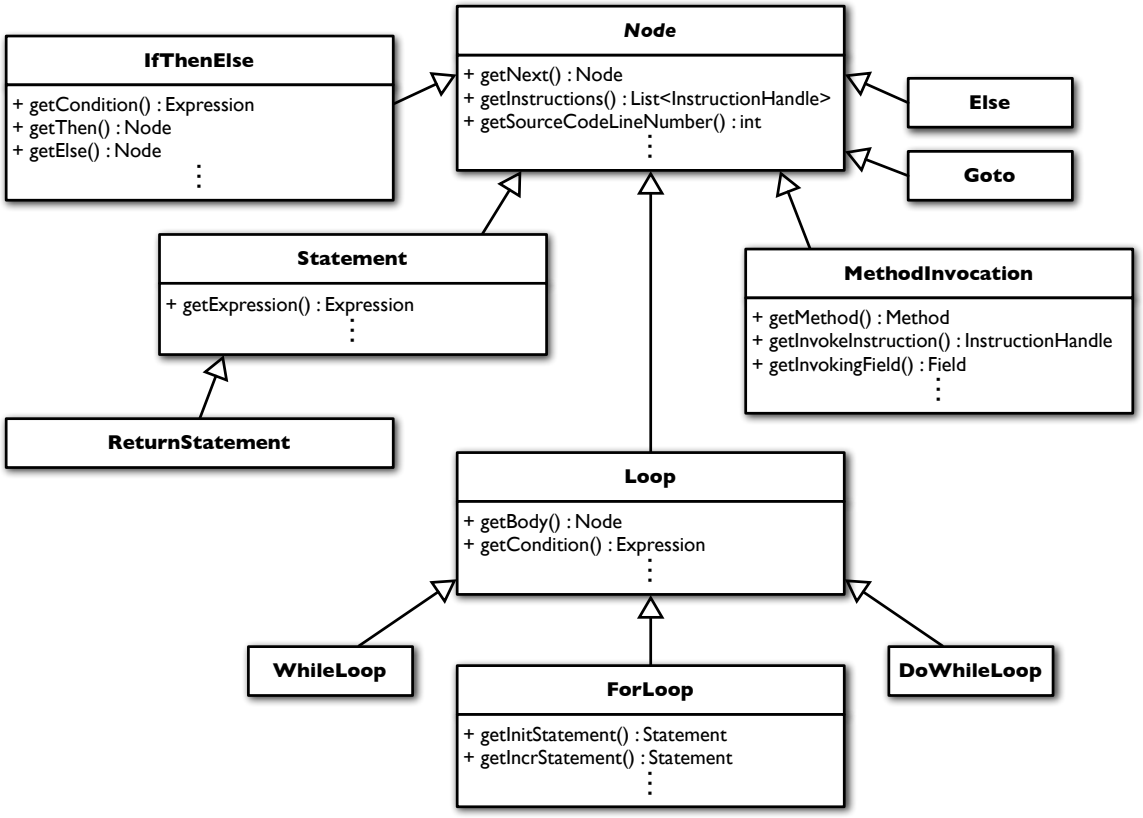

Figure 4.12: Cascade, a control flow analyzer with annotation support, translates Java programs into control flow data structures represented by the classes shown in this UML diagram. Each class provides operations for obtaining information about the control flow element, such as the bytecode sequence it represents.

Cascade is implemented in Java and is built directly on top of JODE. It translates JODE's decompiler-specific control flow data structures into a general-purpose class hierarchy, as shown in Figure 4.12. Other tools, such as WCET analyzers, can query these classes directly from Cascade, shielding them from the complexities of the decompiler library. Cascade also provides important reflection services for loading classes, obtaining method handles, computing the static code size of a method, and so on.

Another key benefit of Cascade is its modularity. Most control flow analysis tools are monolithic; they are woven tightly into the programming fabric of some other tool, such as a WCET analyzer or a compiler. Cascade is different in that it can

act as a general-purpose control flow analyzer. It is therefore useful not only for the interactive analysis tools of the Volta stack but for any application requiring access to the control flow structure of a Java program. For example, Cascade can act as an enhanced disassembler utility, much like dis [153], jasmin [154], or javap [155], but with a graphical depiction of the disassembly rather than the canonical text dump.

To achieve this modularity, Cascade is designed purely as a control flow analyzer. The specifics of WCET analysis are kept out of the tool entirely. This self-contained design allows other researchers to build on top of the implementation. For example, one could write a new WCET analyzer, or some other tool entirely, without any conflicts between existing tools in the Volta project that also rely on control flow analysis.

For the benefit of such tools, Cascade exposes an API for operations such as:

- Iterating through control flow structures
- Identifying loops
- Finding method invocations
- Obtaining the bytecode, as well as the human-friendly decompiled source code, associated with any given node
- Translating bytecode into BCEL format [136], a *de facto* standard for disassembly of class files
- Exporting control flow data to Scalable Vector Graphics (SVG), Graphviz (DOT), Graph Modeling Language (GML), Graph Markup Language (GraphML), and plain text file formats

The latter two features are particularly valuable because they allow Cascade's output to be fed into other tools for further processing and visualization. For example, BCEL compatibility enables interoperability with existing Java tools that are also centered around the BCEL format. Likewise, the DOT export format allows control flow graphs to be rendered using the layout algorithms of Graphviz [156], as shown in Figure 4.8.

Note that this figure is not a mock-up; it is actual output generated by Cascade of the program in Figure 4.6. It demonstrates the improvements in readability that annotated control flow offers compared to the current standard of control flow visualization depicted in Figures 4.3, 4.4, and 4.5. The figure also illustrates how Cascade automatically annotates each vertex in the graph with the corresponding source code provided by the decompiler. Without this extra information, the graph would be much more difficult to comprehend, and the relationship between the control flow and the original program would be far from obvious. Cascade is the first control flow analyzer to provide such a feature.

4.4.1 Control Flow Graphs vs. Control Flow Trees

The idea of annotated control flow analysis seems to imply that the only interesting problem is how to combine source annotations with control flow analysis. The development of Cascade revealed many additional challenges, however. Simply constructing the visualization of a control flow graph in a scalable manner is non-trivial. Not only do performance problems emerge when laying out a large graph structure on a two-dimensional surface, there is also the issue of human comprehension of such a complex set of data. The main problem is that graphs are a rather unintuitive means of visualizing program flow. Indeed, visualization of control flow data as a

graph is largely an artifact of the algorithms used to analyze that data, rather than a deliberate design choice with the user in mind.

The central problem of using graphs to visualize control flow is that source code is not structured as a network of vertices but rather as a tree. A well-formed Java program, for example, is expressed as an abstract syntax *tree*, not an abstract syntax *graph*. A graph simply does not correspond to the tree-like structure of high-level source code. The circuitous graph of Figure 4.8, for instance, does not match the hierarchical structure of Figure 4.6.

This problem is exacerbated by the fact that conventional graph layout algorithms [157] have no knowledge of the underlying code, often resulting in eccentric graphs without any patterns that are recognizable in the original source code structure. The flow will twist and turn in unpredictable directions that depend on the algorithm used to lay out the graph. In Figure 4.8, for instance, which was laid out with Graphviz's hierarchical algorithm, the control flow moves right, then left, then down, and finally back up again, making the flow of execution difficult to follow. Even when annotated with source code, graphs representing computer programs are less than intuitive.

Yet another drawback is that graphs are susceptible to the butterfly effect.[4] One small change to the control flow, such as adding a single line of code, can disrupt the layout and may drastically alter the appearance of the graph. For example, adding one statement after the for loop of Figure 4.6 causes Block 7 in Figure 4.8 to jump from the right-hand side of the graph to the left. Such arbitrary changes disorient the user and slow down the analysis process because of the extra time needed for the

[4]The "butterfly effect" is the notion that a stimulus as small as the flapping of a butterfly's wings might propagate changes in the atmosphere that may ultimately alter the path of a tornado. This whimsical scenario has formal roots in chaos theory, which states that small variations in the initial condition of a system may produce large variations in its long-term behavior. The phrase was coined by Edward Lorenz, who discovered that entering 0.506 instead of 0.506127 as the input to a weather simulator produced drastically different results [158].

human eye to adjust to the new layout.

Graphs are also problematic when trying to determine certain structural properties of the control flow. In particular, higher-level tools invariably need to query the control flow analyzer about its underlying structure, but a graph can make answering these queries very expensive. Solving a WCET problem, for instance, may require knowing whether an incoming edge of a control flow vertex lies within the body of a for or while loop. However, the query "Is this vertex part of a cycle?" necessitates a depth-first search with a time complexity of $O(vertices + edges)$. Running this search on each loop of the control flow adds considerable delay to the WCET computation, especially for larger programs.

To solve these problems inherent in a graph structure, Cascade offers flexibility in how it represents control flow data. It exposes through its API both a traditional control flow graph as well as a *control flow tree*. Representing control flow as a tree instead of as a graph offers notable benefits:

- A control flow tree is structurally identical to the original source code's syntax tree. Therefore, even a trivial tree layout algorithm—one that merely adds indentation on each descent—automatically follows conventional source code formatting rules. There is no need for a complicated and time-consuming graph layout algorithm. Instead, if and else blocks, as well as for and while loop bodies, are simply indented, as shown in Figure 4.13. This type of visualization has a clear and natural correspondence to the original source code that makes it easier to read. Compared to Figure 4.8, for example, it does not require the eye to follow the flow of control in arbitrary directions. The flow advances only down and to the right, just like source code.

- Visualizations of control flow trees are immune to the butterfly effect. Adding

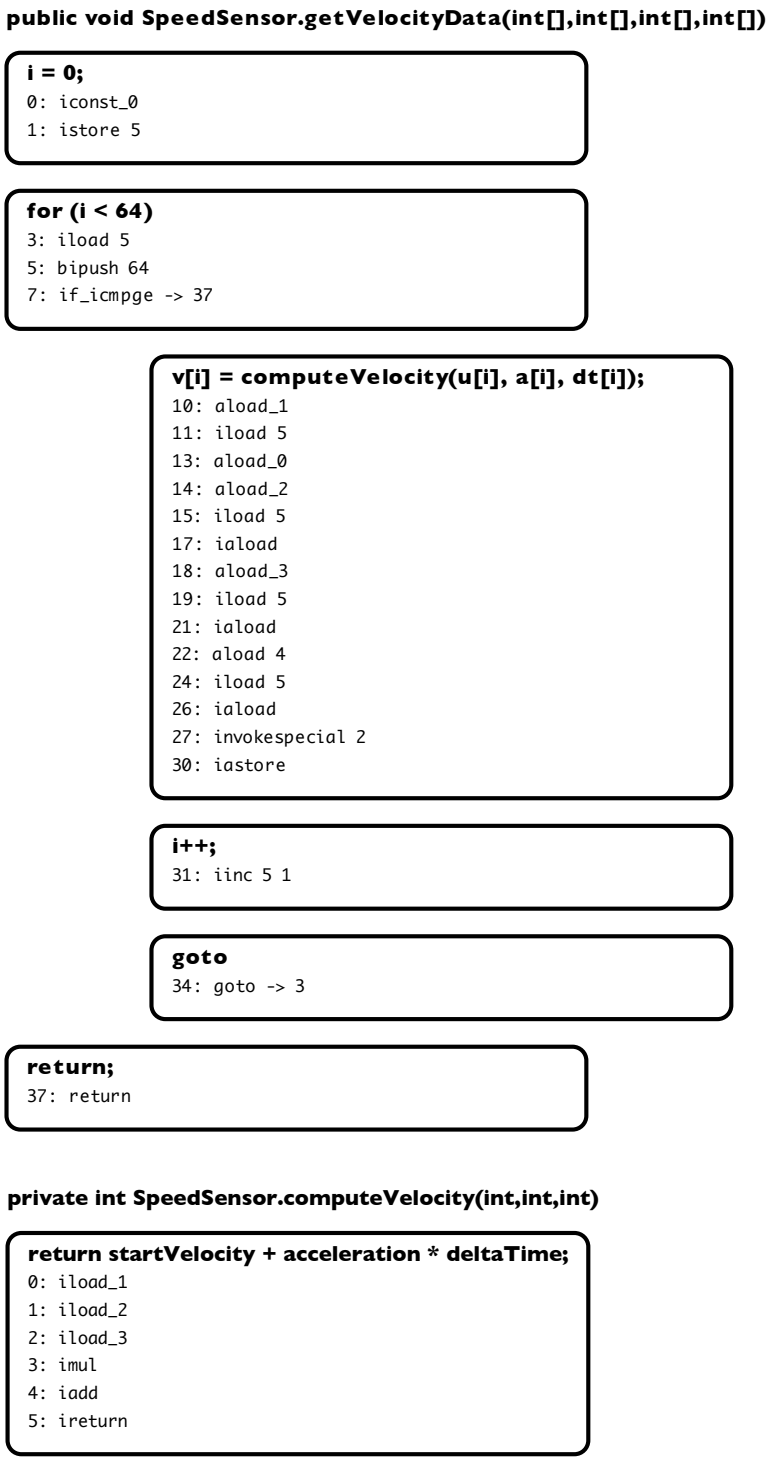

Figure 4.13: Control flow trees capture the same information as traditional control flow graphs, but they match the original source code structure, making comprehension easier. In this figure, for example, which was produced by Cascade from the code in Figure 4.6, the for loop is indented just as it would be in formatted source code. (The goto and return nodes appear in the tree as placeholders for their respective bytecode instructions, whereas in the source code they are implicit.)

a new statement simply shifts all subsequent statements to allow space for the new one. The overall tree structure does not change and remains stable in response to small changes in control flow. (Conceivably, a specialized graph layout could be created to produce a tree-like visualization that is also immune to the butterfly effect, but such an algorithm would likely be more complex and considerably slower than an equivalent tree layout algorithm.)

- Deriving information about the structure of a tree is generally faster and easier than for a graph. The time complexity of graph-based search algorithms tends to be a function of the number of vertices plus the number of edges in the graph, while the complexity of tree-based algorithms tends to be a function of the height of the tree. For example, determining whether a node in the control flow tree is part of a loop is simpler and more efficient than making the same determination for a vertex in a control flow graph. The latter requires a depth-first search, while the former only requires walking up the tree and visiting each parent until a loop node (or the root of the tree) is found. Similarly, tree structures are often easier to traverse. Consider, for instance, the for loop of Figure 4.8. A graph-based algorithm would have difficulty traversing from Block 2 to the inside of the for loop: Is it Block 6 or Block 7? Both are attached to Block 2's outgoing edges and thus they cannot be distinguished based on local information. A tree-based algorithm, on the other hand, could instantly proceed to the loop node simply by traversing to the node's child—a constant-time operation.

In addition to these advantages, control flow trees retain the same information found in a control flow graph. The underlying bytecode and an equivalent flow of control are still captured by the tree structure. It loses nothing; it is merely a different representation of the flow information. For this reason, the tree is the fundamental

abstraction of control flow in Cascade. When given a Java class file as input, Cascade constructs a control flow tree, rather than a control flow graph, and it exposes this tree structure through its API.

Some tools, however, may still require a graph representation of control flow. Many established WCET algorithms expect a control flow graph, for example. For the benefit of such tools, Cascade exposes a graph-based façade of the tree through its API. Tools built on top of Cascade can simply choose the desired representation by invoking the appropriate methods in Cascade's API. With this flexibility, Cascade enables a unique new breed of hybrid high-level tools that can use graphs for analysis but trees for visualization.

4.4.2 Performance of Cascade

Given that raw speed is a fundamental requirement of interactive analysis (see Section 2.3), the performance of Cascade deserves examination. The speed at which the tool constructs data structures and renders visualizations should be fast enough to analyze a program as it is written. Otherwise, Cascade could become a bottleneck not only for control flow analysis tasks but also for any tool that relies on its services.

Prior work has suggested that construction of control flow graphs is such a time-consuming process that the analysis results should be cached [159]. While caching is by no means a futile endeavor, performance measurements of Cascade show that it is often unnecessary. Construction of control flow data structures for programs of very high cyclomatic complexity [160] takes just a few hundred milliseconds even on today's low-end workstations. Longer delays are more likely to exist in higher-level tools, such as WCET analyzers, than in the control flow construction process.

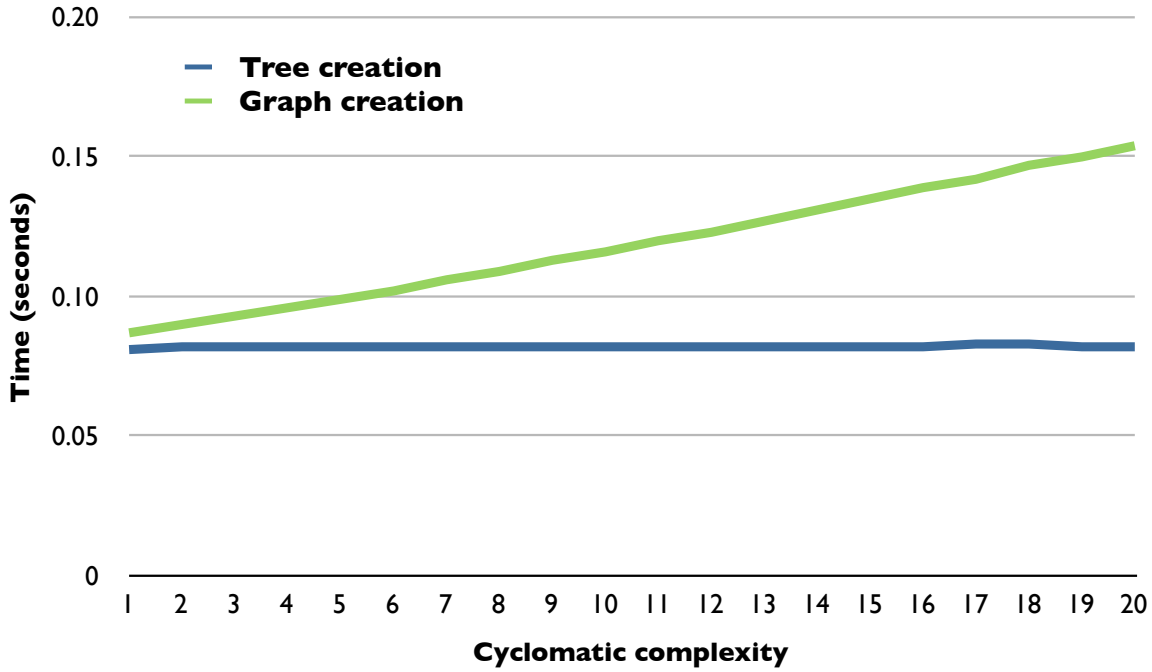

Figure 4.14: This graph compares the speed of creating a control flow tree versus a control flow graph using the Cascade tool.

Figure 4.14 illustrates these performance measurements in more detail. The results were obtained by measuring the time Cascade requires to create a control flow tree and a control flow graph. All tests were conducted on an IBM-compatible desktop workstation equipped with dual 700 MHz Intel Pentium III processors with one gigabyte of RAM running Java 1.6.0 on top of the Linux 2.6.17 kernel. The cyclomatic complexity of the fabricated input program was gradually increased to determine how performance varies as the size and complexity of the input increases.

As shown in the graph, both the tree and the graph control flow data structures are quite fast, requiring a fraction of a second even with hundreds of lines of code at very high complexity. (The twentieth benchmark has a cyclomatic complexity of twenty and contains approximately 300 lines of non-commenting source statements.) In terms of scalability, however, the control flow tree variant is the clear winner. The computation time for a graph grows linearly, while the time for a tree is nearly

constant. This result is yet another argument in favor of using control flow trees in interactive analysis.

The speed of visualization is also a pivotal metric. Interactive analysis naturally requires interactive visualization, and therefore fast rendering of control flow data structures is a criterion that should not be overlooked. Here again, control flow trees have the advantage over control flow graphs. The time complexity of drawing a tree structure is linear, whereas graph drawing algorithms are roughly exponential in complexity.

Figure 4.15 provides evidence of this relationship. The input data and test environment are identical to that of Figure 4.14, but rather than measure the time to create a control flow data structure, the metric in this case is the time taken to convert an existing structure to SVG format. Rendering to SVG provides a common reference point for drawing speed because it only exercises the ability of a rendering algorithm to lay out a data structure onto a two-dimensional surface. Font selection, rasterization, and other tasks associated with drawing to a computer screen are factored out entirely, keeping the benchmarking process simple and direct.

These results show that as the size and complexity of a program increases, control flow graphs quickly become unsuitable for interactive visualization. Trees, in contrast, are a more appropriate choice when control flow data must be redrawn on every change to the program under analysis. Even at the highest complexity level in these benchmarks, control flow trees can be rendered in less than two seconds. The speed is even more impressive considering that a C-based program running as native code—Graphviz 2.8—performed the graph rendering, while interpreted Java code—Cascade's SVGTreeWriter class—performed the tree rendering and still managed to outpace the graph algorithm in every test case.

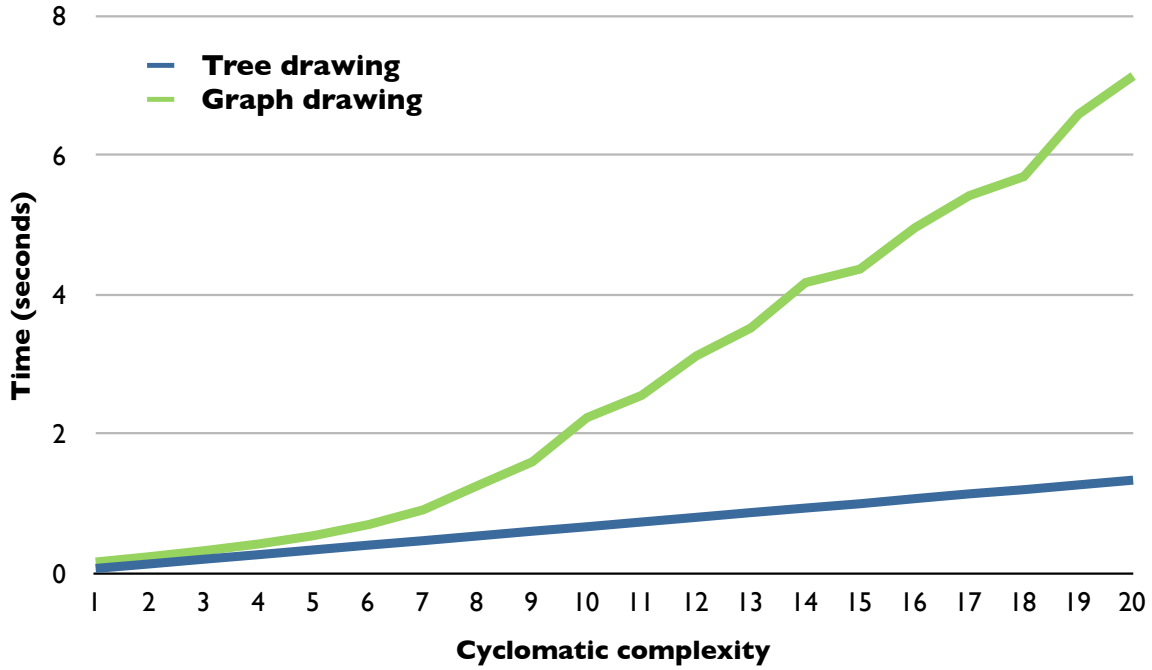

Figure 4.15: This graph compares the speed of drawing a control flow tree versus a control flow graph using the Cascade tool.

The superior speed of control flow trees is a catalyst in allowing direct integration of Cascade into a development environment. As shown in Figure 4.16, the result of control flow analysis can be displayed alongside the original source code and updated automatically as the code changes. The host environment in the figure is the jEdit [161] text editor, but the idea can be ported to other programming tools, such as Eclipse [162] or NetBeans [163]. This kind of parallel, interactive analysis at the control flow level sets the stage for more advanced high-level tools, as discussed in Chapter 5.

4.4.3 Limitations of Cascade

Cascade is notable for its unique characteristics, including annotated control flow analysis and support for control flow trees. It also offers several less novel but nonetheless practical features. For example, it retains bytecode instructions throughout the

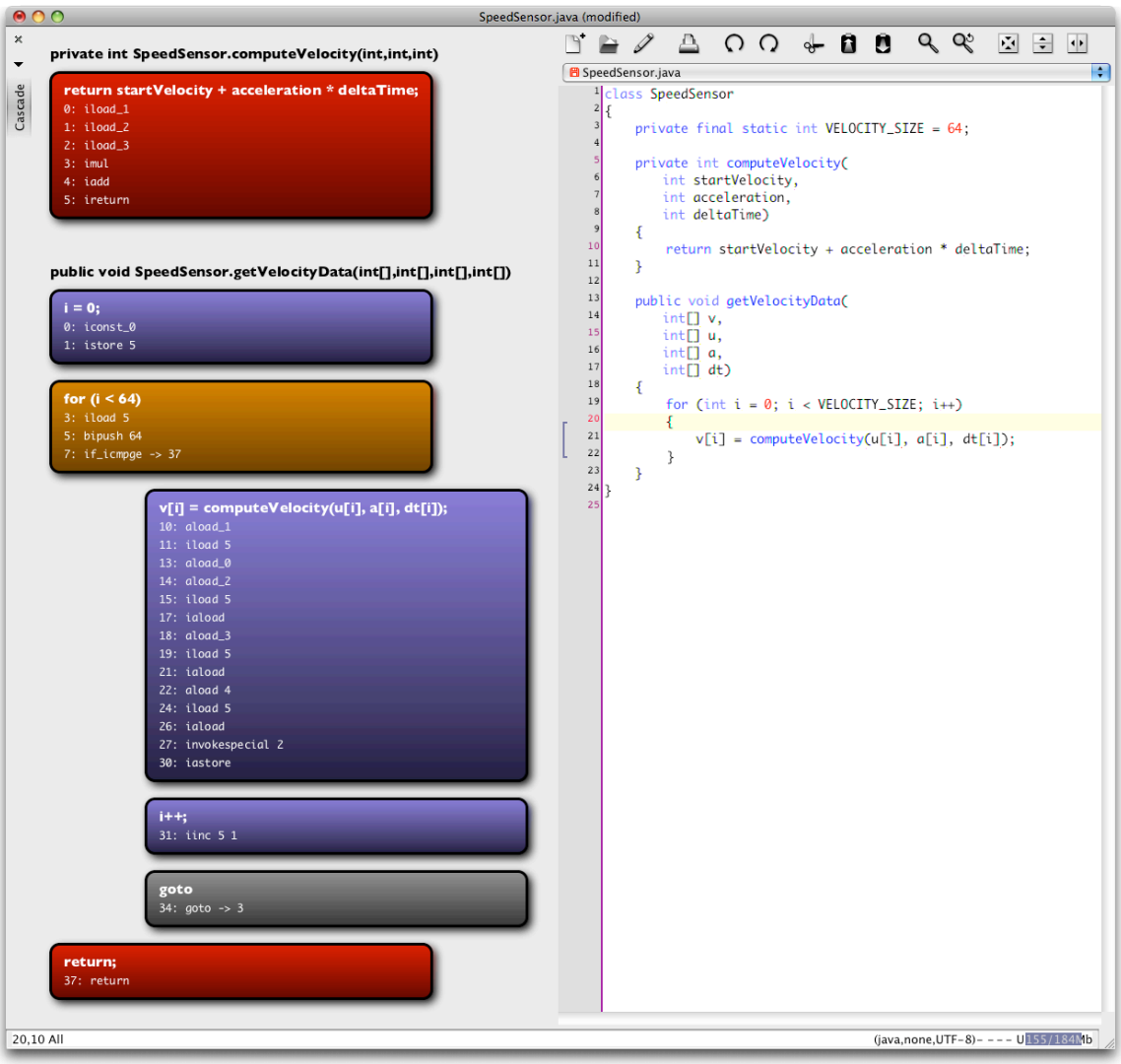

Figure 4.16: This screenshot of jEdit, a text editor for programmers, shows how the speed of control flow trees enables interactive analysis. On the left-hand side is a Cascade plugin for jEdit that visualizes a control flow tree of the current buffer and automatically updates the display as the code evolves. (The source code for this plugin is available in the Volta distribution.)

analysis process. While most compiler-based control flow tools simply discard them, Cascade propagates them from the original class file all the way up to its high-level data structures. It is also able to map these bytecodes to basic blocks or to individual source code statements for finer granularity. This property is particularly useful for WCET analysis tools.

In other areas, however, Cascade is currently lacking. As a proof-of-concept prototype, it is hampered by certain simplifying assumptions. Cascade does not allow labeled break or continue statements, for instance. This omission is not very serious because, although some legitimate programs may contain such statements, they are unnecessary and usually undesirable. Like the much-maligned goto statement, they can make programs arbitrarily hard to understand and optimize [164], and they violate the single-entry/single-exit paradigm of structured programming [165]. They are also quite rare in practice. For example, the java and javax packages in the Java Standard Edition 1.4 contain 220,000 non-commenting source statements, but only 52 of these (0.02%) are a labeled break or continue.

A more significant limitation of Cascade is its ignorance of synchronized statements and exception handling. The synchronized keyword is quite common when Java code is designed for a multithreaded environment, and the try/catch combination is pervasive in almost any Java program. Unfortunately, such statements complicate control flow reconstruction [149], and they are difficult to analyze for worst-case execution time. Some researchers in the real-time field are even abandoning synchronized statements altogether in favor of time-triggered techniques [43]. Given Cascade's primary role as a foundation for WCET analysis, the loss of support for synchronization and exception handling is objectionable but not critical.

Cascade also makes no attempt to solve the polymorphism problem. Object-oriented languages depend heavily on dynamic dispatch of methods to implement polymorphism, but Cascade is a purely static control flow analyzer—it performs no data flow analysis—and has no way to predict which object in the class hierarchy will be invoked by a particular polymorphic call. Of course, an exact prediction is impossible in the general case, but prior work has provided static techniques for determining the run-time types of variables, often narrowing the invocation target to a small number

of candidates [166]. Other approaches to the problem, including program slicing [167] and devirtualization [168], show promise but are still a subject of ongoing research. For this reason, the current version of Cascade neglects the issue of polymorphism, though a future one must address it.

Chapter 5

Interactive Worst-Case Execution Time Analysis

Chapter Summary

Context WCET analysis is a fundamental technique for guaranteeing the proper timeliness of a program. It begins with a control flow analysis to break down the program into its constituent parts: loops, branches, and basic blocks. A low-level analysis is then performed to compute the WCET of each basic block in isolation. Finally, a longest-path search determines the worst-case path through these blocks.

Prior Work When targeting Java microprocessors, control flow analysis and low-level analysis are relatively straightforward. Most work therefore centers on the longest-path search, for which there are two leading approaches: tree-based algorithms and the implicit path enumeration technique (IPET). The latter is more popular, having been implemented in tools such as Cinderella, Chronos, and Bound-T.

Problems The advantage of IPET is that it can account for data dependencies such as the false path problem simply by adding an appropriate constraint equation to the problem formulation. (Achieving the same result for tree-based techniques requires changing the algorithm itself.) Finding these constraints, however, is far from trivial and may require an extensive data flow analysis. In addition, IPET is NP-hard in complexity and therefore extremely slow. For small programs, it can calculate the WCET in seconds, but medium programs take minutes, and very large programs may take days. (By comparison, tree algorithms have linear running times.) When WCET analysis is this slow, it must be postponed until a final validation phase, but fixing timing errors after the code has been written is expensive, time-consuming, and may necessitate a redesign of the system.

New Claims This chapter claims that adoption of WCET analysis among industry practitioners is hindered by the predominance of IPET. If a faster WCET algorithm could be used, reducing analysis time to seconds instead of minutes, it could be integrated into every step of the development process and allow early detection and removal of timing errors. Focusing on the speed of analysis, not just its accuracy, will lead to increased productivity and perhaps higher quality code when building hard real-time systems. The end goal, therefore, is to make WCET analysis fast enough to be *interactive*, providing the developer with continuous feedback from the moment the first line of code is written.

Results These claims of improved productivity and quality as a result of faster WCET analysis are not proven in this work. Instead, the advantages are simply assumed to exist, and the focus then becomes *how* to achieve near-instant analysis results. New techniques for increasing the speed of analysis, and then integrating those results into a development environment, are provided in Chapter 6.

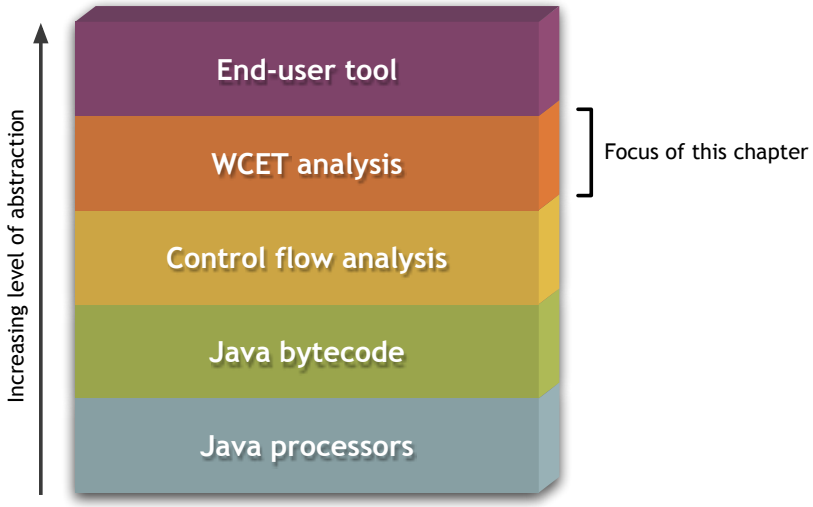

Figure 5.1: Tools for interactive analysis must operate across the multiple layers of abstraction shown in this diagram. At the core lies the simplifying assumption of a Java processor running Java bytecode, followed by control flow analysis to transform the bytecode into high-level data structures. WCET analysis algorithms are then built on top of these structures.

The annotated control flow analyzer of Chapter 4, together with the hardware and software platforms of Chapter 3, provides a foundation for interactive analysis. The next step is to build upon this foundation by adding worst-case execution time analysis, as shown in Figure 5.1.

Knowing the WCET is essential for interactive analysis, which mandates continuous and interactive feedback in the coding cycle to detect timing errors the moment they occur. The goal is to weave knowledge of WCET into every thread of software development, from the moment the first line of code is written, to allow time-sensitive programs to evolve without sacrificing guarantees on their predicability.

Even without this notion of interactivity, WCET analysis is vital to the design of any hard real-time system. No scheduling algorithm, for instance, can provide valid results without this analysis. It is the very bedrock of real-time system theory. In practice, however, WCET analysis is often neglected or merely "guesstimated." Although WCET research is improving, and commercial tools have helped, analysis of non-

trivial programs remains limited and *ad hoc*. Even when an analysis is performed, it may not provide a true guarantee, leading to unsafe software, or the results may be overly pessimistic, leading to a waste of resources.

Consider, for example, a real-time system that must sample an analog sensor every 50 milliseconds. Static analysis of this system for a particular processor might declare its worst-case execution time to be 40 milliseconds, indicating that the code can always meet its 50-millisecond deadline. If the analysis were not as thorough as it should have been, however, and the true WCET were in fact higher than 50 milliseconds, then the system could overshoot its deadline at some point and fail to sample the sensor. This is an example of an *unsafe* WCET analysis.

Alternatively, consider the same situation if static analysis had instead indicated that the WCET were 80 milliseconds. The system designer might choose to double the speed of the processor, pushing the WCET down to around 40 milliseconds and thereby meeting the deadline. While this tactic would work (assuming that the analysis was indeed a true upper bound on WCET), increasing the processor speed would be pointless if the analysis had been overly pessimistic, and the true WCET on the original processor was actually less than 50 milliseconds. The extra cost of the faster processor would then go to waste since it would be reducing the worst-case time unnecessarily.

In a more extreme situation, the analysis could have been so conservative, and its upper bound on WCET placed so high, that no processor in the world would be fast enough to reduce the worst-case time to less than 50 milliseconds. For these reasons, a safe WCET analysis is not enough; it must also be *tight*. Figure 5.2 illustrates these trade-offs.

Unfortunately, placing an upper bound on execution time that is both safe and tight

Figure 5.2: This diagram illustrates the effects of safety versus tightness in WCET analysis. Note that in the safe-but-not-tight analysis, the system designer will somehow have to reduce the WCET of the task, even though no reduction is necessary.

is exceedingly difficult. The mercurial behavior of modern processors, due to caching and other factors described in Section 2.2, is only part of the problem. Another key factor is an insatiable thirst for abstractions: Software development thrives on abstraction to simplify almost everything, transforming what would otherwise be a mammoth effort into a manageable task.

In most software systems, even time is an abstraction, causing tremendous complications for WCET analysis. In the dynamic memory heap, for example, an allocation request could complete almost immediately, or it could be substantially delayed in the event of a page fault. The non-real-time software developer usually does not care, as long as the overall speed is adequate. Not having to worry about the details of memory management is the payoff for this abstraction, but as a side effect, its variance between best-case and worst-case times is enormous. Accounting for such abstractions is so difficult that execution time on many processors and software platforms is effectively unpredictable.

To attack this problem, Chapter 3 proposed a restriction on the software and hardware platform of a real-time system. Selecting Java processors for deployment and the Java language for code creation simplifies the entire WCET analysis process. In particular, running code on a Java-based processor such as the JOP eliminates many sources of

unpredictability, thereby improving the tightness of a WCET estimate.

Yet even with these simplifying assumptions, a static computation of WCET is still difficult. In fact, it is theoretically impossible due to the Turing halting problem [169]. Knowing whether an algorithm halts is undecidable, and consequently, the WCET of an arbitrary program is always infinity. This particular conundrum, however, is well-known in the field of WCET research, and to work around it, two additional restrictions on a real-time system are commonly put into place:

- All loop statements must have bounded iterations. That is, the programmer, relying on knowledge of the program's run-time conditions, must tell the analyzer how many times each loop would execute in the worst case.
- As a corollary to the previous requirement, recursive function calls, which are a special kind of loop, are prohibited.

These two restrictions alone solve the halting problem, but they are not enough to make tight WCET analysis tractable. The usual abstractions found in traditional software development conspire to make the goal extraordinarily complex. As a result, two more simplifying assumptions are typically conceded:

- Dynamic memory heap allocations, as well as garbage collection to free them, are disallowed. This restriction solves much of the abstraction problem, given that the highly variable delays caused by abstraction are most often a result of heap maintenance.
- Dynamic dispatch of method invocations, found in object-oriented languages like Java, is prohibited. Although this restriction is not a consequence of the halting problem, mechanisms for dealing with polymorphism in WCET analysis are inadequate and remain an ongoing area of research [170, 171, 66].

These assumptions peel away the layers of abstraction that make WCET analysis so difficult. Although they limit software design to an austere style, at the same time they open doors to new ways of validating real-time systems, including interactive analysis techniques. Future advances in WCET research may very well loosen some of these restrictions, but for the moment they are necessary to enable WCET analysis of an entire program implemented in a modern, general-purpose programming language.

The following sections examine prior work in WCET analysis techniques based on these assumptions. They also explore new ways of extending the techniques to make them *interactive*. The focus is not on tight WCET analysis but rather on integration of analysis into the software development life cycle. By improving the practicality and usability of these techniques, industry adoption of WCET analysis will likely increase, making real-time systems safer and more reliable.

5.1 The Theory of WCET Analysis

Section 2.2 described the basic concept of WCET for a single statement: a non-branching, straight-line block of code. Naturally, real programs have many such statements, and the flow of control jumps to and from them. Manual computation of WCET quickly becomes unfeasible except for the most trivial code, and perhaps for simple assembly language programs such as the one in Figure 2.4. Some sort of automated approach for finding WCET is a necessity.

Hunting down this automated approach has been a formal topic of research since the 1980s, starting with Kligerman and Stoyenko's description of analyzing Real-Time Euclid programs [24]. Since then, the challenge of WCET has spawned hundreds of research papers as well as a yearly workshop focused specifically on the problem. (It

is called, quite unsurprisingly, the International Workshop on Worst-Case Execution Time.) Several worthwhile papers offer summaries of the most significant advances in these pursuits:

- In a guest editorial, Puschner and Burns look back on the first decade of WCET analysis research, examining special achievements and recent advances [172]. They note that after ten years of research, there is still confusion between execution-time analysis and response-time analysis.
- Engblom et al. provide a broad overview of the established WCET analysis methods as of 2003 [173]. A noteworthy contribution of this work is a feature-by-feature comparison of the major research efforts.
- In his Master's thesis, Stoif breaks down the known WCET analysis techniques, covering everything from object-oriented language issues to cache and pipeline concerns [174].
- In an updated version of Puschner's editorial, Wilhelm et al. survey the techniques for WCET analysis from a more recent perspective [175]. A subtle but significant variation in this paper is its focus on end-user tools rather than on the analysis methods themselves. (When Puschner wrote his original editorial in 2000, no such tools existed.)

These papers show that the topic of WCET analysis has attained a certain level of maturity, reaching a point where all known approaches can be classified into three phases: *control flow analysis*, *low-level analysis*, and *longest path computation*. This section provides an overview of these phases with a scope limited to the assumptions presented on page 107. It focuses specifically on building a theoretical foundation for interactive analysis techniques. It also pays special attention to the role of Java in WCET analysis, supplementing the discussions in Chapter 3.

It does not, however, examine any measurement-based approach to finding WCET. Although survey papers document such techniques, this section rejects them for safety reasons. Measurement is risky because it cannot guarantee that all inputs the system will encounter are tested, as illustrated by Figure 2.2. Attempting to measure all possible inputs would fail due to the huge search space involved. For example, a single stateless function with three 8-bit integers as inputs would require more than 16 million different tests. Even at a rate of 10,000 tests per second, computing the WCET with this method would take 27 minutes, an untenable amount of time if interactive analysis is the goal. And if the integers were 32 bits wide, as is usually the case in modern high-level languages, the calculation would last two quintillion years.

Despite the inadequacy of measurement in finding a safe WCET bound, it does solve the problem of tight WCET analysis, and for this reason it remains an active area of investigation. Some researchers believe that a hybrid of static and measurement-based approaches, combined with formal probability analysis, is the best way (and, some would say, the only way) to achieve a tight bound on WCET [93, 176, 177]. This chapter, however, does not consider such techniques because they cannot offer a true guarantee on timeliness.

5.1.1 Control Flow Analysis

Prior to any calculation of a WCET bound, information about the feasible execution paths in a program must be collected. This step is called *control flow analysis* but may also be referred to as *high-level analysis*, *program flow analysis*, or simply *path analysis*. The purpose of this analysis is to collect flow information, also known as *flow facts*. Identifying precise flow information normally requires some form of control flow reconstruction directly from binary code.

For example, a WCET analyzer for Java might parse the bytecode of a class file, perhaps with a utility such as BCEL [136]. By examining the destination address of branching instructions in the bytecode, the analyzer can generate a model of the program's high-level constructs: loops, if statements, and basic blocks. Thus, the input to the control flow analysis step is a sequence of binary code, and the output is a data structure representing the flow of control through that code.

However, a perfect reconstruction of flow is, in the general case, impossible. For programs containing loops, the flow is essentially unbounded; an algorithm cannot know how many times a loop will iterate in the worst case. While recent work has proposed abstract interpretation as a means of finding certain loop bounds automatically [178, 179], the more common approach is to rely on source code annotations.[1] These annotations must be inserted manually when writing the loop constructs, so they require more effort from the developer and are error-prone, but they make WCET analysis much faster and simpler. Section 6.4.2 will examine the topic of loop annotations in more detail.

For the Volta suite of interactive analysis tools, the component called Cascade, introduced in Chapter 4, takes care of control flow reconstruction. Purely a control flow analyzer and ignorant of any data flow information, Cascade depends on loop annotations to form a complete model of the program under analysis. This particular limitation is permissible according to the assumptions on page 107.

A unique advantage in Cascade, in comparison to most other control flow analysis tools, is its focus on Java. Normally, identifying execution paths in machine code is difficult due to optimizing compilers rearranging and transforming code to gain

[1] An *annotation* in this context refers to a metadata facility that allows source code elements to be marked as having a particular attribute. The term should not be confused with the idea of annotating the elements of a control flow data structure with source code expressions, as discussed in Chapter 4.

higher execution speed and lower memory consumption. Java compilers, however, do not perform any such optimizations at the bytecode level. Tools like Cascade are therefore able to map bytecode to high-level control flow constructs more easily. This advantage allows fast and efficient WCET analysis, as required for interactivity.

5.1.2 Low-level Analysis

Once the control flow has been reconstructed and loop bounds obtained, the process continues with *low-level analysis*, also known as *execution-time modeling*, or simply *exec-time modeling*. This step assigns each instruction in the program an execution time. The microsecond values in Figure 2.4, for instance, must be derived from a model of the target processor. Low-level analysis provides this deep knowledge without consideration of the global control flow and data dependencies of the program.

In actual implementations, low-level analysis may be combined with the longest path computation step (described in Section 5.1.3), but it is frequently treated as a separate topic pedagogically because of its complexity. Pipelining and caching effects, for example, mean that the timing of an individual instruction may not be constant but could vary on each execution. The complications of these effects force many papers on the subject to focus on just one component of the problem.

In this simplified discussion, low-level analysis is broken down into two components: *instruction timing* and *instruction caching*.

Instruction Timing

Multi-level caches, branch prediction, and out-of-order execution induce a processor state whose exact value depends on a large execution history. Modeling this history

leads to a state explosion for the final WCET calculation. As a result, low-level WCET analysis usually requires simplifications of the CPU model, producing an excessively conservative estimate.

A novel solution, as discussed in Section 3.4, is to rely on a Java-specific processor such as JOP, which strikes a balance between average case performance and ease of WCET analysis. JOP avoids complex features like pipeline dependencies, prefetch queues, and automatic stack dribbling, as found in other Java processors [100]. As a consequence, there are no timing dependencies across bytecode boundaries, and pipeline analysis [180] can be omitted. The rules to aggregate timing values [181] can be applied without introducing significant conservatism.

This simplicity in Java processors is an asset for low-level WCET analysis. It can ignore execution history and processor state, deriving the cycle count for each bytecode instruction in isolation. Furthermore, low-level analysis need not consider how bytecode will be translated into native instructions, as there is no just-in-time compilation on Java-specific processors.

On the JOP, for instance, the cycle count for the GETSTATIC bytecode instruction is $12 + 2r_{ws}$, where r_{ws} is the number of wait states for a memory read. Almost all bytecode timings can be computed with a simple formula such as this, assuming that the instruction is already in the instruction cache. The only input variables are the instruction opcode and the memory wait states.

Instruction Caching

These simple instruction formulas are due in part to the lack of a true data cache on the JOP, unlike other processors that include both instruction and data caches. Even in resource-constrained embedded systems, these caches are often mandatory due to

the growing gap between processor performance and memory access time. Of course, they also complicate the entire WCET analysis process. Significant effort has been expended on modeling the caches for WCET analysis [33, 32, 31], but the problem remains an active area of research.

As a compromise between memory performance and ease of analysis, the design of JOP introduces two time-predictable caches: A *stack cache* [182] to speed up access to variables and operands on the execution stack, and a *method cache* [95] as a special kind of instruction cache. (Access to the heap is not cached.)

The stack cache is a simple two-level on-chip memory. The two top-most elements of the stack are held in registers, and the subsequent elements are stored in on-chip block RAM. There is no automatic exchange between on-chip RAM and the main memory, as in picoJava [100], which would introduce complex timing interactions between instructions. The exchange is under microcode control and can be restricted to method invocation or thread switching.

For caching of instructions, the nature of Java is uniquely beneficial for WCET analysis. Specifically, bytecode instructions that jump outside of a method boundary do not exist. As long as the entire method is loaded into the cache, every branch instruction in that method will be a cache hit. This observation greatly simplifies low-level analysis because the context of the instruction, such as whether it lies within a loop, can be ignored completely without neglecting the effects of the cache. The instruction cache only needs to be considered when invoking a method or returning from one.

JOP's instruction cache takes advantage of this fact. Instead of a traditional block-based cache, which may contain only portions of a method, JOP's cache is method-based. It always stores complete Java methods, never portions of one. It is filled only on an invoke or a return instruction; all other instructions are a guaranteed cache hit.

JOP's method cache is also unique in that its architecture can vary according to the desired sophistication of WCET analysis. In sharp contrast to block-level instruction caches, its most basic form is the *single method cache* [95], in which the total size of the cache matches the size of the largest method to be executed. In this configuration, every method invocation and return is a guaranteed miss, as shown in Figure 5.3. (Note that this is still a caching solution because it converts all non-invocation and non-return instructions into cache hits.)

Although the single method cache makes WCET analysis simple and fast, the cache miss on every invocation and return causes a substantial slowdown, especially given the large number of method invocations in typical Java software. To increase the cache hit ratio, JOP can also be configured to store more than one method at a time. For example, a *dual method cache* [95] stores two methods at once using a least-recently used replacement strategy. As before, both areas of the cache must be large enough to hold the largest method in the program. (The largest method could end up in either area, depending on the call graph.)

While the dual method cache improves performance, it also complicates WCET analysis. Whether a method invocation is a hit or miss depends not only on the structure of the program but on the input data, as well. For example:

```
for (int i = 0; i < 10; i++)
    if (i % 3 == 0) methodA();
    else methodB();
```

Without elaborate data flow analysis to determine when `i % 3 == 0`, a WCET analyzer must assume that the invocations of `methodA` and `methodB` are always misses. Otherwise, it cannot guarantee a safe estimate.

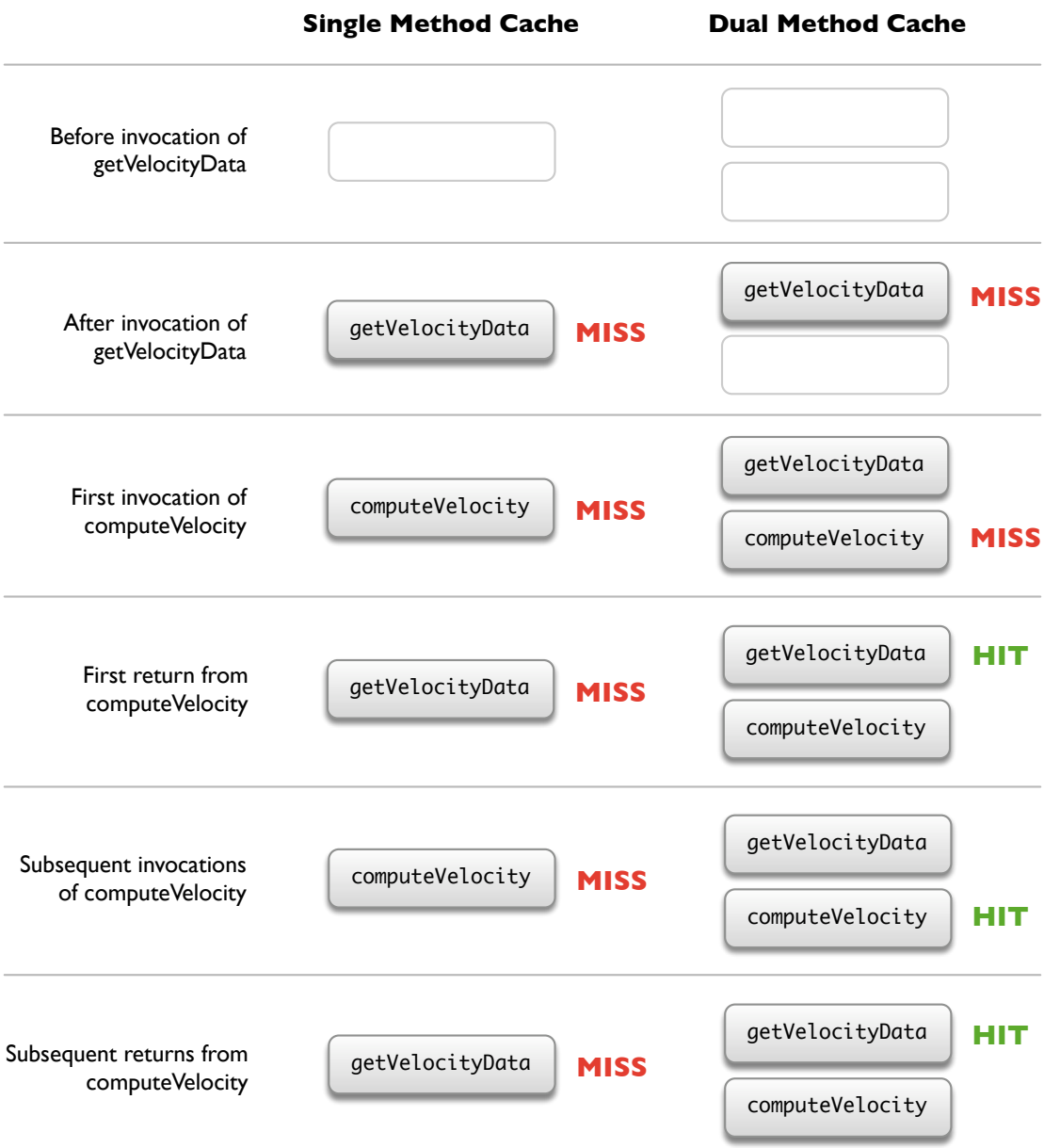

Figure 5.3: A pure control flow analysis can identify guaranteed hits and misses of JOP's method cache in certain situations. This diagram shows one such situation: a leaf method within a loop, as in Figure 4.6. All invocations of computeVelocity are misses in the single method cache, but only the first invocation is a miss for the dual method cache.

Fortunately, other code structures are more amenable to analysis. If, for example, the call to methodB were removed, and methodA makes no further invocations, then two guarantees can be made: the invocation of methodA and the return from methodA will *always* be cache hits (except for the first iteration when methodA is loaded into

the cache, as shown in Figure 5.3).

By identifying such structures in the code—that is, a loop that executes only one kind of method—analyzers can improve their estimate of the WCET. Section 6.4.4 describes how to to achieve these improvements in the context of interactive analysis.

5.1.3 Longest Path Computation

Once the control flow has been constructed and a means of low-level analysis identified, the core computation begins. The basic idea is to search for the longest possible control flow path through the program. (It is somewhat like the shortest path problem from graph theory in reverse.)

This step, which is known as *longest path computation*, *longest path search*, or *calculation of execution scenarios*, identifies feasible paths derived from control flow analysis and assigns execution times to instructions derived from low-level analysis. Sufficient information is then present to compute the final WCET bound.

Existing techniques for performing this computation come in three varieties: a tree algorithm, a path enumeration algorithm, and an implicit path enumeration algorithm. The first form is based on recursing through a tree representation of the control flow, while the latter two forms are based on control flow graph searching. (For additional work comparing these three algorithms, refer to Engblom, Ermedahl, and Stappert [183, 184].)

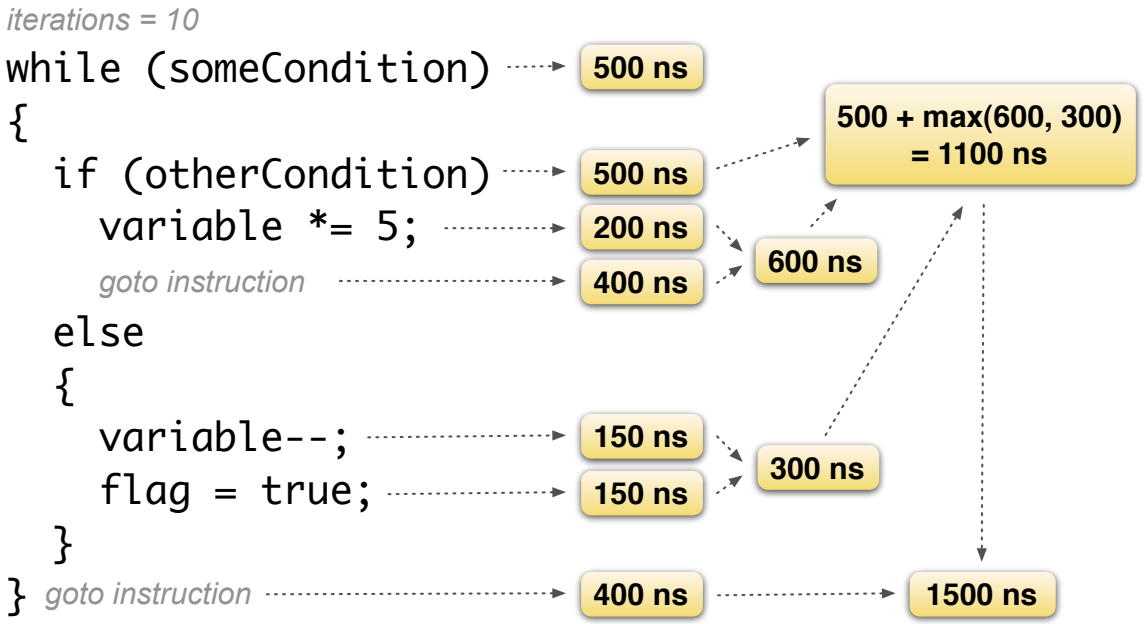

Figure 5.4: A tree-based WCET algorithm recurses to the leaves of a control flow tree and returns the sum for each node. The execution time for a basic block is simply summed, but when a branch is encountered, the worst-case time among the possible paths is returned, resulting in a WCET of 1500 for the loop body shown here. The body is then multiplied by the maximum number of iterations for a final WCET of 500 + 10 (1500 + 500) = 20,500 nanoseconds.

The Tree Algorithm

The tree-based approach, also known as the *structural* approach, was among the very first implementations of WCET analysis [185, 186]. It operates by recursively descending the nodes of a program's control flow tree, returning the execution time for each node. The value returned for the root node is the total WCET. Figure 5.4 shows an example of this concept for a simple loop.

As the algorithm encounters each control flow node, it must decide how to compute the WCET based on the node's type. For straight-line code, the time to execute each instruction is simply summed. For branches (if and switch statements), the path whose execution time is highest—the "worst" path—is taken as the total time. For loops, the maximum number of iterations is multiplied by the WCET of the loop's

```
getWCET(node)

  if node is null return 0

  if node is IF_THEN_ELSE
    cycles = getExprWCET(node.conditional_expression) +
             max(getWCET(node.then_branch), getWCET(node.else_branch))

  else if node is LOOP
    expression_cycles = getExpressionWCET(node.conditional_expression)
    cycles = expression_cycles +
             (getWCET(node.loop_body) + expression_cycles)
                 * node.loop_bound

  else if node is STATEMENT
    cycles = getExpressionWCET(node.statement_expression)

  return cycles + getWCET(node.next)

getExpressionWCET(expression)
  return sum of CPU cycles of all instructions in the basic block
```

Figure 5.5: This program listing is pseudocode for a generic tree algorithm. The algorithm assumes that the control flow is represented as a tree structure, such that branches (if statements) and loops are in child nodes, and subsequent statements are in sibling nodes. A null sibling indicates that there are no further statements on the given node's lexical level.

body. Figure 5.5 provides pseudocode for this generic tree algorithm.

In addition to being relatively easy to write and to understand, tree-based algorithms have benefits that are not so commonly recognized. Raw speed is one example. The expected running time of a recursive descent to determine WCET is $\theta(n)$, where n is the number of nodes in the control flow tree. (In contrast, graph-based algorithms are theoretically NP-hard problems.)

Despite these advantages, the tree-based algorithm has fallen out of favor among some WCET researchers. It suffers from certain drawbacks, such as a susceptibility to the false path (also known as infeasible path) problem, in which data dependencies between two if statements can fool the algorithm into computing an overly pessimistic

WCET [187].

To illustrate, consider the program listing in Figure 5.6. A quick scan reveals four possible paths through the code:

- A $\rightsquigarrow$ C (WCET $\approx$ 1010)
- A $\rightsquigarrow$ D (WCET $\approx$ 20)
- B $\rightsquigarrow$ C (WCET $\approx$ 2000)
- B $\rightsquigarrow$ D (WCET $\approx$ 1010)

On closer inspection, however, one of these paths is actually a *false* path. The path from B to C is impossible because `x` cannot be both less than zero and greater than 42. A purely structural analysis algorithm would ignore this fact and conclude that the WCET for the program is 2000 (not counting the time for the `if` conditionals), nearly double the true WCET of 1010. The algorithm therefore produces a *safe* value but not a *tight* one.

In general, tree-based algorithms have difficulty handling any type of dependency across sibling nodes due to the nature of tree traversal. With sufficient effort, however, solving these tricky situations in tree-based analysis is still possible. Colin [188] describes how a tree-based algorithm can work around the false path problem, for instance.

The Path Enumeration Algorithm

Another approach for computing the longest path is to enumerate all possible paths through a control flow graph and simply select the longest of the set. The idea is to

```
if (x >= 0)
{
    // A: WCET = 10
}
else
{
    // B: WCET = 1000
}

if (x > 42)
{
    // C: WCET = 1000
}
else
{
    // D: WCET = 10
}
```

Figure 5.6: This simple program demonstrates the *false path* problem in WCET analysis. The worst-case execution times for the four blocks of code—A, B, C, and D—are shown in the comments. The true WCET is 1010 (not counting the conditionals), but the tree analysis algorithm described in Figure 5.5 would compute 2000.

take advantage of standard graph algorithms and data structures for the computation. For example, Altenbernd adapted a recursive branch-and-bound graph algorithm to compute the maximum-delay-to-sink for all vertices [187].

These *path enumeration* algorithms (also called *path-based* algorithms) share some of the tree-based approach's weaknesses, such as a susceptibility to the false path problem. As before, this weakness can be addressed by adding more complexity to the algorithm. Stappert describes one such technique whereby paths deemed infeasible are pruned from the set [189].

Unlike tree algorithms, however, path enumeration algorithms are generally unable to contend with loops. Loops appear frequently in control flow graphs, but standard graph algorithms typically cannot handle bounded loops. (In Stappert's work, loops were simply prohibited, exploiting the observation that synthesized real-time code usually has a simple structure absent of loops.) When loops are present, path enu-

meration becomes very computationally inefficient: The number of paths to explore tends to grow exponentially with the number of control flow branches [190].

These drawbacks, combined with the lack of any major advantages over tree-based approaches, have stifled the growth of path enumeration as a viable algorithm for WCET analysis.

The Implicit Path Enumeration Algorithm

In an effort to solve thorny analysis problems such as false paths, an alternative algorithm known as *implicit path enumeration* has emerged [190, 191]. The technique is based on the observation that the only objective for most applications of WCET analysis is to determine worst-case time. Identifying the actual worst-case path through the code is usually unimportant.

With this observation, WCET computation reduces to an instance of a well-known problem from graph theory: Finding the maximum feasible flow through a single-source, single-sink directed acyclic graph. This "max flow" problem can be solved by a variety of techniques: Ford-Fulkerson, Edmonds-Karp, push-relabel, and others.

The graph in this case is the control flow graph of a computer program, and the flow capacity of an edge is simply the instruction time of a basic block, as determined by low-level analysis. The single-source requirement is clearly met, since programs always start from exactly one location, as is the directed graph requirement, since programs cannot run in reverse. Less obvious is the fact that control flow graphs also satisfy the single-sink requirement because, while there may be multiple `return` statements in a program, they can all flow to a single "exit" vertex. (The WCET of the edges between the `return` statements and the exit are set to zero.) The acyclic requirement, however, is one that control flow graphs do *not* satisfy. Any program

with a loop construct will have cycles, defeating algorithms such as Ford-Fulkerson.

Luckily, one of the tactics for solving max flow problems can handle cycles with aplomb. Called integer linear programming, or ILP, it solves the loop problem by defining maximum flow according to *constraints* on the legal flow through the graph. Accounting for a loop is simply a matter of adding an additional constraint to bound the amount of flow—that is, the number of iterations—through the loop.

Computing WCET then becomes a problem of translating the control flow graph into a series of constraint equations and running these equations through an ILP solver to find the maximum. The paths through the graph are never actually explored; they are derived implicitly from the constraints, thus giving rise to the term *implicit path enumeration technique*, or simply IPET.[2]

To illustrate, consider the control flow graph of Figure 5.7, whose source code is given in Figure 5.4. An IPET-based algorithm would add constraints for each vertex in this graph, as shown in Figure 5.8. The constraint on Block 7, for example, is $edge7 + edge8 = edge9$, indicating that the incoming flow is equal to the outgoing flow. To account for the loop, an additional constraint for Block 2 is added: $10 \cdot edge10 = edge2$, indicating that the flow through Edge 2 is ten times as large as the flow through Edge 10. Feeding these constraints to an ILP solver would show that the maximum value of the objective function is 586, meaning that the WCET of Figure 5.4 is 586 cycles.

An important point to remember is that a tree-based algorithm would produce exactly the same result. It would also find the result more quickly, since the time it requires grows linearly as the complexity of the analyzed program increases. By comparison,

[2]Some research papers refer to implicit path enumeration as the "ILP technique," but strictly speaking, this is a misnomer. Integer linear programming is only the method used to find the maximum flow, not the technique itself. Methods other than ILP could be applied to IPET, as long as they can handle bounded cycles in the graph.

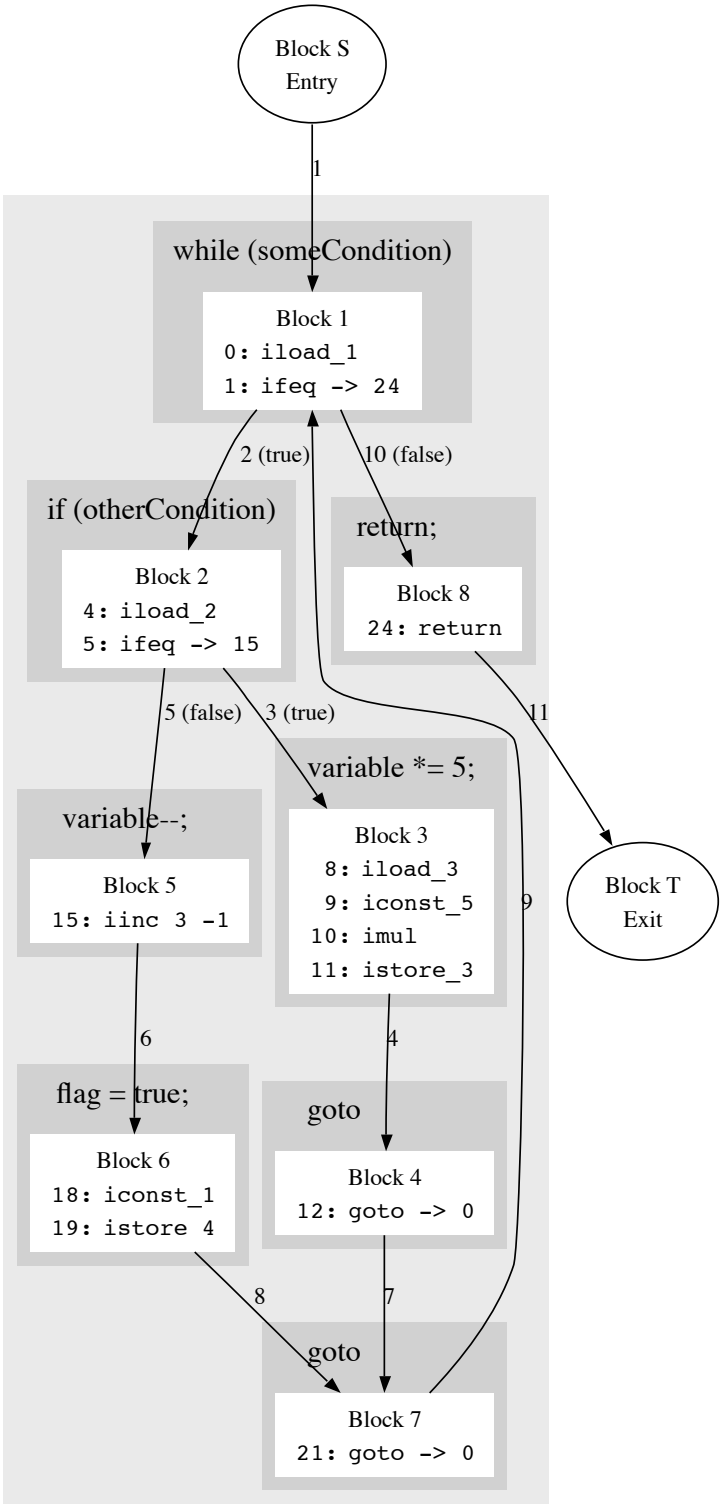

Figure 5.7: IPET algorithms operate by finding the maximum flow through a control flow graph, such as the one shown here of the source code in Figure 5.4.

```
/* Objective function */
max: +block1 +block2 +block3 +block4 +block5 +block6 +block7 +block8;

/* Edge constraints */
+edge1 = 1;
+edge11 = 1;
+edge1 -edge2 +edge9 -edge10 = 0;
+edge2 -edge3 -edge5 = 0;
+edge3 -edge4 = 0;
+edge4 -edge7 = 0;
+edge5 -edge6 = 0;
+edge6 -edge8 = 0;
+edge7 +edge8 -edge9 = 0;
+edge10 -edge11 = 0;

/* Loop constraints */
-10 edge1 +edge2 = 0;

/* Basic block constraints */
+5 edge1 +5 edge9 -block1 = 0;
+5 edge2 -block2 = 0;
+38 edge3 -block3 = 0;
+4 edge4 -block4 = 0;
+8 edge5 -block5 = 0;
+3 edge6 -block6 = 0;
+4 edge7 +4 edge8 -block7 = 0;
+21 edge10 -block8 = 0;
```

Figure 5.8: To find the maximum flow through a control flow graph, IPET algorithms transform it into a series of constraint equations that are then fed into an integer linear programming solver. In this example, Figure 5.7 has been expressed as an ILP problem in lp_solve [192] format.

integer linear programming can be quite slow; it is an NP-hard problem that grows exponentially [193]. This slothful performance makes IPET impractical for the kind of interactive analysis described in Section 5.3.

Because IPET produces the same value that a basic structural algorithm gives for the code in Figure 5.4, and it does so more slowly, it would be virtually useless if not for one powerful advantage: Adding constraints to an ILP formulation is almost trivial. The implication is that IPET can handle the false path problem quite elegantly. To do so, one only needs to supply a constraint indicating that the infeasible path

(e.g., B $\rightsquigarrow$ C in Figure 5.6) has zero flow capacity.

Knowing what constraint to add, however, is *not* trivial. Automatic discovery requires extremely complex data flow analysis, while a manual approach—via source code annotations, for instance—is vulnerable to human error and adds an additional burden on the user. Unless one of these two approaches is put into place, IPET offers no improvement and degrades to the same level of pessimism as a tree-based algorithm.

5.2 The Practice of WCET Analysis

These techniques for static computation of WCET have been circling academia for over a decade, but for the most part they have failed to migrate into industry practice. Even the very idea of static analysis, for any purpose, is still somewhat uncommon outside of the research community [194]. The medical device industry, for example, has only recently begun to embrace static analysis as a mechanism for detecting software flaws [195].

In anticipation of a coming acceptance of static analysis in general, and static WCET analysis in particular, researchers have created various prototypes of software tools that allow end users to determine the worst-case performance of their code. The majority of these tools target C, while a small subset are able to analyze Ada and some less popular languages. A growing number target Java, as discussed in Appendix A.

Compared to other categories of software tools, the number of these WCET analyzers is actually quite small. Only a dozen or so have ever been built, and a mere fraction of those are still actively maintained. This section highlights a few of the tools in this rarefied group, demonstrating how the techniques of Section 5.1 have been put

into practice. (For a more comprehensive and detailed feature comparison of current WCET analysis tools, refer to Wilhelm et al. [175].)

5.2.1 Research Prototypes

One of the earliest manifestations of WCET analysis as a stand-alone tool is *Cinderella* (see Figure 5.9), named in honor of the fairy tale heroine who had a critical real-time deadline. Cinderella is also notable as the first tool to support IPET-based analysis. Consisting of approximately 10,000 lines of C++ code, it takes as input an executable file, its source code, and user-specified functionality constraints (in order to deal with the false path problem, for instance). It then searches for loops and prompts the user to enter iteration bounds for each one it finds. Finally, it computes the WCET for a single function based on a model of the Intel i960 processor. Pessimism for Cinderella is on average about 100% according to benchmarks provided by its authors. (That is, its predictions are about twice as large as the true worst-case times.)

One of the usability weaknesses of Cinderella is that it stores loop bound information separately from the source code. Tools such as the Hades Embedded Processor Timing ANalyzEr [186], or Heptane, popularized the idea of source code annotations, which help maintain the accuracy of loop bound declarations by binding them to their loop definitions. For example:

```
for (b = 2; b <= NumSamples; b = b << 1) [11, pow(2, (i + 1))]
    for (i = 0; i < NumSamples; i += b) [(2048 / nlast(P, 1))]
        for (j = i, n = 0; n < BlockEnd; j++, n++) [nlast(P, 2) / 2]
            ...
```

Heptane's annotations are quite sophisticated because they can express boundaries

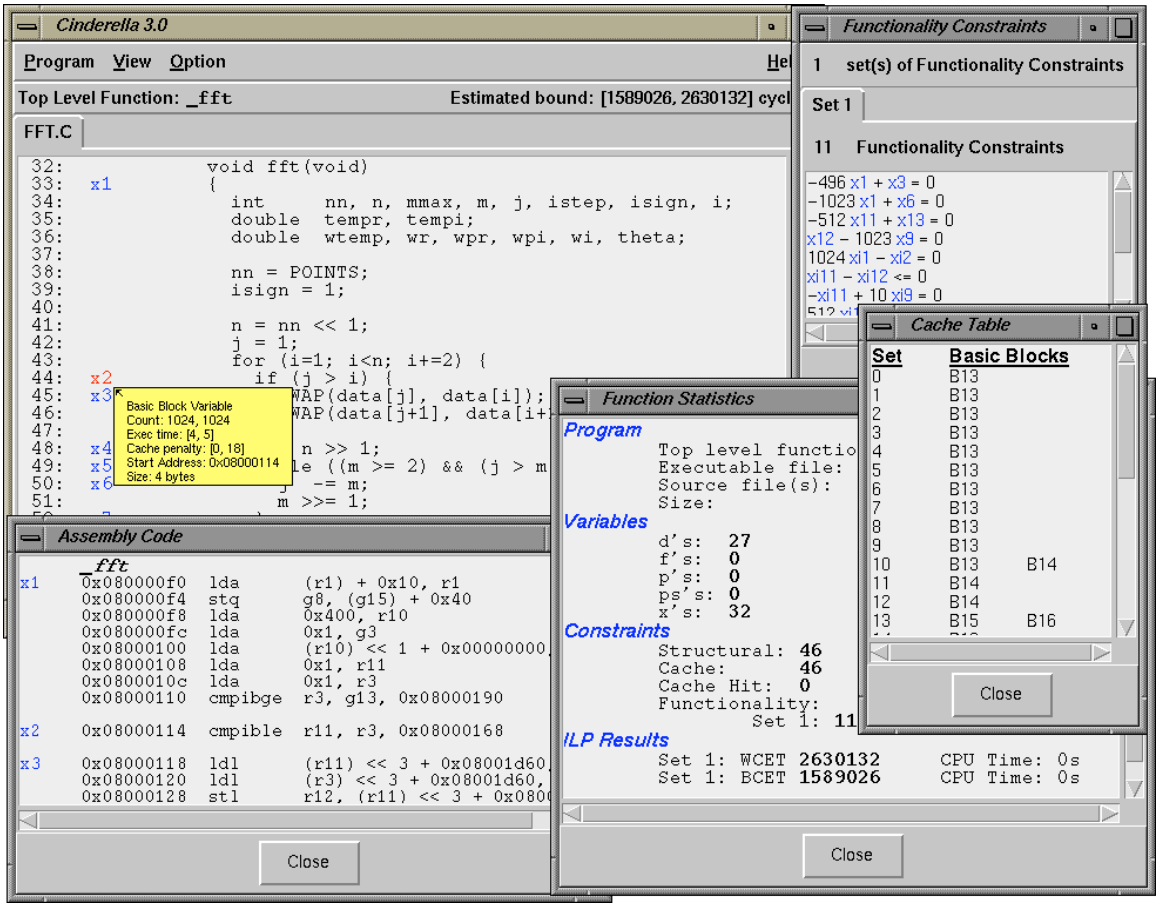

Figure 5.9: This screenshot shows Cinderella, the first stand-alone tool designed for static WCET analysis, calculating the best-case and worst-case time for a given C function.

for non-rectangular loops—that is, nested loops whose inner loop bound depends on the index of the outer loop index. If the annotations specified only a simple constant for the bound, such loops would cause a pessimistic WCET because the true worst-case bound would be much less than the annotation. Instead, Heptane supports expressions, such as nlast(P, 2), which refers to the upper bound of the second surrounding loop.

As powerful as these symbolic annotations are, they still depend on manual intervention by the developer. A simple human error when typing the annotation could lead to an unsafe WCET. More recent tools attempt to determine loop bounds au-

tomatically and prevent such mistakes. TuBound [196], for instance, performs loop bound analysis to determine the IPET constraints for certain kinds of loops, then inserts these constraints directly into the source code for later processing by a WCET analyzer.

As a possible sign of maturation in the field of WCET analysis, TuBound is constructed in a modular fashion, as are many of its contemporaries. For example, modularity in the SWEdish Execution time Tool [197], or SWEET, enables different analysis algorithms and timing models to work together independently. They communicate through well-defined data structures that represent the control flow graph and processor timing model. SWEET supports path enumeration, implicit path enumeration, and a hybrid of the two known as the *clustered* technique [184]. Likewise, the Open Tool for Adaptive WCET Analysis [198], or OTAWA, is intended to support many different approaches, including future techniques that have hardly been explored, such as how to model multiple-issue pipelines and multithreading.

Another tool worthy of note is Chronos [199], which offers some of the most detailed micro-architectural modeling yet available, including out-of-order pipelines and dynamic branch prediction. It is also flexible because it relies on the SimpleScalar simulator that can be configured to handle different processor architectures. Chronos is distributed under an open-source license, allowing extensibility to new features and estimation techniques. Figure 5.10 shows a screenshot of Chronos in action.

5.2.2 Commercial Tools

The long-term goal of almost any of these research prototypes is to find a path to market. Ideally, a prototype should be able to grow into a commercial-quality piece of software that is as commonplace and user-friendly as the compiler, debugger, and

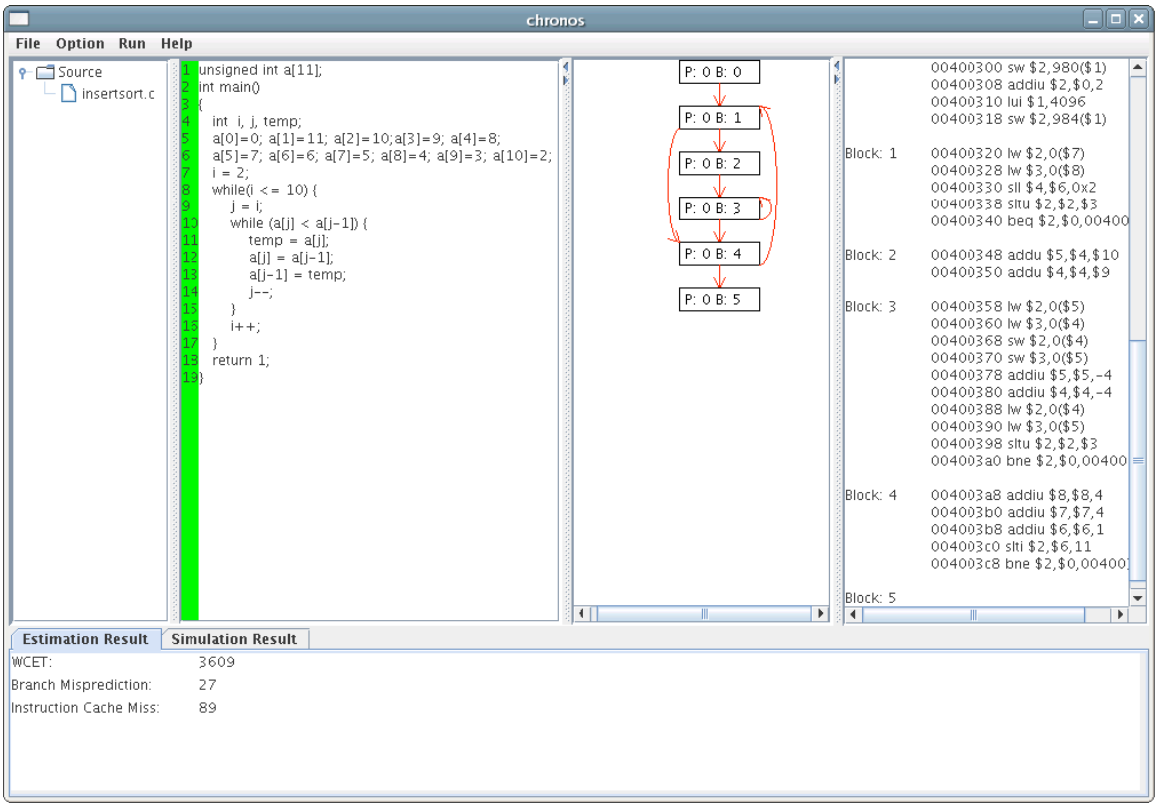

Figure 5.10: The Chronos analyzer, whose user interface is shown in this screenshot, takes into account branch prediction and instruction caching when computing WCET.

other tools found in a real-time developer's toolbox. In the realm of static WCET analysis, only a handful of prototypes have managed to make the jump from academic experiment to shrink-wrapped software: Bound-T, aiT, and RapiTime.

Originally developed for timing validation of on-board software in spacecraft, Bound-T [200] is the first static WCET analyzer to be sold commercially, having entered the market in 2001. It is based on IPET and is able to determine bounds automatically for simple counting loops. (Complex loops require special annotations from the user.) Although Bound-T can target different processors for analysis, it works best only with simple varieties, such as Intel's 8051 series of 8-bit microcontrollers, since it lacks support for cache analysis.

Arriving just two years after Bound-T, the aiT tool [34, 35] is similar in feature

set. It relies on IPET and can automatically detect certain types of loop bounds. For more dynamic sections of code, it requires annotations, but these can also be used to specify recursion depths and false paths. According to its manufacturer, aiT produces very tight WCET predictions with an average pessimism ratio of only 10% for the processors that it supports. The tool also includes several graphical modes for visualizing hardware states and control flow data, as shown in Figure 2.5.

In a departure from purely static WCET analyzers, the RapiTime tool [201] is built on the assumption that a complete and accurate timing model of a processor, as required for static analysis, is not always available. CPU manufacturers may not want to publish a detailed model of their processors due to trade secrets, or they may not see sufficient demand to justify the effort. Even when a model is available, it may be oversimplified, leading to pessimistic WCET estimates.

Instead, the developers of RapiTime argue that the best model of a processor is the processor itself. The tool relies on the statistics theory of copulas [176] to build up a model of the processor according to measurements of execution time of sub-paths in the code. It then performs offline static analysis to determine the overall paths. Finally, it combines the path analysis information with the measurement data to capture the worst-case execution time. This hybrid approach is probabilistic and therefore suffers from the risks of measurement, as discussed in Chapter 2, in that it may produce an unsafe WCET estimate.

5.3 Interactive WCET Analysis

Despite the sophistication and variety of these tools, they all suffer from a common weakness: Little or no thought is given to improving the *speed* of analysis, only its

accuracy. A typical example is an analysis framework created by Zhao et al. [202]. For small programs, it can calculate the WCET in seconds, but medium programs take minutes, and large programs take hours or even days [175].

The sluggishness of these tools means that interactivity is presently impossible. Real-time system development will never be made interactive if there continues to be a delay of hours, or even just minutes, during the coding cycle. Developers cannot afford to wait for a lengthy WCET analysis to complete after every change to a program. As a result, timing errors will remain invisible until the testing phase, too late for speedy correction during implementation. The goal of "bug prevention over bug detection" (page 30) is lost.

Instead, analysis of real-time software should happen *in real time*. Imagine a development environment in which WCET analysis of an entire program takes only a few seconds. What kind of impact would this have on making analysis more common, even routine? Would it lead to higher quality code? Would it make real-time software development easier and more accessible to a wider range of programmers? And are these programmers willing to give up some analysis accuracy in exchange for analysis speed?

This dissertation will not answer these questions. Proving beyond doubt that fast, interactive WCET analysis is demonstrably beneficial is a job left for future work, perhaps through a series of experiments in human-computer interaction. In the meantime, the advantages are assumed to exist, and the truly challenging question is *how* to make WCET analysis interactive. The remainder of this dissertation will present a possible answer and explore what form this interactivity would take. The belief is that focusing on the speed of analysis, not just tightness, will lead to new and exciting ways of building hard real-time systems.

5.3.1 Back-annotation

As a first step toward interactivity, the practice of WCET analysis needs to move beyond the assembly language level. Even when a real-time system is implemented in a high-level structured language, the WCET tools of today force the developer to think at a much lower level. For example, the aiT tool [35] displays timing data only in relation to executable machine instructions. The developer must digest assembly opcodes, hexadecimal addresses, and other implementation details in order to make sense of the analysis. The abstractions provided by a higher-level language are erased.

Ideally, a WCET analysis tool should instead have a deep integration with the source language. It should be able to *annotate* every statement in the source code with its worst-case execution time. By conducting an analysis at this high level, developers no longer need to switch periodically between source language and assembly language. They can remain focused on the original code, leading to a more natural and productive development process.

This concept, often referred to as *back-annotation* [193], is essentially a mapping from analysis results to source code, instead of merely indicating the worst-case time of the program as a whole. A back-annotation[3] of the code in Figure 5.4 would break down each individual statement to indicate, for example, that the source code lines of the else branch each have a 150-nanosecond cost.

The mechanics of back-annotation are similar in nature to integrated development environments such as Eclipse [162], where a Java compiler runs concurrently in the background to continuously check source code for compiler errors as it is typed. In a

[3]Once again, the term *annotation* is overloaded. Chapter 4 used the term to describe source code information attached to the nodes of a control flow data structure. Appendix B used it in the more traditional sense of a metadata facility for a programming language. In this case, the word means something slightly different; it refers to timing information that is woven into a view of source code or simply displayed in the margin of the view on the appropriate line.

similar fashion, a fast WCET analysis tool could run in parallel with a source code editor, providing immediate feedback to the developer, just as the compiler does. Instead of compiler errors, however, the feedback would come in the form of worst-case time information.

This agile approach is in stark contrast to the strict build-then-analyze style of traditional WCET tools. It aids developers in eliminating overly long critical paths—a common headache in hard real-time software—before they can occur. For example, Schoeberl and Pedersen describe a performance benchmark ("UdpIp") whose estimated WCET is 18 times larger than its observed WCET [135]. The authors note that the high degree of pessimism was due to the lack of WCET analysis during development of the benchmark. If interactive analysis had been available, the developer would have been more likely to detect the lengthy worst-case path and taken steps to reduce it, either by refactoring the code or redesigning the program.

5.3.2 Related Work

These ideas of interactivity and back-annotation are not entirely new. Prior work has danced around the concepts but has not sufficiently explored them. Too often, they are mentioned only peripherally or as an avenue of future work.

Occasionally, a true implementation will appear, but a key element will be lacking. Perhaps an interactive environment is presented, but it will operate at the assembly language level, not the source code level. Other times, back-annotation will be implemented, but there will be no mention of interactivity or of how fast the results can be produced.

One of the earliest attempts at interactivity came in 1996 when Ko et al. developed

a graphical interface for a WCET analysis tool [203]. The interface allowed the user to select a specific portion of source code for analysis, and the tool would then return the WCET of the selection. The primary innovation in this work was to allow specification and presentation of timing predictions at the source code level while retaining the accuracy of low-level analysis. Back-annotation was not provided, however, and there was no investigation into the speed of analysis.

Another prototype for an integrated development environment came from Ribeiro et al. [204]. The aim was to provide continuity in a real-time software project through the phases of implementation, debugging, and testing. A novel feature in the environment was a graphical display of control flow showing each source code element's contribution to the total WCET. This was one of the very first realizations of back-annotation, although the data was displayed in a separate window and was not integrated into an interactive source code editor. There was also no mention of the speed at which the graphic could be generated.

A step closer toward true back-annotation—that is, into the original source code, not as a separate visualization—arrived as a side-effect of Kirner's study of optimizing compilers in the context of WCET analysis [205]. A prototype tool chain was created to visualize WCET calculations as back-annotations into source code, as shown in Figure 5.11. However, the visualization was static text and was not integrated into a source code editor or other development environment. The calculation method also relied on IPET and was likely too slow for interactive analysis.

From the Java domain, the research prototype Skånerost was also an endeavor into interactive analysis [206]. This real-time software development environment combined WCET analysis and compilation to provide frequent feedback to the programmer, updating continuously as the source code changes. Skånerost made no attempt at back-annotation or decompilation and presented analysis results as raw bytecode

```
/* processor: m68000 */
/* memory wait states (r/w):  0/ 0 */
------ CYCLES(bubble) = 47034 ------
  1|-------------------------------#define N_EL 10
  2|-------------------------------
  3|-------------------------------
  4|-------------------------------/* Sort an array of 10 elements */
  5|-------------------------------void bubble (int arr[])
  6|  1|   16,    0(   1,   0) -{
  7|-------------------------------  /* Definition of local variables */
  8|-------------------------------  int i, j, temp;
  9|-------------------------------
 10|-------------------------------  /* Main body */
 11|  3|   24,    0(   1,   0) -  for (i=N_EL;
 12|  4|  228,  100(  10,   9) -       i > 1;
 13|  2|  216,   90(   9,   9) -       i--)
 14|-------------------------------       maximum (N_EL - 1) iterations
 15|-------------------------------  {
 16|  2|  180,    0(   9,   0) -        for (j = 2;
 17|  4| 3132,  900(  90,  81) -             j <= i;
 18|  2| 1944,  810(  81,  81) -             j++)
 19|-------------------------------             maximum (N_EL - 1) iterations
 20|-------------------------------        {
 21|16|13122,    0(  81,   0) -               if (arr[j-1] > arr[j])
 22|-------------------------------              {
 23|  9| 7614,    0(  81,   0) -                     temp = arr[j-1];
 24|14|11988,    0(  81,   0) -                     arr[j-1] = arr[j];
 25|  6| 6642,    0(  81,   0) -                     arr[j] = temp;
 26|-------------------------------              }
 27|-------------------------------        }
 28|-------------------------------  }
 29|  2|   28,    0(   1,   0) -}
```

Figure 5.11: A WCET analysis tool from Kirner [205], the output of which is shown in this figure, could back-annotate source code as a side effect of the analysis process. It was too slow to be interactive, however, and was not integrated into a development environment.

instructions.

In contrast to the custom development environments proposed by earlier work, Fauster et al. showed how to integrate WCET analysis into established, real-world source code editors [207]. Their prototype was based on the idea that the difference between best-case and worst-case execution times grows as the number of possible paths through the program grows. Therefore, reducing the number of paths would bring the BCET

and WCET closer together, thereby reducing pessimism of the analysis. To aid in this approach, they created a plug-in for vim, a popular source code editor, that would analyze code in the background and locate portions containing more than one possible control flow path. These portions were then highlighted in vim's window. The speed at which this process took place was never mentioned, raising doubt that it was truly interactive.

A rather unusual approach to interactive WCET analysis came from Zhao et al., who created a "WCET tuner" that was integrated with a compiler [208]. It was based on the observation that compiler optimization settings (e.g., space or time) have an effect on WCET, but the precise effect is difficult to predict. Sometimes it may increase the WCET; other times it may decrease it. Instead of manually checking all possible settings for the one that produces the best WCET, the authors created a tool that uses a genetic algorithm to narrow down the ideal optimization sequence. The tool is not interactive, though; it merely offers the user an opportunity to adjust the search parameters. There is no feedback once the algorithm begins its search.

Not one of these prior efforts focused on improving the speed of WCET analysis—a necessary ingredient for interactivity. Yu and Mitra were one of the few to recognize the deficiency of IPET in this respect [209]. Even when using ILOG CPLEX, a leading commercial ILP solver, they noted that solving a WCET problem can take an entire day. As a solution, they suggested changing the underlying hardware to support a WCET-friendly instruction set. Their experiments indicated that this change can, in some cases, reduce analysis time from 24 hours to just a few seconds. However, they did not consider application of this technique to interactivity and back-annotation.

Among commercial WCET analysis tools, RapiTime is the only product to offer a feature that comes close to back-annotation. The tool, shown in Figure 5.12, is able to color-code the worst-case path in the source code. The information does not include

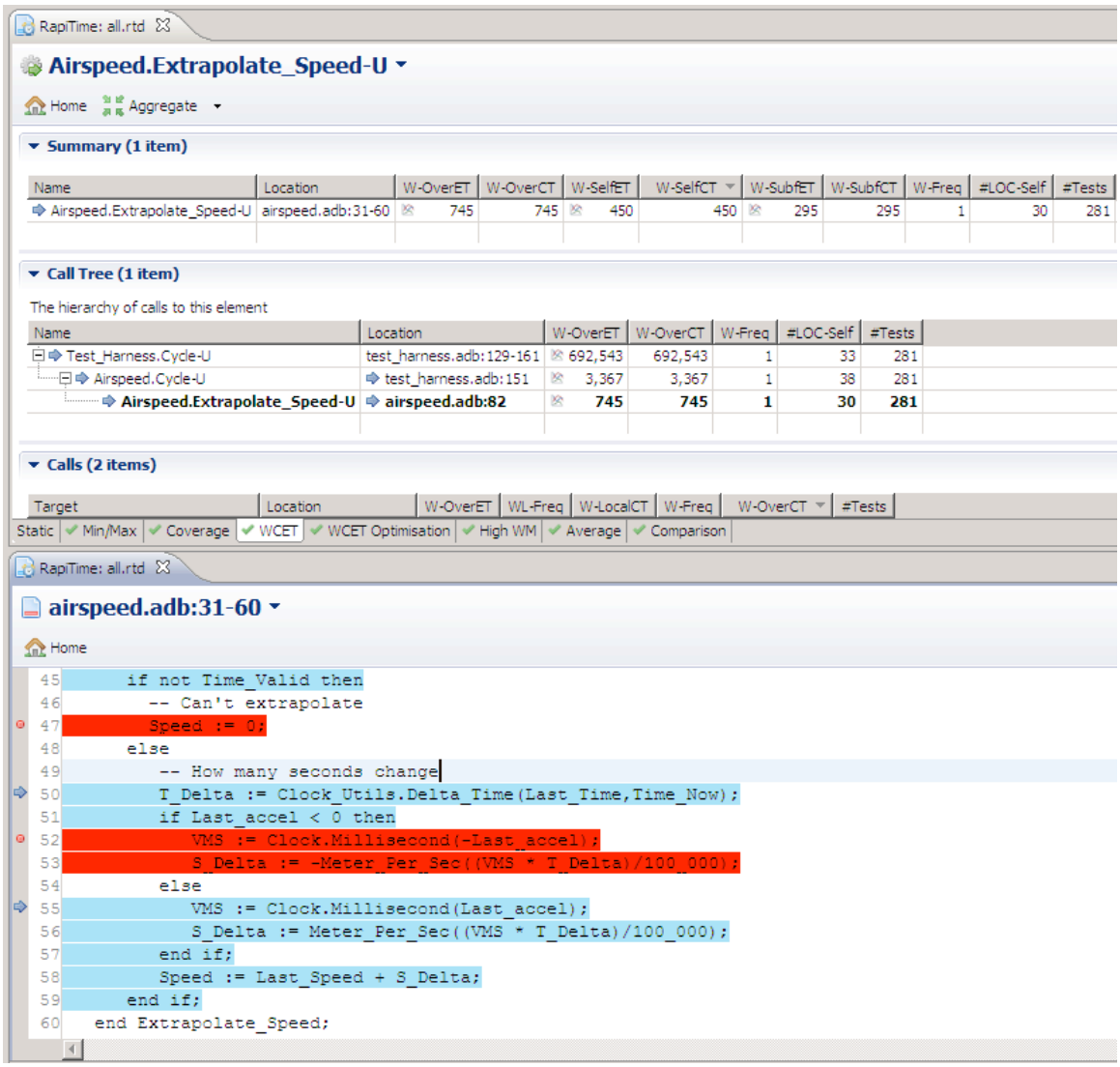

Figure 5.12: A rough approximation of back-annotation can be found in the commercial analysis tool RapiTime. It is able to color-code the worst-case hot spots of a program, as shown on this screenshot.

the actual numeric worst-case times, however, and the color-coding is displayed as part of a static, read-only report. It is not interactive.

5.4 The Road to True Interactive WCET Analysis

Clearly, interactive WCET analysis is far from reaching its full potential. Existing approaches are too limited; either they do not support true back-annotation, or they are not fast enough to be interactive. Even the perpetually increasing power of the modern workstation is unlikely to mitigate the fundamental performance problems in WCET analysis. Highly multicore processors, for instance, are poised to become the norm, and a parallelized IPET algorithm executing on one would certainly reduce analysis time. Yet the advent of faster hardware does not change the fact that IPET is an NP-hard problem, and analysis algorithms will only grow in complexity, as will the real-time programs that they target. These complexity increases will very likely nullify any future speed improvements in workstation hardware.

The simplifying assumptions presented in Chapter 3 offer a way out. By placing a few pragmatic restrictions on the underlying hardware, as listed on page 36, WCET analysis becomes more tractable and therefore faster. The following chapter looks at how to construct an interactive analysis tool—including full support for back-annotation—by building upon these assumptions.

Chapter 6

Clepsydra: An Interactive WCET Analysis Tool

Chapter Summary

Context Building on this idea of interactive analysis, the next step is to explore and refine techniques for increasing analysis speed. Other important goals include shielding the developer from the low-level assembly code used for the analysis and integrating the analysis results into a traditional development environment.

Prior Work One way of providing analysis feedback is to annotate each source code construct (`if` statements, method invocations, etc.) with its worst-case time. Known as back-annotation, this idea is not entirely new and has been implemented to some degree in earlier research. A WCET analysis tool from Kirner, for example, includes some basic WCET information alongside a text dump of the program's source code, but this was largely a side-effect of combining the tool with optimizing compilers. In the commercial space, the only product that comes close to back-annotation is RapiTime, which is able to highlight lines of source code that correspond to worst-case hot spots in execution time.

Problems All prior efforts in back-annotation exhibit two key drawbacks. First, they are not interactive. Their reliance on IPET means that they cannot generate WCET analysis results fast enough to provide continuous feedback. The analysis must be put off until a later phase of the development cycle. Second, they visualize the results only in terms of a static, read-only report. There is no attempt to integrate the results directly into a development environment.

New Claims There are two ways to solve these problems and achieve true interactive analysis: Make IPET faster, or make the tree technique more accurate. Given that IPET is inherently slow, this work claims that the only solution is to maintain the high performance of tree methods while improving their accuracy. A new algorithm is presented that shows how this feat can be accomplished for the special case of a dual-method cache. The intent is to prove that the high accuracy of IPET can be achieved using the faster tree-based algorithms. (The larger problems of garbage collection, polymorphism, and automatic loop bound detection are left for future work.)

Results Empirical measurements comparing the tree algorithm presented in this chapter with an equivalent implementation of IPET show that the new approach attains precisely the same accuracy without sacrificing superior speed. IPET suffers from exponential running time as program complexity grows, while the tree algorithm's growth is nearly constant, requiring just milliseconds even at high complexity. The algorithm is then applied to the problem of interactive back-annotation. An editor plug-in is presented to demonstrate continuous feedback of WCET analysis results while also shielding the developer from the underlying machine code.

Papers on the theory of WCET analysis are easy to find, but implementations are far less common. Tools for WCET analysis in the Java domain are even less common, virtually non-existent. Without working implementations to build upon, finding new avenues of research and testing new theories are exceedingly difficult. The lack of implementations also prevents real-time developers from gaining the productivity advantages offered by Java.

At the same time, research in the theory of WCET analysis tends to converge on low-level problems such as compiler optimization [210], but these advancements typically have not propagated into high-level solutions for the real-time software industry. In a 2003 survey of WCET tool users [99], half of the respondents said that such tools are lacking:

1. A very rough first estimate

2. Back-annotation of results into the source code

Given these responses, the WCET research community has failed to address certain needs of real-time practitioners. The vast majority of techniques are concerned only with obtaining a tight and accurate bound. Likewise, the larger field of real-time systems research has fixated on schedulability analysis for the past four decades, but reports of its successful application in industrial settings are quite rare. In fact, it is easier to find reports of *unsuccessful* attempts at moving real-time systems theory outside of the academic environment [211]. There has been little or no attention on other features that industry desires, such as finding a rough but adequate WCET estimate very quickly, then back-annotating those results into the source code.

To address these problems, the Volta tool suite introduced in Section 2.4 includes

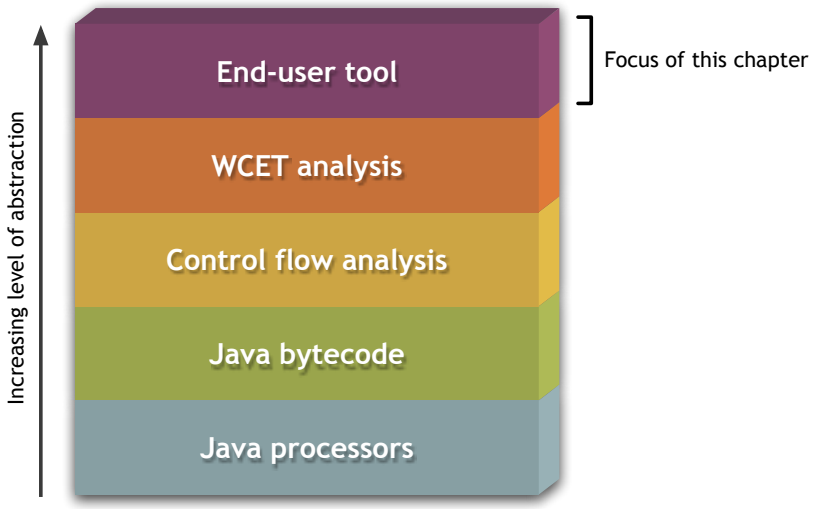

Figure 6.1: WCET tools for interactive analysis are built atop the multiple layers of abstraction shown in this diagram.

a worst-case execution time analyzer called Clepsydra.[1] It implements all of the theory described in Section 5.1 while adding some innovations to improve the speed of WCET analysis. With these improvements, Clepsydra shows that the interactive analysis proposed in Section 5.3 is an achievable goal. It also demonstrates the first implementation of automatic, round-trip back-annotation in a WCET analysis tool.

The sections that follow discuss the design and functionality of Clepsydra and, as illustrated in Figure 6.1, how it builds upon the foundations of previous chapters.

6.1 An Overview of Clepsydra

The design of Clepsydra revolves around three basic goals:

Shield the developer from assembly code One of the ideals of interactive analysis is to support high-level languages (see page 31). In keeping with this goal, Clepsydra hides low-level assembly code and provides WCET analysis results

[1] A clepsydra is a clock that measures time by the escape of water.

entirely within the context of Java. To do so, it relies on Cascade to generate annotated control flow data structures of the type described in Section 4.2. Clepsydra then calculates the WCET of each source code element of this structure. Finally, it maps the WCET information back to the original source code listing to produce back-annotations. Assembly code is never exposed to the user during this process.

Provide interactive WCET analysis Most of the existing analysis tools described in Section 5.3.2 are monolithic, stand-alone programs. In contrast, Clepsydra is designed to link seamlessly into a software developer's natural environment: the source code editor. It is able to place back-annotations directly into the editor's window, as shown in Section 6.3, so that WCET information is available as the program is written. In order to support this kind of interactive analysis, Clepsydra provides a speedy tree-based algorithm enhanced with certain abilities for instruction cache analysis. It is as accurate as prior IPET-based implementations [135] but is much faster. (Refer to Section 6.5.1 for performance benchmarks.)

Expose a modular and extensible structure Like the rest of the Volta tools, Clepsydra's architecture is designed for integration and extensibility [212]. It can supply back-annotations to virtually any development environment that supports add-on components. It is also distributed under an open-source license so that any portion of the code can be enhanced or simply exported to an outside project, facilitating future research. Clepsydra also achieves flexibility through a modular internal architecture. As discussed in Section 5.1.3, WCET analysis techniques offer different strengths and weaknesses: Some run fast but may not find a tight bound; others can be made tighter but require exponential running time. No single approach is ideal, and for this reason Clepsydra makes

analysis techniques pluggable via the Strategy pattern [213]. Users can swap implementations cleanly without having to understand Clepsydra's internal workings. This flexibility is particularly important with respect to interactivity. Fast tree-based analysis is used by default for interactive back-annotation, but if the developer suspects the results are too pessimistic, Clepsydra can switch to a slower technique for tighter WCET estimation.

Figure 6.2 provides an overview of how Clepsydra combines all of these goals into a coherent analysis process. The analysis begins when the developer supplies a Java source file, which is immediately fed into a custom Java compiler that supports WCET annotations (see Section B.5). Cascade then reconstructs the bytecode output of this compiler into an analysis-friendly control flow graph or tree. Finally, Clepsydra performs the actual analysis and produces worst-case timing values for every statement and compound structure in the decompiled source code. The dotted line in the figure represents this back-annotation from Clepsydra's output to the source code.

From all this effort, Clepsydra obtains an important and novel result: Back-annotation becomes *interactive*. The process shown in Figure 6.2 may seem elaborate, but the measurements of Section 6.5.1 reveal that this round trip analysis completes in a fraction of a second on modern workstations (depending on program size). Thus, timing data can be integrated into the design of a hard real-time system *as it is constructed*. The effect of code changes on worst-case time can be seen almost instantly. No WCET tool has ever supported this kind of interactive, closed-loop back-annotation.

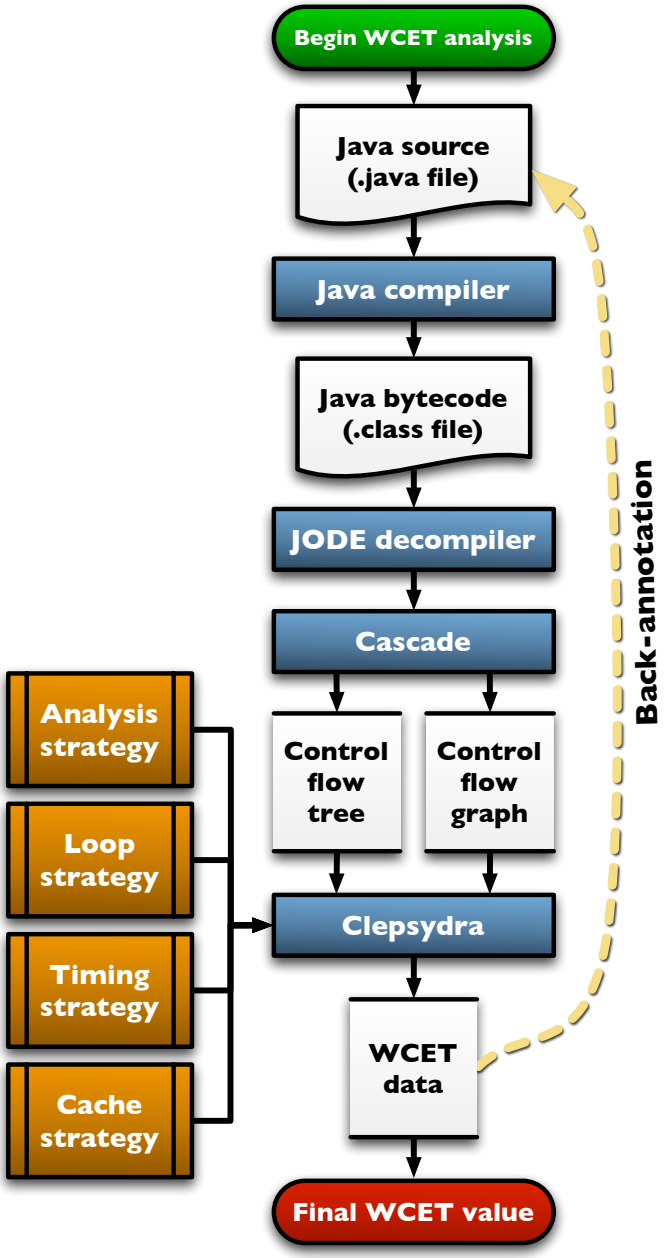

Figure 6.2: The Cascade and Clepsydra tools work together to provide interactive WCET analysis with back-annotation, represented in the figure by a dotted line.

6.2 Assumptions and Limitations

One likely reason that interactive back-annotation had not previously been achieved in WCET analysis tools is the prevalence of C. Its low-level nature, combined with

optimizing compilers and diverse processors, makes back-annotation so difficult to implement properly that most tools have simply avoided it altogether.

Although capable languages other than C have long been available, developers of real-time systems are a notoriously conservative bunch. There is a tendency to stick with traditional methods and tools rather than move to unfamiliar ones, even when the new solutions are more productive than the old [214]. Overriding the legitimate cautiousness of this risk-averse group requires a language with compelling advantages and a clear migration path from C.

As noted in Section 3.2, Java fits this mold. It is becoming a legitimate candidate for real-time systems, and it offers a syntax familiar to C programmers. At the same time, the rigor of its specification—definite assignment, checked exceptions, and other safety features—prevent mistakes that C would allow.

In addition, Java bytecode is much closer to the source language than the machine instructions found in C executables. It allows a near-perfect reconstruction of the original source, assuming debugging symbols are available. Back-annotation is then a much simpler process because the bytecode under analysis can be mapped to its equivalent source code.

This mapping is not feasible for any arbitrary platform, however. Java is more dynamic in nature than C, making it difficult to analyze for worst-case execution time. The Java virtual machine simply offers too many sources of unpredictability. As outlined in Section 3.4, a promising solution is to deploy Java software to a Java-specific processor. Such processors eliminate the virtual machine and offer enough determinism to make Java suitable for hard real-time systems.

Based on these observations, a complete list of Clepsydra's assumptions can be assembled. It combines the requirements of interactive analysis (page 36) with conventional

timing analysis restrictions (page 107). Together, these assumptions provide enough simplification to allow Clepsydra to support interactivity with back-annotation. They are, in summary:

- The program under analysis is written entirely in Java.
- The program will execute on a Java chip, such as the aJile or JOP.
- All loop statements have bounded iterations that are supplied to Clepsydra, either as source code annotations or some other means.
- Recursive function calls are prohibited.
- Dynamic memory heap allocations, as well as garbage collection to free them, are prohibited.
- Dynamic dispatch is prohibited. Therefore, object-oriented programming, which relies on dynamic dispatch of virtual methods, is also prohibited.
- Exception handling is prohibited.

These restrictions may seem Draconian and contradictory. For example, Clepsydra is unable to cope with two of Java's most celebrated features: object orientation and garbage collection. Even without these features, however, many of the advantages of Java over C, such as the high-level analysis described in Section 3.2, remain. And of course, future research could eventually loosen these restrictions. Experiments in hard real-time garbage collection, for instance, are ongoing [81, 75]. Approaches for WCET analysis of dynamic dispatch are also making progress [215, 66]. For now, these restrictions offer a reasonable compromise for the sake of simplicity, a necessary trade-off considering the complexity of interactive WCET analysis.

6.3 An Editor Plugin for Back-annotation

With these simplifying assumptions in place, interactive back-annotation becomes feasible. Clepsydra is able to produce a mapping between source code line numbers and their corresponding WCET values. An interesting problem, then, is how best to expose this ability to real-time software developers.

One way of providing back-annotation is by integrating with programming tools that are already in use. Programs like Eclipse [162], NetBeans [163], and Microsoft Visual Studio provide various mechanisms, known as *plugins*, for customizing the appearance and behavior of the editing environment. Plugins can therefore be designed to insert WCET back-annotations directly into the editor window. As the developer edits the source code, the plugin can run a WCET analysis in the background and automatically update the back-annotations to match the changes.

As proof of this concept, the suite of Volta tools includes a prototype editor plugin. This particular prototype is based on the programmer's editor jEdit [161], but the approach can be adapted for any tool that allows the contents of its editor window to be decorated by a plugin.

Figure 6.3 shows a screenshot of the jEdit plugin in action. The red text attached to each statement, as well as the method as a whole, indicates how much time it consumes in the worst case. (The times shown in this example are based on the 100 MHz JOP.) Note that these back-annotations were inserted automatically by the plugin following a WCET analysis of the code. The plugin re-runs the analysis and generates new back-annotations whenever the developer saves changes to disk.

Looking more closely at Figure 6.3 reveals one of the key benefits of interactive back-annotation. The screenshot shows two separate implementations of a hash function:

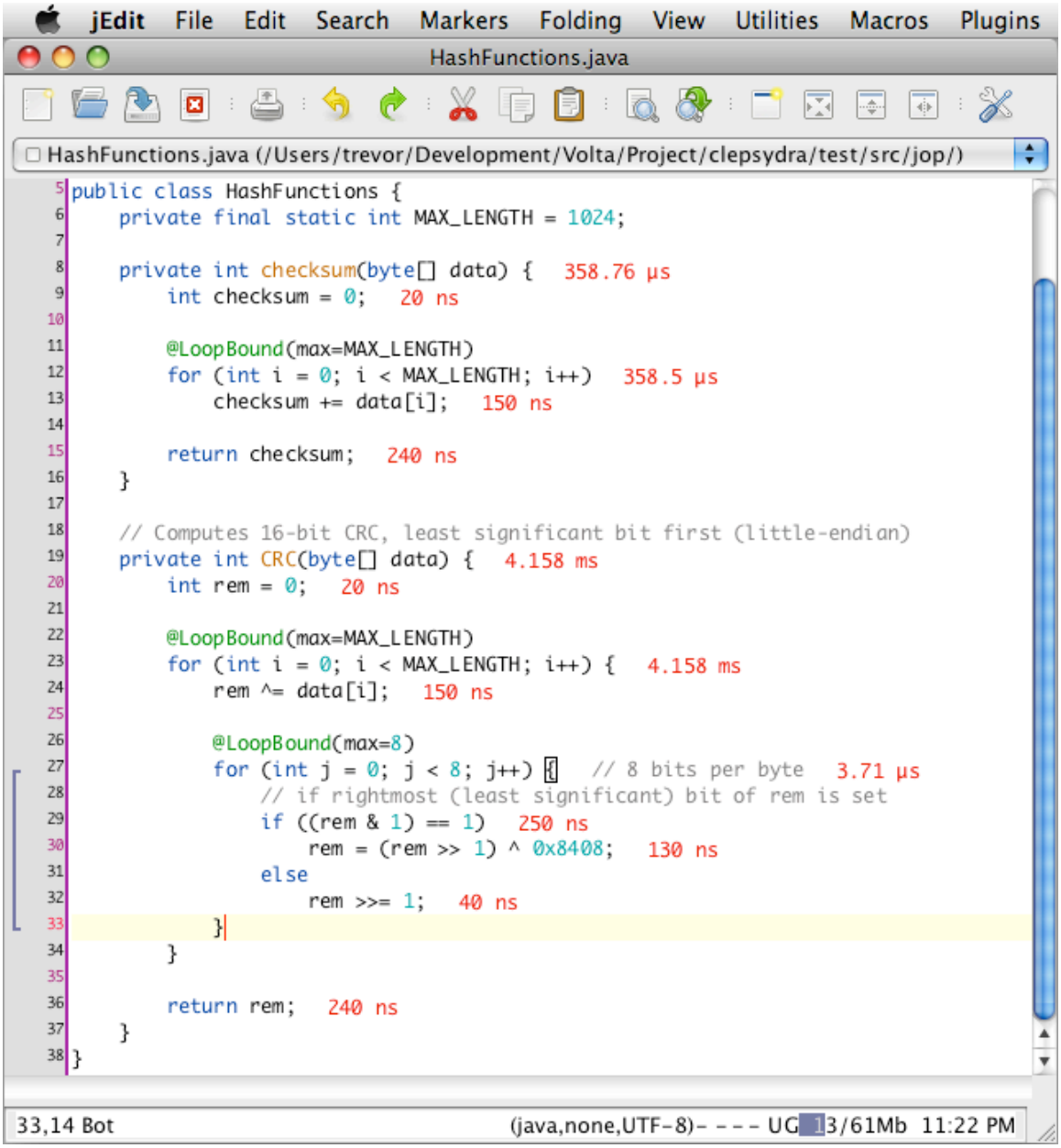

Figure 6.3: This screenshot shows a Clepsydra plugin for the jEdit programmer's editor. The red text, which was inserted automatically by the plugin, indicates the WCET of the corresponding line.

one computes a simple checksum; the other computes a more robust but also more computationally intensive cyclic redundancy check (CRC). Without running the code or even switching out of the programming environment, the developer knows that the CRC method has a much larger worst-case execution time: 4.2 milliseconds versus 360

microseconds, a ratio of nearly 12. The developer can use this information directly when designing the real-time system. If, for example, a task requires a worst-case response time of four milliseconds, the developer will know instantly that the CRC function cannot be incorporated safely into the task.

The experimental process of creating this plugin helped crystallize the benefits of interactive back-annotation, but it also unmasked some potential pitfalls in making the approach practical. One issue is the problem of compiler optimization. A compiler may, in some circumstances, produce bytecode whose control flow differs slightly from what is expected. Consider a snippet such as this:

```
if (b)
    return 0;
else
    val = 10;
```

Many Java compilers would remove the else construct—suppressing the goto instruction that would otherwise be generated for the if statement—because it is unnecessary. These sorts of optimizing modifications to the control flow could disrupt Clepsydra's back-annotation algorithm. One possible solution is to modify the compiler to disable all such optimizations. However, given that optimization in Java compilers is usually very limited, a less extreme tactic would simply compare the decompiled code to the original code. In the rare cases when a discrepancy is found, a warning could be issued that would prompt the developer to rewrite the code in a manner that is compatible with Clepsydra's back-annotation.

Another problematic issue is dealing with limitations of the line number table. When debugging symbols are enabled, Java provides a table that maps every bytecode instruction to its corresponding source code statement. This map is indispensable for

back-annotation because it indicates on which line of the source code the annotation should appear. Unfortunately, Java's line number table only extends to the method *implementation* and provides no mapping for the method *declaration*. Clepsydra works around this problem by requiring the source code file as input, which it scans to determine the method declaration line. A more robust workaround would modify the Java compiler to generate an additional line number mapping for method declarations.

6.4 The Modular Components of Clepsydra

The previous sections of this chapter painted an overview of Clepsydra's primary features and limitations. This section peels away these outer layers for a look at how Clepsydra's internal architecture enables flexible WCET analysis. It separates the analysis process into four modular components: an analysis strategy, a loop bound strategy, a timing strategy, and a cache strategy.

6.4.1 Analysis Strategy

The first example of this flexibility can be found in the way Clepsydra deals with high-level WCET analysis. Too often in research literature, authors seem to focus on one single analysis approach and discount the others. A better tactic is to combine them in some way. For instance, a tree-based approach could be used for very fast WCET estimation to speed the development cycle. Later, if the developer decides that the estimation is too pessimistic, more time could be expended on some other approach, such as IPET, for tightening the bound.

No single approach is ideal, and for this reason Clepsydra makes analysis techniques pluggable via the Strategy pattern. Developers can switch between them with relative

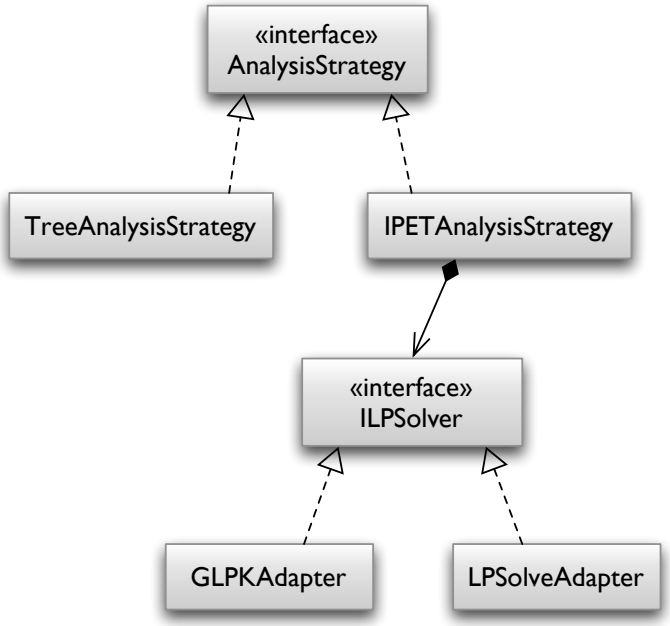

Figure 6.4: Clepsydra supports interchangeable analysis algorithms by means of its AnalysisStrategy interface, an implementation of the Strategy pattern. Tree-based and IPET algorithms are provided by default; more can be added seamlessly. In addition, the IPET algorithm relies on an Adapter pattern interface for pluggable ILP solvers.

ease, allowing the same Clepsydra framework to be used as existing techniques are refined and new ones are created.

By default, Clepsydra provides ready-to-use implementations of both the tree and IPET techniques. It implements the tree technique in approximately one hundred lines of pure Java code, but its IPET implementation is significantly more complex. Part of this complexity is due to the graph-based nature of the technique; in addition, Clepsydra incorporates external C-based libraries to solve the requisite integer linear programming (ILP) problem, due to the lack of open-source solvers in Java.

Clepsydra relies on the Adapter pattern to provide a common interface to these libraries (see Figure 6.4), allowing the developer to switch between them as necessary. For example, a commercial ILP solver, which is usually faster than an open-source equivalent, can be integrated into Clepsydra by writing a thin adapter for it.

Due to the fundamental problems raised in Section 5.1.3, Clepsydra does not include out-of-the-box support for the explicit path enumeration algorithm. However, it can be added without any modifications to Clepsydra's core architecture. The algorithm would only need to implement the AnalysisStrategy interface.

6.4.2 Loop Bound Strategy

Another example of Clepsydra's flexibility lies in its handling of loop bounds. Loop bounds are a prerequisite for WCET analysis, but there is no single "right way" to determine the maximum iterations of a loop.

Some approaches depend on source code annotations inserted manually by the author of the code [34, 216, 217]. They require extra effort from the developer and are error-prone [35], but they make WCET analysis faster and simpler. Other approaches depend on deep data flow analysis to find loop bounds automatically (if possible). They prevent errors due to inserting a wrong annotation, but they are quite complex, difficult to implement, and may substantially reduce the speed of analysis to a point where interactivity becomes impossible.

Recognizing that no single algorithm is best, Clepsydra factors out loop bound determination via the Strategy pattern. Developers can plug in a default strategy supplied by Clepsydra, or they can implement their own without having to understand the details of Clepsydra's design. Strategies can also be swapped at runtime to adjust to the needs of the user.

The default strategy currently provided in Clepsydra is annotation-based. It assumes that the developer has annotated every loop construct with its maximum number of iterations. An example of such annotations can be seen in Figure 6.3. The developer

has indicated that the loop on line 27, for instance, iterates no more than eight times.

Note that the annotations in Figure 6.3 are not, in fact, legal. Although Java supports the annotation *syntax* of lines 11, 22, and 26, the *location* of these annotations will cause compiler errors. Currently, Java allows annotations only on declarations, not on statements, though prior work has discussed the importance of removing these restrictions on annotation placement, especially for WCET analysis [218]. Benefits include "for free" syntax checking, type safety, and tool support. Such features have not been available in the annotation mechanisms of traditional analysis tools.

Therefore, Clepsydra works in conjunction with a version of Sun's standard Java compiler that has been modified to allow annotations on statements. It parses the annotations for syntactical correctness and creates a representation of them in the Java class file output. The default loop bound strategy in Clepsydra then obtains the loop bound values by loading the class annotations using BCEL [136], a third-party tool for accessing Java class files. (Refer to Appendix B for a broader discussion of WCET annotations in Java.)

This process effectively solves the Turing halting problem for WCET analysis, but it is a source of human error. There is no way to validate whether the annotations are correct; Clepsydra must simply trust them. As the code evolves, the developer must remember to evolve the annotations along with it. If the two ever become out of sync, the WCET estimation could be extremely pessimistic or, even worse, extremely unsafe.

A more reliable approach is to attempt some form of data flow analysis to find loop bounds automatically. Although not every loop bound can be determined this way, many common loop patterns are amenable to analysis, such as the simple counting loops of Figure 6.3. Recent work has focused on *abstract interpretation* to discover

the bounds of such loops without the need for source code annotations [219, 178, 220, 179]. Other tactics involve annotations only to describe the range of input variables (e.g., i1 <= 0 && i1 < str.length) from which loop bounds can then be derived [221].

In either case, the topic of automatic loop bound analysis is a subject of ongoing research. As it matures, a particular technique can be encapsulated as a loop bound strategy and integrated into Clepsydra as a replacement for, or perhaps a complement to, the existing annotation-based strategy. For example, Clepsydra could use an abstract interpretation strategy for simple loops and a source code annotation strategy for complex ones.

6.4.3 Timing Strategy

After the control flow has been analyzed and the loop bounds determined, the final WCET values must be realized for a particular Java processor. This involves obtaining cycle counts for individual bytecode instructions. Clepsydra relies on the Strategy pattern to make these counts pluggable, depending on the processor target.

Clepsydra provides a JOP strategy by default. As JOP's architecture evolves, its timing may change, but Clepsydra requires only an updated strategy to support the new design. Likewise, entirely different processors, such as the aJile or Cjip, can be added with minimal impact on Clepsydra by supplying the appropriate strategy. It must implement a simple Java interface, as illustrated in Figure 6.5.

Originally, the Strategy pattern for low-level timing analysis was not part of Clepsydra's design. The initial goal was to define an XML schema that would model processor timing data. In practice, however, encapsulating such data in the static tree structure of an XML file is troublesome due to state dependencies. For instance,

```
public int getCycles(short opcode, boolean cacheHit)
{
    int methodLoadTime = getMethodLoadTime(cacheHit);

    switch (opcode) {
        case SIPUSH:
            return 3;

        case LDC:
            return 7 + readWaitStates;

        case LDC2_W:
            cycles = 17;
            if (readWaitStates > 2)
                cycles += readWaitStates - 2;
            if (readWaitStates > 1)
                cycles += readWaitStates - 1;
            return cycles;
        .
        .
        .
```

Figure 6.5: Clepsydra relies on the Strategy pattern to make processor timing definitions easily swappable. This listing shows a portion of the JOP timing strategy.

as shown in Figure 6.5, the cycle timing for a double-word constant push instruction (ldc2_w) varies depending on the number of wait states in the target processor's memory subsystem. Modeling this logic in Java rather than XML is far simpler, and therefore the Strategy approach is preferable for WCET analysis tools like Clepsydra.

6.4.4 Cache Strategy

The fourth and final modular component in Clepsydra is a method cache strategy. Without it, Clepsydra would only be able to analyze single methods in isolation. Analysis of whole programs across method invocations requires a model of the target processor's instruction cache. (Refer to Section 5.1.2 for a description of method caching.)

An IPET Strategy for Cache Analysis

Before such a model can be applied to Clepsydra, the underlying control flow graph must first be modified to represent the effects of the cache. Figure 6.6 shows an example of these modifications for the code in Figure 4.6. The extra cache miss blocks in the graph (numbers 8, 9, and 13) represent the alternate path in the control flow in the event of a method cache miss. These blocks are added to return instructions as well as method invocations because both can cause cache misses. The execution time, or "cost," of control flow passing through the blocks is equal to the miss penalty alone (i.e., the number of extra cycles needed to load the method), not the total time for invocation or return.

With the control flow graph adjusted for method cache support, the next step is to add the appropriate constraints in the IPET analysis to account for the method cache. For the single method configuration, no additional constraints are necessary because the miss path must always be taken, and this happens automatically as a result of finding the worst case path.

For the dual method configuration, Clepsydra's default cache strategy supports only the simple analysis described in Section 5.1.2. (It is the same technique first described by Schoeberl and Pedersen [135].) When the strategy encounters a method invocation, it first checks whether the invocation occurs within a loop and whether it is the only one of its type in the entire loop body. (The same method can be invoked multiple times in a loop body without causing cache misses.) If both conditions are true, the invocation is a miss on the first iteration but a guaranteed hit on all subsequent iterations. Clepsydra uses this fact to add special method cache constraints in the ILP formulation. In the case of Figure 6.6, for example, it would add the following constraints for the invocation in Block 10:

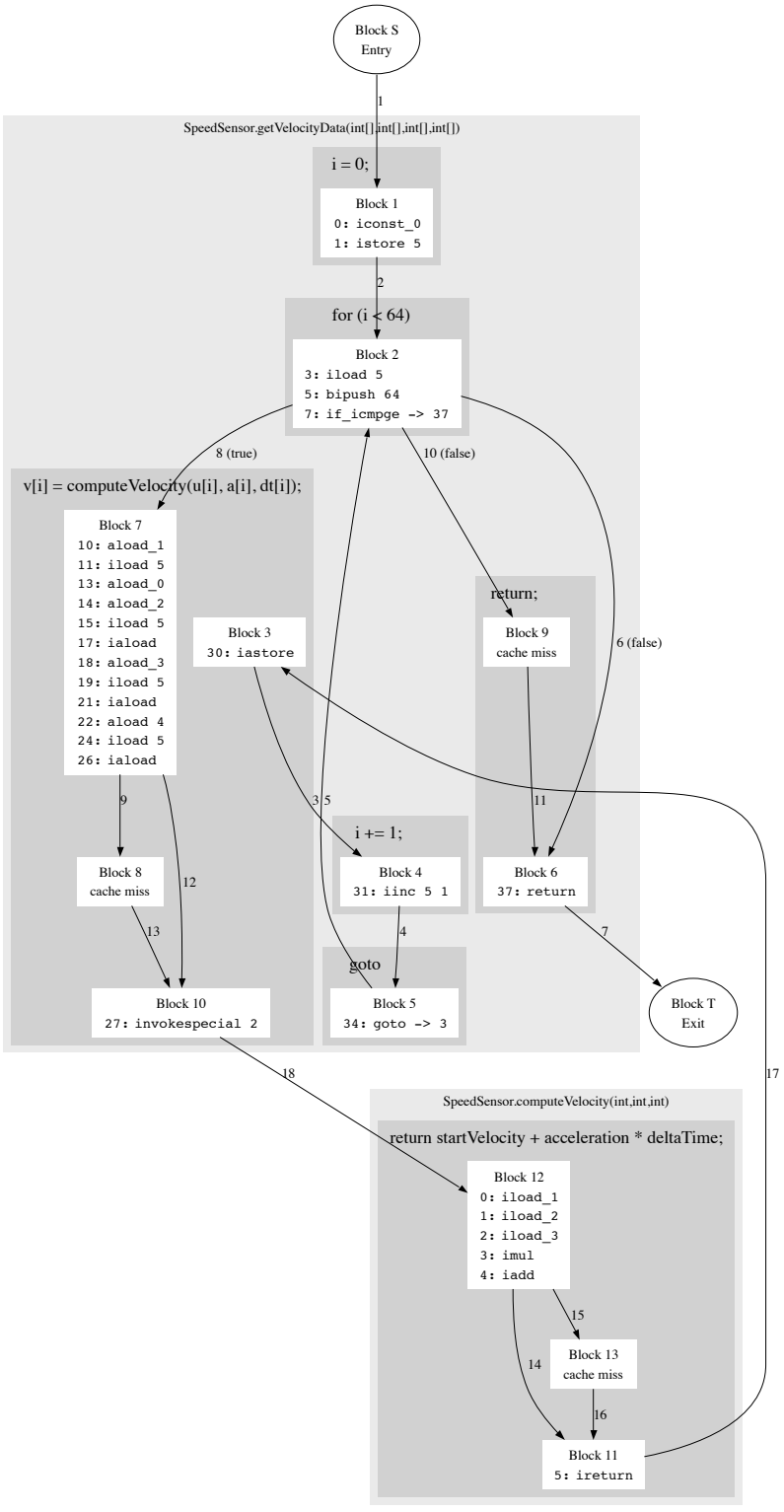

Figure 6.6: To support method cache analysis, control flow graphs must be augmented with cache miss blocks, as shown in this example of the code from Figure 4.6. Structurally, it is identical to the graph in Figure 4.8 except for blocks 8, 9, and 13.

```
edge9  = edge2
edge12 = (n-1) edge2
```

where the constant n is the loop bound (64 in this example).

The first constraint states that the flow through Edge 9 is the same as the flow through Edge 2, meaning that Block 8 will be followed exactly once (that is, a single cache miss) because Edge 2 will be followed exactly once. The second constraint states that the flow through Edge 12 is 63 times as large as the flow through Edge 2. In other words, Block 8 will be bypassed 63 times because there will be 63 cache hits.

For return instructions, Clepsydra considers only leaves in the call tree. (A leaf is a method that invokes no methods.) Any return instruction in a leaf is always a guaranteed hit for the dual-block cache, regardless of loop structure. Cascade's API makes testing for this condition easy; it requires only a simple expression:

```
getTree().getMethodInvocations().isEmpty()
```

This expression obtains the control flow tree belonging to the return instruction, then checks whether the set of method invocations in that tree is empty.

Figure 6.7 shows the result of a complete analysis that puts all of these constraints together. The listing is an ILP formulation by Clepsydra of the control flow in Figure 6.6. Note that the basic block constraints for the cache misses in this particular example evaluate to zero. The reason is that the miss penalty for small methods, such as computeVelocity, is so low that it is masked by the time required for the invocation itself. Thus, the constraint simply disappears, resulting in an effective cache hit for invocations and returns in all situations. (Larger methods would result in non-zero

```
/* Objective function */
max: +block1 +block2 +block3 +block4 +block5 +block6 +block7 +block8
     +block9 +block10 +block11 +block12 +block13;

/* Edge constraints */
+edge1 = 1;
+edge7 = 1;
+edge1 -edge2 = 0;
+edge2 +edge5 -edge6 -edge8 -edge10 = 0;
-edge3 +edge17 = 0;
+edge3 -edge4 = 0;
+edge4 -edge5 = 0;
+edge6 -edge7 +edge11 = 0;
+edge8 -edge9 -edge12 = 0;
+edge9 -edge13 = 0;
+edge10 -edge11 = 0;
+edge12 +edge13 -edge18 = 0;
+edge14 +edge16 -edge17 = 0;
-edge14 -edge15 +edge18 = 0;
+edge15 -edge16 = 0;

/* Loop constraints */
-64 edge2 +edge8 = 0;

/* Cache constraints */
-edge2 +edge9 = 0;
-63 edge2 +edge12 = 0;

/* Basic block constraints */
+3 edge1 -block1 = 0;
+8 edge2 +8 edge5 -block2 = 0;
+13 edge17 -block3 = 0;
+8 edge3 -block4 = 0;
+4 edge4 -block5 = 0;
+21 edge6 +21 edge11 -block6 = 0;
+44 edge8 -block7 = 0;
-block8 = 0;
-block9 = 0;
+75 edge12 +75 edge13 -block10 = 0;
+23 edge14 +23 edge16 -block11 = 0;
+39 edge18 -block12 = 0;
-block13 = 0;
```

Figure 6.7: Clepsydra produces this ILP formulation, shown in lp_solve format, after applying the IPET analysis strategy and a dual method cache strategy to Figure 4.6.

cache miss penalties.)

While this approach produces the expected WCET value, it is based on IPET, which,

as discussed in Section 5.1.3, is computationally expensive. Even for relatively small programs, computing WCET in the presence of method invocations can take ten seconds or more; larger programs may require several minutes even on a fast workstation. For interactive analysis of real-time software, this delay is unacceptable.

A Two-Pass Variation on the Tree Strategy

To solve this problem, Clepsydra includes a tree-based algorithm capable of analyzing the dual method cache. It offers accuracy that is identical to the IPET algorithm but executes in a fraction of the time. (An empirical performance analysis of the algorithm is provided in Section 6.5.1.)

The algorithm is a two-pass variation of the standard tree analysis approach (i.e., recursive descent). In the first pass, it treats all method invocations as cache misses. It then executes a second pass through the control flow tree and examines only the method invocations. For invocations determined to be cache hits, it multiplies the number of hits by the miss penalty for that particular invocation. This penalty is actually the *gain time* because the first pass assumed only cache misses. Finally, the algorithm simply subtracts the gain time from the first pass total to arrive at the final WCET calculation. Figure 6.8 provides a pseudocode listing of this approach. A complete implementation can be found in the Volta distribution.

Motivation for the Two-Pass Variation

This two-pass variation of the traditional tree strategy is not intended to be a general-purpose replacement for IPET. Rather, it is an example of how the tree-based approach can be refined and extended so that it produces WCET estimates that are just as accurate as more recent techniques. Other approaches, such as IPET, are still

```
getWCET(node)

  if node is null return 0

  if node is IF_THEN_ELSE
    cycles = getExprWCET(node.conditional_expression) +
             max(getWCET(node.then_branch), getWCET(node.else_branch))

  else if node is LOOP
    expression_cycles = getExpressionWCET(node.conditional_expression)
    cycles = expression_cycles +
             (getWCET(node.loop_body) + expression_cycles)
                 * node.loop_bound

  else if node is STATEMENT
    cycles = getExpressionWCET(node.statement_expression)

  return cycles + getWCET(node.next)

getExpressionWCET(expression)
  return sum of CPU cycles of all instructions in the basic block
  (assumes all invocation instructions are cache misses)

getGainTime()
  gainTime = 0
  for each node in the control flow tree
    if the node contains a method invocation
      gainTime += number of cache hits of the invocation *
                  cache miss penalty
  return gainTime

getTotalWCET()
  return getWCET(root node) - getGainTime()
```

Figure 6.8: The standard recursive descent algorithm in tree-based analysis cannot account for method invocations because method cache hits are not constant across loop iterations. A two-pass variation of the algorithm, shown here in pseudocode, solves this problem.

valid and they produce the same quality of results. In fact, when speaking strictly of accuracy, there is no reason to use this two-pass variation of the tree-based approach. It will produce the same WCET as an equivalent IPET-based algorithm.

But accuracy is not the only criterion of an analysis technique. As discussed in Section 5.3, the speed of analysis is also significant, especially when considering inter-

active analysis. In this respect, IPET is inferior; it is simply too slow for an interactive analysis of a non-trivial program, as evidenced by the performance evaluation to be presented in Section 6.5.1. The past fifteen years of research into IPET have not been able to overcome the exponential running time inherent to the technique.

Based on these observations, then, there are two overall strategies for achieving fast, interactive WCET analysis:

- Make a slow but accurate technique fast
- Make a fast but inaccurate technique accurate

The two-pass variation presented in this section is an instance of the latter. The motivation is that the tree-based technique does not need to achieve an accuracy that is *better than* IPET. It only needs to be *as good as* IPET because, once it achieves parity in accuracy, it wins the speed contest. The linear running time of the tree approach makes it superior to IPET, at least with regard to interactive analysis.

In other words, IPET is not the end-all be-all of WCET analysis. Despite the fact that it has received the majority of attention in the last decade of WCET research, it does not render other techniques obsolete. The tree-based approach in particular is still highly relevant for interactive analysis. The challenge now is to improve this approach so that it brings the best of both worlds: speed and accuracy. The extension presented here, though non-trivial, demonstrates that such improvements in tree-based analysis are possible.

Future Work in Cache Analysis

Even with the support of this enhanced tree algorithm, there is still much room for improvement in Clepsydra's cache analysis. For example, one way to extend Clepsydra is with analysis of the *variable block method cache* [222]. It achieves better overall performance than the single and dual method caches, but it requires a more careful and complex analysis. Still, it is more WCET-friendly than a traditional direct mapped or associative cache because the analysis can be restricted to method invocation and return instructions.

Another avenue of future research is "what-if" analysis. Simply by swapping strategies, Clepsydra could determine which cache architecture yields the optimal WCET for a particular program. Configurable processors such as JOP could then be fitted with the appropriate architecture. Users could also experiment with uncommon variations—a triple method cache, for instance—and test their impact on WCET.

Of course, all of this method-based analysis applies only to the instruction cache. Support for a data cache would require major additions to Clepsydra in the form of data flow analysis. Future work should not neglect this aspect, however, given that data caches can greatly improve performance. For example, the data cache in the picoJava-II processor adds a 34% performance boost over the JOP, which has no data cache [223].

6.5 Evaluation

As a complement to the guided tour of Clepsydra's architecture in Section 6.4, this section evaluates the performance of its implementation for both speed and accuracy.

6.5.1 Performance Analysis

One of the key benefits of Clepsydra is interactive development of real-time software, such as the back-annotation feature described in Section 5.3.1. Given the importance of speed in interactive back-annotation, the running time of the various WCET analysis techniques becomes a prime consideration. Almost all prior work has focused on reducing pessimism in WCET analysis, but there has been little to no emphasis on reducing its execution time. The idea of near-instantaneous analysis offers the potential for a new breed of real-time software development tools, such as the editor plugin described in Section 6.3.

An obvious question to answer, then, is how fast the analysis techniques perform. Tree-based analysis has long been recognized as the fastest; its running time grows linearly with the size of the program. In contrast, the IPET technique is slower; it has in the worst case NP-hard complexity. Before discounting IPET, however, one should consider the remarkable speed of today's processors. Could the IPET approach be fast enough for interactivity on modern workstations, despite its complexity?

Performance measurements with Clepsydra show that this is not yet possible, as evidenced by Figure 6.9. The chart shows benchmarks for three analysis techniques: the tree technique, IPET using lp_solve, and IPET using GLPK [224]. (The seemingly redundant benchmarks for IPET are designed to rule out the possibility that a lackluster performance of IPET is merely the result of an inefficient ILP solver.) The benchmarks, which can be found in the Volta distribution, indicate how each technique performs as the cyclomatic complexity of a contrived input program increases. All tests were conducted on a 2 GHz Intel Core Duo machine running Java 1.6.0.

The results show that the tree-based technique is the clear winner. Its performance, while linear in growth, appears virtually constant, requiring only a few milliseconds

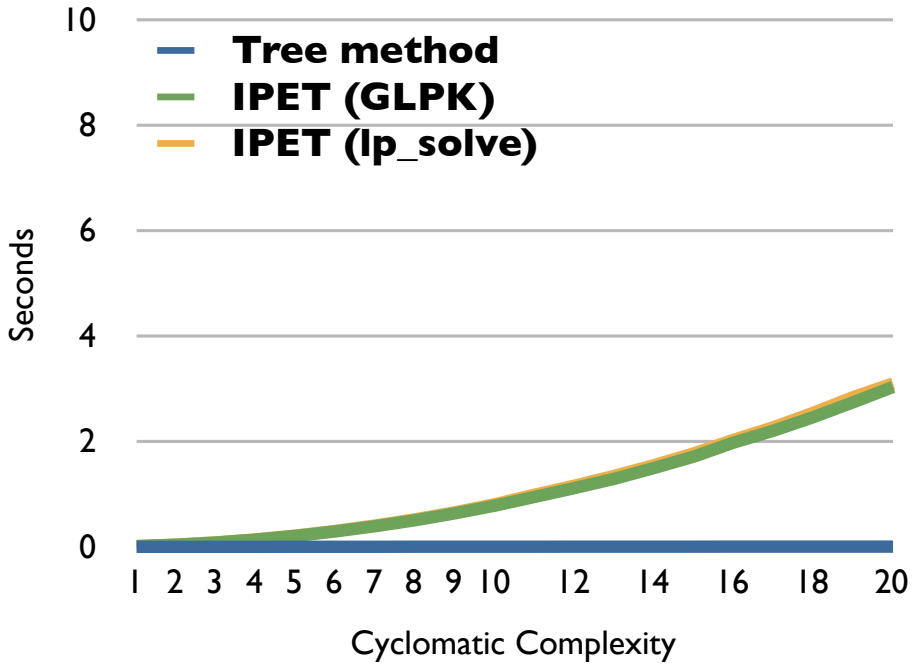

Figure 6.9: Performance testing of Clepsydra shows that the tree-based technique is the best choice for fast, interactive analysis.

even at high complexity. As expected, the IPET technique is much slower, growing exponentially with program size. For interactive back-annotation, tree-based analysis is unequivocally the best choice and deserves greater attention from the research community.

Figure 6.9 does not consider cache analysis, however; it only gives the analysis speed for a single method. In the presence of method invocations, there is a question of whether the tree-based technique's stellar performance can remain. A speed evaluation of all three algorithms was again conducted using the same hardware and software testbed, this time running a benchmark that invokes methods. The benchmark measures the performance of the algorithms under increasing complexity by increasing the height of the call stack. (For example, a method that calls a method that calls a method has a stack height of three.)

Figure 6.10 shows the results. Although the presence of methods slowed the per-

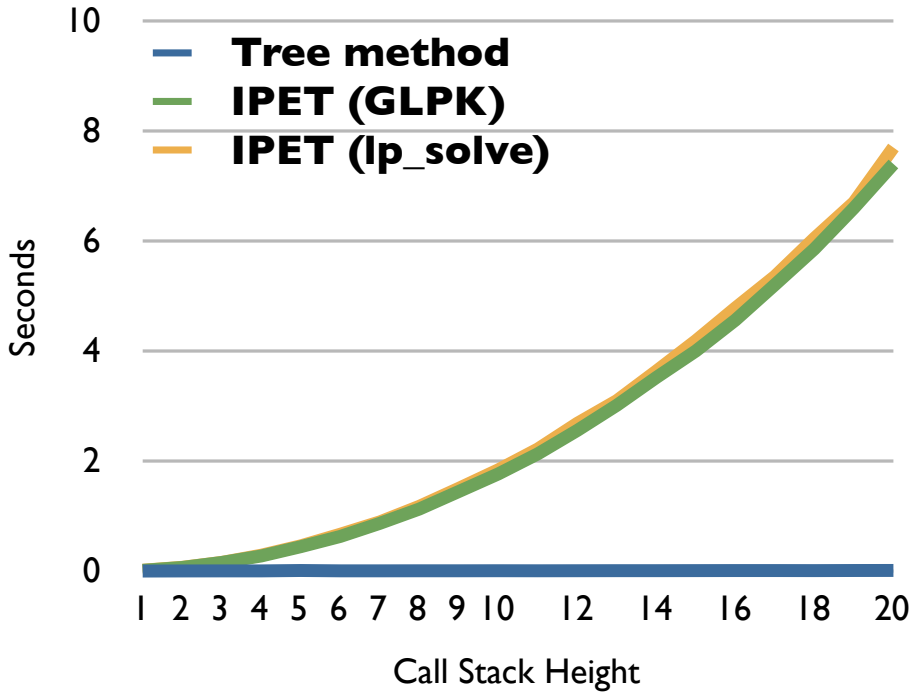

Figure 6.10: Even in the presence of method invocations with cache analysis, the tree technique handily outperforms the IPET alternative.

formance of all techniques, the trends were the same: The tree-based approach is extremely fast while the IPET approach is exponentially slower. The tree method is still the best choice for interactive analysis, even with the overhead of method cache support.

6.5.2 Accuracy Analysis

The primary focus of Clepsydra is to explore the potential of fast and flexible WCET analysis tools, but the accuracy of those tools is certainly a factor as well. For WCET, accuracy is normally defined in terms of *pessimism*—the amount by which the predicted and measured WCET values differ.

To evaluate the pessimism of Clepsydra, a set of fifteen WCET benchmark programs was created. It is based on a similar suite of benchmarks from the Mälardalen Real-

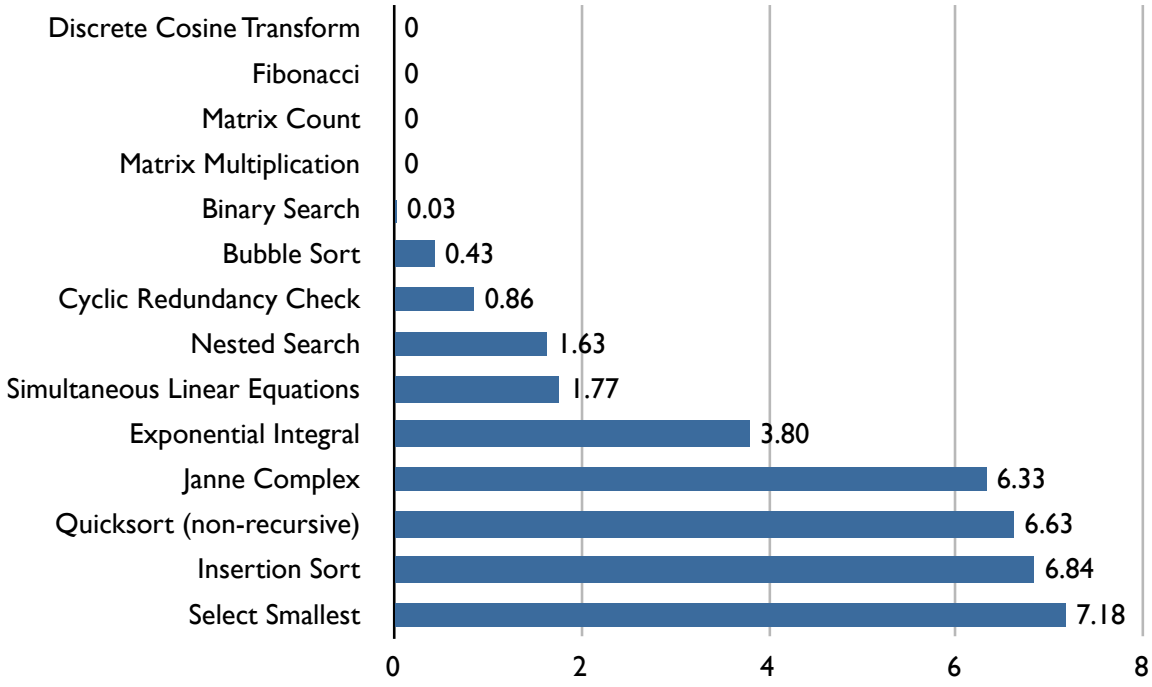

Figure 6.11: These numbers represent Clepsydra's pessimism ratio for a variety of WCET benchmarks. The ratio in each case compares the WCET predicted by Clepsydra to the true WCET measured on a 100 MHz JOP.

Time Research Center [225]. (Yet another benchmark suite, called PapaBench [226], also exists but was not selected for this work.) Provided in the Volta distribution under a public domain license, it is intended to serve as a *de facto* standard for evaluating any Java-based WCET analysis tool, not just Clepsydra.

Figure 6.11 shows the results of running these benchmarks on Clepsydra. For each benchmark, Clepsydra ran in its default configuration: a JOP processor as the target, the dual-method cache analysis strategy, manual annotations to detect loop bounds, and the two-pass tree analysis algorithm described in Figure 6.8. (Clepsydra's implementation of IPET is not shown because its results are identical in every case to the tree approach.)

The benchmarks vary widely. Discrete Cosine Transform, Fibonacci, Matrix Count, and Matrix Multiplication exhibit the ideal behavior of 0% pessimism because they

are simple loops.[2] The Select Smallest benchmark, a complex piece of code with many nested conditionals and loops, fared the worst at more than 700% pessimism. (That is, the time predicted by Clepsydra was about 7 times larger than the actual worst-case time measured on the JOP.) The Petri Net benchmark is missing from these results because it contains a single method 20 kilobytes in size, which overflows the method cache in the Altera Cyclone configuration of the JOP used in the tests.

The poor pessimism for some of these benchmarks is in part due to their nature. They are designed to stress typical weaknesses that often afflict WCET analyzers. The Janne Complex benchmark, for example, has an inner loop whose maximum number of iterations depends heavily on the outer loop's current iteration number. Structural analyzers that ignore data flow, such as Clepsydra, suffer greatly from this behavior.

Overall, the largest increase in pessimism was often observed to be a result of ineffective loop bound annotations. The Insertion Sort and Quicksort benchmarks in particular expose this problem; they contain inner loops whose bounds depend on the outer loop's state. The loop bound annotation mechanism currently supplied with Clepsydra can only specify constant bounds, leading to overly conservative estimates. Future work should focus on improved data flow analysis and loop bound detection.

Nevertheless, for the benchmarks that represent numerical computations typically found in real-time systems, such as the matrix and DCT benchmarks, the bounds are quite tight. The ideal pessimism ratio of 0% is clearly an attainable goal when using Java-based processors.

[2]Contrary to Figure 3.8, the pessimism ratio of the Bubble Sort benchmark is not even close to 0%. The discrepancy exists because the ratios presented here were computed automatically by Clepsydra, which currently relies on statically specified loop bounds. The results of Figure 3.8 were achieved by performing a manual WCET computation and assuming that a dynamic loop bound detector is in place and able to account for the dependency of the inner loop's bound on the outer loop's iteration.

6.5.3 Correctness Analysis

In the process of creating Clepsydra, a fundamental question arose: Static WCET analysis is safe in theory, but is it truly safe in practice? Even if the theory of a particular analysis technique is sound, the implementation of that technique may contain bugs that result in unsafe WCET estimation.

The potential for errors during analysis is a serious issue, given that hard real-time software is typically the foundation of mission- and even safety-critical systems. The analyzer may check the software for correctness, but what checks the analyzer for correctness? One might think to apply some formal verification technique to prove the analyzer's correctness, but this only shifts the problem, as the question then becomes how to check the analysis checker, *ad infinitum*.

This conundrum is, of course, nothing new. Verifying correctness has been a major concern for the aviation and space industries, for example, where a single failure can cause millions of dollars of damage and possibly the loss of human life. One solution that has been applied in these fields is known as *N-version programming* [227].

This software engineering practice involves N teams of developers working independently on N unique implementations of the same program. In the deployed system, all implementations run concurrently, and when their computations are complete, a separate program examines the results and decides which answer to accept. For instance, if two implementations of an algorithm agree on a result, but the third one differs, a voting procedure would reject it as incorrect. The approach results in a degree of tolerance to software defects—increasing with the size of N—because each version of the program checks the other N–1 versions.

For some researchers, however, N-version programming is a discredited idea. They

argue that the assumption of independent failures among the N versions is statistically invalid [228]. In other words, different programming teams can make similar mistakes. The approach may also be rejected because in many cases it is simply too expensive to be practical. An organization must create a second or third team for the extra implementations, which may double or triple the cost.

Yet in many cases, N-version programming is a sensible solution. Even researchers outside the realm of safety-critical and fault-tolerant systems have found it useful. A book preservation project, for example, has relied on N-version programming to digitize old texts using two separate character recognition programs [229]. If the programs disagree, the discrepancy is reported and a human takes over. The approach works in this case because the programs are commercial off-the-shelf products, not custom-built, so the total cost of creating the redundant implementations is amortized.

A similar approach can be applied to WCET analysis tools. For instance, if Clepsydra indicates that the WCET of a task is 10 milliseconds, while two similar tools say that the value should be 11 milliseconds, the user can reject Clepsydra's result. In addition to providing safer WCET analysis for users, this tactic would also help detect bugs in new analysis tools that are still undergoing development.

Even when these tools are developed in isolation, N-version programming can still provide benefits when experimenting with new longest path search algorithms. For instance, creating the two-pass variation of the tree algorithm (see 6.4.4) was aided by Clepsydra's ability to switch seamlessly between tree-based and IPET algorithms. Comparisons with a known good technique led to faster bug discovery and correction for the two-pass algorithm.

The eventual proliferation of WCET analysis tools will therefore become an instance

of N-version programming. Teams of programmers from around the globe will be working independently to create the same type of tool, each team implicitly providing a check for the other, helping to ensure that static WCET analysis is safe. Importantly, the work of creating these N redundant implementations is *distributed*, making the cost of N-version programming virtually free. It is also faster and simpler than previous techniques for ensuring accuracy, such as those based on test case generation [230], because it does not require a vast input space.

Of course, this scenario assumes that a sufficient number of interoperable analysis tools will one day be readily available, and thus Clepsydra is hopefully only one of many WCET tools that are yet to come.

Chapter 7

Interactive Timing Analysis of Software Libraries

Chapter Summary

Context Libraries are vital to the software development process. By providing reusable code, they save developers time that would otherwise be spent creating sorting algorithms and other general-purpose functions. However, most libraries are designed to have good *average-case* performance, and little thought is given to their *worst-case* performance. As a result, the execution time of their operations may be difficult to predict, making them unsuitable for real-time systems.

Prior Work Prior work in this area is extremely limited. In one study, a set of trigonometric library functions, which are often used in typical real-time applications, were parameterized such that they could be tuned to achieve a balance between accuracy and timeliness. In other work, a library of collection classes, XML parsers, and other utilities was implemented specifically with predictability in mind. Called Javolution, it included special support for the scoped memory facility of the Real-Time Specification for Java.

Problems The problem with Javolution and similar approaches to real-time libraries is that they only *claim* to be predictable. They do not attempt to provide any guaranteed bound on worst-case execution time. For example, the liberal use of exception handling in Javolution prevents these guarantees, since no algorithms yet exist for computing WCET in the presence of exceptions. Its use of scoped memory, which is very difficult to analyze, is also an impediment.

New Claims For hard real-time systems, where guaranteed predictability is not just important but crucial, a new approach to software libraries is necessary. Such libraries should conform to safety-critical specifications that demand complete WCET analyzability and other forms of static verification. Achieving this goal demands certain restrictions: 1) The maximum bound of every loop in the library must be known; 2) exceptions are prohibited; and 3) dynamic memory allocations (after initialization) are prohibited. As a demonstration of how to create an analyzable library, this chapter presents Canteen, a set of predictable collection classes. It provides hard real-time versions of an array, linked list, set, and map, each of which conforms to its equivalent standard Java interface, including support for generics.

Results The Canteen library demonstrates new techniques for replacing dynamic memory allocations with memory pools in complex data structures. It also shows how to solve the special problem of loop bound annotations in libraries, in which the true loop bound cannot be determined until it is paired with user code. Empirical measurements show that these techniques result in predictable performance and memory usage. Further measurements also show that Canteen achieves throughput on par with conventional library code, confirming that speed need not be sacrificed for real-time predictability.

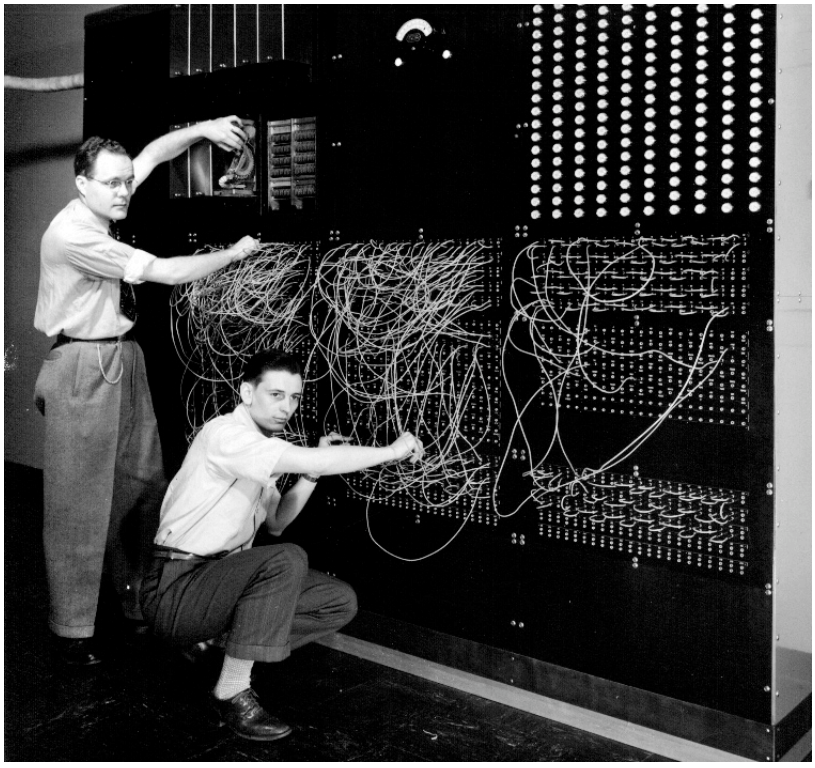

Figure 7.1: Conceived by Harvard professor Howard Aiken, the Mark I was a room-sized, relay-based calculator. (Photograph courtesy of Computer History Museum.)

In the summer of 1944, Grace Murray Hopper reported to the Computation Lab at Harvard University to help program Mark I, the first large-scale automatic digital computer in the United States. Programming the 4500-kilogram machine, shown in Figure 7.1, required punching holes on a piece of paper tape. Because the Mark I was not a stored-program computer, Hopper had to repeatedly code certain instruction sequences that were used again and again onto successive pieces of tape. She soon realized that if new programs could somehow reuse the pieces that had already been coded, much effort could be saved. Subsequent modifications to the Mark I allowed multiple tape loops to be mounted and reused [231]. Hopper's time-saving idea was one of the very first implementations of a *library*.

Libraries are vital to the software development process. By providing reusable code, they save developers time that would otherwise be spent creating containers, iterators,

sorting algorithms, and other general-purpose functions. Libraries also improve code quality: Because they are typically shared among many developers, bugs in libraries are more likely to be detected and fixed. (This is an instance of Linus's Law: "Given enough eyeballs, all bugs are shallow." [232])

In addition, libraries are more likely to offer higher performance and more features because they are usually crafted by experts in the library's domain. An ordinary developer creating the same code for individual use may not have the required expertise nor the time to optimize and fine-tune the code. Consider, for example, the hypotenuse function, $\sqrt{x^2 + y^2}$, which may be implemented in Java as:

```
return Math.sqrt(x*x + y*y);
```

While this code may seem perfectly valid, it is actually a naïve implementation due to its susceptibility to floating point rounding errors [233]. Compare it to the code in Figure 7.2, which shows a much more sophisticated implementation that limits the rounding error to one unit in the last place. The floating point expert (or experts) who designed this code even made it adaptive by having it choose between two different algorithms, depending on the input values, in order to ensure the highest accuracy in all cases. Developers working on application code would be unlikely to go to such lengths should they need to write similar helper functions.

This strategy of encapsulating expert knowledge in a library has been repeated many times over. Libraries for parallel processing are implemented by experts in thread synchronization and concurrency. Libraries for image processing are implemented by experts in hidden Markov models and linear transformations. To put it simply, a library is an expert-in-a-box, ready to be opened when needed [234].

```
/* Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
 * Developed at SunSoft, a Sun Microsystems, Inc. business. Permission
 * to use, copy, modify, and distribute this software is freely granted,
 * provided that this notice is preserved. */

double a=x,b=y,t1,t2,y1,y2,w;
int j,k,ha,hb;

ha = __HI(x)&0x7fffffff;    /* high word of  x */
hb = __HI(y)&0x7fffffff;    /* high word of  y */
if(hb > ha) {a=y;b=x;j=ha; ha=hb;hb=j;} else {a=x;b=y;}
__HI(a) = ha;   /* a <- |a| */
__HI(b) = hb;   /* b <- |b| */
if((ha-hb)>0x3c00000) {return a+b;} /* x/y > 2**60 */
k=0;
if(ha > 0x5f300000) {   /* a>2**500 */
   if(ha >= 0x7ff00000) {   /* Inf or NaN */
       w = a+b;         /* for sNaN */
       if(((ha&0xfffff)|__LO(a))==0) w = a;
       if(((hb^0x7ff00000)|__LO(b))==0) w = b;
       return w;
   }
   /* scale a and b by 2**-600 */
   ha -= 0x25800000; hb -= 0x25800000;  k += 600;
   __HI(a) = ha;
   __HI(b) = hb;
}
if(hb < 0x20b00000) {   /* b < 2**-500 */
    if(hb <= 0x000fffff) {  /* subnormal b or 0 */
        if((hb|(__LO(b)))==0) return a;
        t1=0;
        __HI(t1) = 0x7fd00000;  /* t1=2^1022 */
        b *= t1;
        a *= t1;
        k -= 1022;
    } else {        /* scale a and b by 2^600 */
        ha += 0x25800000;   /* a *= 2^600 */
        hb += 0x25800000;   /* b *= 2^600 */
        k -= 600;
        __HI(a) = ha;
        __HI(b) = hb;
    }
}
/* medium size a and b */
w = a-b;
if (w>b) {
    t1 = 0;
    __HI(t1) = ha;
    t2 = a-t1;
    w  = sqrt(t1*t1-(b*(-b)-t2*(a+t1)));
} else {
    a  = a+a;
    y1 = 0;
    __HI(y1) = hb;
    y2 = b - y1;
    t1 = 0;
    __HI(t1) = ha+0x00100000;
    t2 = a - t1;
    w  = sqrt(t1*y1-(w*(-w)-(t1*y2+t2*b)));
}
if(k!=0) {
    t1 = 1.0;
    __HI(t1) += (k<<20);
    return t1*w;
} else return w;
```

Figure 7.2: This C code is an excerpt of the hypotenuse function in FDLIBM, a floating point math library that was later incorporated into Sun's implementation of the Java virtual machine. All Java developers now benefit from the optimization effort that went into this library.

7.1 Worst-case Execution Time in Libraries

Today, almost every new program relies on shared libraries for operating system services, middleware, graphics, and so on. Popular implementations include the Standard Template Library for C++, the Base Class Library for .NET, and the Class Library for Java. They save developers from having to reinvent the wheel when sorting lists, parsing strings, and performing other common tasks.

While these general-purpose libraries provide a variety of helpful features, they are typically designed to have good *average-case* performance, and little thought is given to their *worst-case* performance. As a result, the execution time of their operations may be difficult to predict.

For example, Java's ArrayList class provides the operation add to append an item to the end of a list. The operation normally executes quickly and in constant time, but if the additional item would exceed the current size of the array, a larger array is created, and the old array's contents are copied into it. Allocating and copying memory are expensive linear-time operations, and as a result, the add operation suffers from a broad variation between its best- and worst-case times.

Figure 7.3 illustrates this behavior in more detail. The data points show the amount of time that the add operation takes as the size of the list increases. The red data set represents the typical case of an ArrayList that grows automatically as new elements are added. Note that almost every operation completes in virtually no time, and yet a few operations (approximately at list sizes of 500, 2500, and 8000) take far longer. These points indicate where the ArrayList was forced to create a larger copy of its internal data store.

The graph reveals a common behavior among standard library operations: Usually

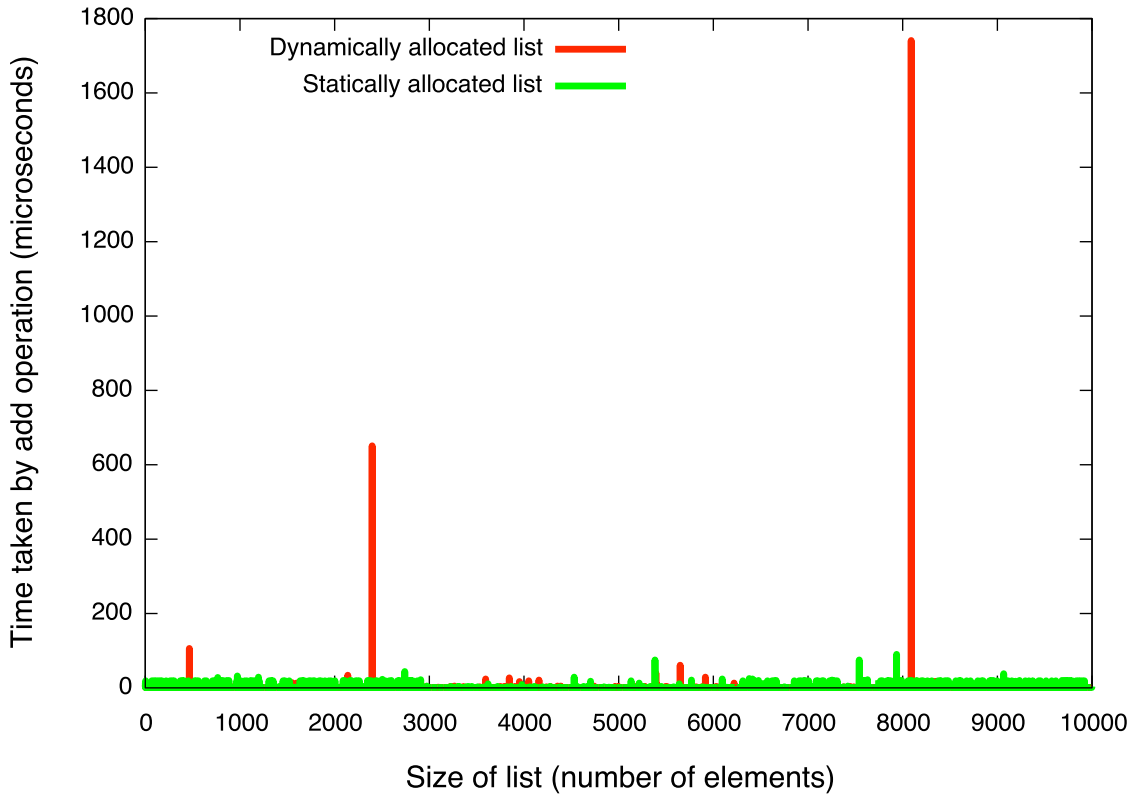

Figure 7.3: Traditional libraries are designed for dynamic allocation, as in this example of Java's ArrayList, which leads to extremely high worst-case execution time.

they execute very quickly but *occasionally* very slowly. In most computing environments, such a variation is not a serious issue. Many applications can tolerate the occasional slowdown as long as the average performance is still good. In real-time systems, however, this unpredictability can cause disaster. If, for example, the software controlling a robotic arm assumes that an operation always executes quickly, any unexpected delay of the operation would break this assumption, disrupting the delicate timing of the system and potentially damaging the arm or the objects it manipulates.

For this reason, standard libraries are conspicuously absent in hard real-time systems. Re-use of code in this domain is typically small-scale and *ad hoc* so that all of the details affecting timeliness can be tightly controlled.

7.2 Goals for Hard Real-time Libraries

A naïve solution to this problem is to take the conservative approach. The software could assume that every library operation exhibits its worst-case behavior. While this tactic prevents scheduling errors by taking into account any unexpected delays, it is also tremendously inefficient. The system would lay idle most of the time, waiting for a potential slowdown even when it does not occur. (A faster processor could reduce the idle time, but this too would be a waste of resources.)

A far superior solution is to design a library specifically for the needs of real-time systems. The goal is to provide general-purpose, reusable code without sacrificing guarantees that the system will meet its hard deadlines. To provide these guarantees, a library must have *known execution time*.

Knowing the execution time demands that all operations in a library be statically analyzable for WCET. This requirement allows WCET analysis tools [135, 193] to verify the timeliness of the system as a whole.

As a corollary, the maximum bound of any loop must be known in order to perform the analysis. For example, the library may have to provide specific information about its loops, perhaps via an annotation mechanism like those described in Section 6.4.2.

7.3 Libraries for Real-time Java

This goal of designing libraries for real-time systems is challenging enough, but it is even more problematic when using Java. The issue of worst-case performance, as in Figure 7.3, is only part of the problem. The biggest hurdle, especially when targeting the Real-Time Specification for Java (RTSJ) [29], is that Java programs tend to rely

on certain idioms and design patterns that clash with the demands of a real-time environment.

Specifically, many conflicts arise from the RTSJ's scoped memory model, a technique for preventing garbage collection delays [72]. When objects are used in mixed contexts—that is, shared by heap and non-heap objects or accessed from different scoped memory areas—the restrictions of the memory model can lead to illegal assignment errors, memory leaks, and other such problems. The standard Java library was never designed to handle these restrictions and therefore does not avoid using objects in mixed contexts, making the library virtually unusable. RTSJ programs cannot even use simple mutable Java classes and must resort to providing their own RTSJ-friendly replacements. (The RTZen middleware library [235], for instance, includes a custom class called FString as a replacement for the standard StringBuffer class.)

Although the RTSJ is silent on the problem of mixing the standard Java class library with a real-time application, this has not stopped developers from attempting to do so. As the official Java library for over a decade, the standard classes have been widely tested and are feature-rich, making them very tempting for developers who wish to avoid implementing and debugging classes that already exist. As a result, some RTSJ developers attempt to work around the problems instead of simply doing without the standard class library.

For example, the scoped memory and real-time thread features of the RTSJ are intended for time-sensitive tasks, such as reading sensor data into a buffer. These critical tasks may be relatively simple and small, and thus they may not require a general-purpose library. Instead, the critical threads can transfer data to non-real-time threads, which can then safely use the standard libraries as needed.

While this tactic is feasible, it limits the ability of real-time threads to do useful work. The increasing sophistication of real-time systems means that these threads are being called upon to perform increasingly complex tasks that necessitate the use of a library. Incorporating such libraries into an RTSJ application demands special compromises and strict programming discipline in order to avoid the complications of scoped memory. Workarounds include:

- Avoiding sharing across scoped memory contexts
- Performing initial accesses in the proper memory area to prevent timing delays caused by lazy initialization
- Explicitly switching memory areas to perform certain library operations

These extra steps can adversely impact developer productivity. Having to take special precautions whenever a library operation is invoked can offset whatever productivity gains were achieved from using the library. For these reasons, the standard Java class library is impractical and largely incompatible with real-time applications.

7.4 Related Work

Given that standard libraries are incompatible with the goals of real-time systems, the strategy presented in Section 7.2—custom libraries designed specifically for predictability—is an effective solution. Such libraries can, in most cases, implement the same interfaces as the standard libraries, making their adoption largely a plug-and-play affair.

While there has been very little prior work in this area, two notable efforts have taken a similar approach, as described in the following sections.

7.4.1 Trigonometric Library Functions

Kirner et al. analyzed the resource requirements of different implementation techniques for mathematical functions [236]. They provided experimental results for trigonometric functions, which are often used in the real-time domain. Computing the translational movements of a robot arm, for example, relies heavily on trigonometry.

The classical approach of calculating a trigonometric function is to calculate its Taylor series iteratively. The advantage is fast convergence: A Taylor polynomial of 14 degrees is sufficient to precisely calculate the trigonometric function of a double value.

The authors compared this common approach against two alternative implementation techniques: 1) Using a bounded degree of the Taylor polynomial, as not all applications require full double preciseness, and 2) using precomputed values—a lookup table—to approximate the result.

Both approaches showed interesting alternatives in the context of real-time computing. For example, the work demonstrated that trigonometric algorithms may be parameterized in order to provide a balance between accuracy and timeliness. Due to the fast convergence of the Taylor series, however, the lookup table is viable only for relatively small degree (about six) Taylor polynomials.

7.4.2 Javolution

Javolution [237] is another example of a custom library for real-time systems. Released as an open-source project in 2004, it replaces Java's standard collection classes (List, Map, Set, etc.) with custom versions designed to have more predictable response time. It is intended to eliminate problems such as:

- Large arrays allocated and copied for internal resizing, resulting in large worst-case times
- Long garbage collection pauses due to memory fragmentation when large collections are allocated
- Sudden bursts of computation (e.g., internal rehashing of hash maps or hash sets)

Figure 7.4 elucidates the latter problem: a delay caused by rehashing. Note that the performance of the FastMap's put operation remains relatively constant, while the standard HashMap class suffers from occasional delays that are orders of magnitude larger than its expected time.

Javolution also includes special support for the scoped memory model of the RTSJ. Unlike the standard Java library, which is susceptible to memory leaks and illegal access errors under the RTSJ, Javolution is scope-safe. It allocates additional memory for a collection from the same memory area as the collection itself. Consider, for example, a map allocated in immortal memory:

```
static Map<Foo, Bar> map = new HashMap<Foo, Bar>();
```

In an RTSJ environment, this code will cause memory leaks when entries are removed. It will also cause illegal assignment errors if new entries are added from a scoped memory area.

With Javolution, these problems can be eliminated simply by instantiating a different kind of map:

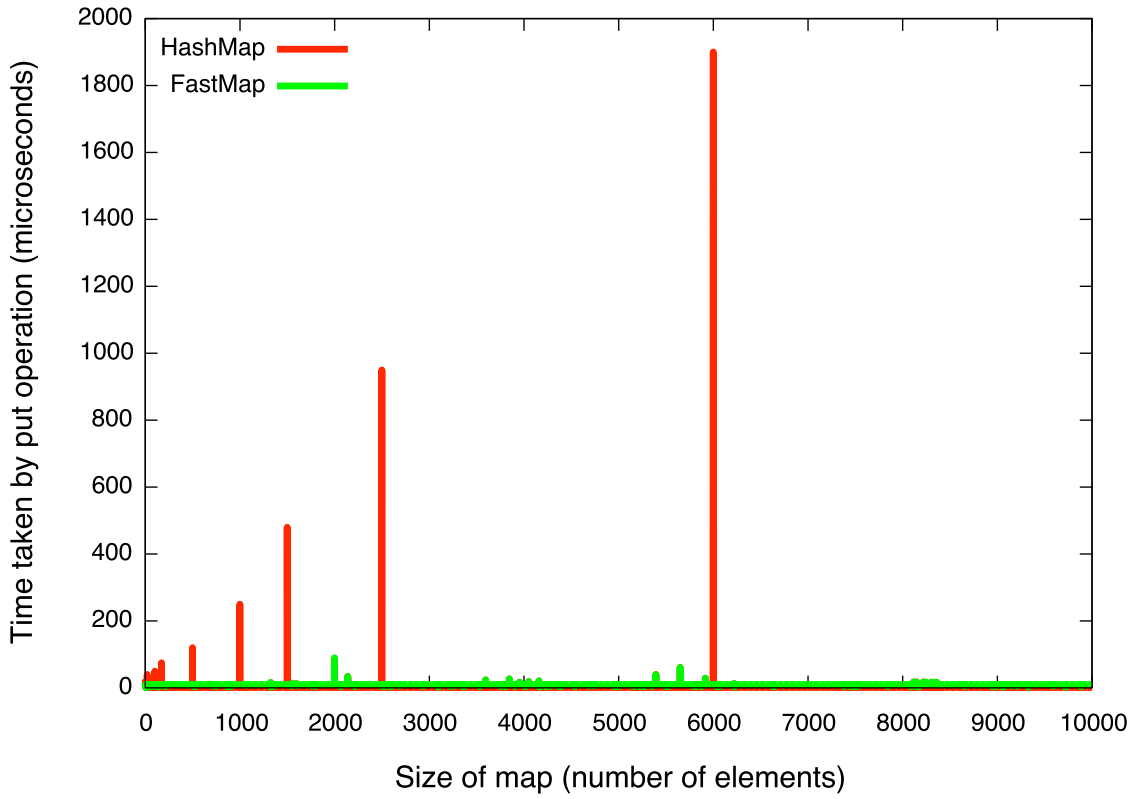

Figure 7.4: Java's standard HashMap class exhibits poor worst-case running time, while Javolution's FastMap class, designed for real-time systems, is much more predictable.

```
static Map<Foo, Bar> map = new FastMap<Foo, Bar>();
```

With this change, the map's entries are internally recycled, and new entries are placed in immortal memory, thereby circumventing the run-time errors that would otherwise result from using the RTSJ's memory model.

Javolution achieves these goals largely by judicious memory management, always favoring time predictability over space efficiency. For instance, it ensures that any capacity increase of a collection occurs smoothly, allocating in small increments instead of a single large chunk. In particular, the FastTable class relies on multi-dimensional arrays to avoid resizing and copying its internal data store.

Another instance of Javolution's space-time tradeoff is its FastMap class, whose entries each have their own entry table. When the map's size increases beyond capacity, it allocates new, larger tables for the entries. Because the old entries are not moved, the map does not suffer from delays incurred by rehashing.

Despite these advantages, Javolution only *claims* to be predictable. It does not provide any bound on worst-case execution time, for example. Its liberal use of exception handling also prohibits static analysis by the current generation of WCET tools. In short, Javolution is ideal for soft real-time systems, but it is inadequate for hard real-time systems where timeliness must be guaranteed.

7.5 Libraries for Safety-critical Environments

Clearly, the idea of a custom real-time library is not entirely new, yet the amount of prior work is quite limited, and there exist many avenues for improvement. One overlooked aspect is the need for safety-critical analysis.

Software is safety-critical[1] when failure of that software can harm human beings. An experimental robot car that drives around a test range in a controlled environment is not safety-critical. A car that drives itself autonomously on public roads, with or without occupants, *is* safety-critical.

Creating software for these safety-critical environments is largely a matter of testing and validation. The goal is to verify, to as high a degree as possible, that the software will work as intended. This requirement presents a new angle on real-time systems that essentially asks, "How many ways can this software go wrong?"

In most cases, answering this question is the job of an analysis tool. Such tools perform

[1]Some researchers use the term "high-integrity" as a synonym for "safety-critical."

static checks to detect inconsistencies and errors in the code, or they may integrate a formal safety specification into the software development process. Examples of such tools include the static analyzer SPARK Examiner [238] and the code generator SCADE Suite [239]. The ability of these tools to verify code correctness is the first step toward obtaining a safety-critical certification such as DO-178B [38].

Most software libraries, however, are an obstacle toward certification. For instance, Javolution is incompatible with current analysis tools because it is peppered with exception handling and, to a lesser degree, dynamic memory allocations. These features typically are beyond the capabilities of the current crop of safety-critical analysis techniques. Ironically, even the strictly defined RTSJ presents a problem because it still allows complex behavior such as dynamic class loading and asynchronous transfer of control (ATC), both of which are extremely difficult to analyze [240].

A typical solution to this problem is to limit the capabilities of the system, preventing the use of features that might complicate testing and verification. The idea is to reduce functionality just enough to allow analysis tools to do their job. SPARK, for example, is a formally-defined Ada-based language that has, according to its author, "just those features required for writing reliable software: not so austere as to be a pain, but not so rich as to make program analysis out of the question." [241] The restrictions of SPARK allow programs to be proven correct.

A related specification, known as Ravenscar,[2] further restricts Ada's tasking model to facilitate static analysis [242]. It forbids dynamic priorities, ATC, dynamic interrupt handling, and other problematic features of task control.

Since the advent of SPARK and Ravenscar in the late 1990s, the pool of programmers

[2]The Ravenscar profile was named after Ravenscar, England, the town in which it was first conceived, but a backronym has since been applied: *Reliable Ada Verifiable Executive Needed for Scheduling Critical Applications in Real-Time.*

who know Ada has shrunk due to its lack of success outside the realm of safety-critical systems [243]. Meanwhile, Java has exploded in popularity, prompting researchers in the safety-critical community to ask whether the principles of SPARK and Ravenscar could be applied successfully to Java.

Some features of Java naturally facilitate the same kind of program verification as enabled by SPARK and Ravenscar. It disallows pointer arithmetic, for example, and it enforces definite assignment. At the same time, other intrinsic Java features hinder static analysis. Dynamic dispatch and garbage collection thwart memory usage analysis, WCET analysis, and schedulability analysis that safety-critical systems require.

To resolve these problems, several proposals are in development that take essentially the same approach as SPARK and Ravenscar. Here again, the goal is to eliminate certain features that are known to inhibit analysis techniques, in effect creating a subset of the standard Java environment. One such proposal is evolving under the auspices of Sun as JSR-302 [244], a community-driven process that should eventually result in a formal specification for safety-critical Java. Mandates of JSR-302 include:

- Garbage collection is not assumed
- All classes are loaded during an initialization phase, not dynamically
- All exception objects must be preallocated
- Three threading models: cyclic executive, single mission, and nested mission

This list is only a sample of the major restrictions of JSR-302; others are still in discussion. The process has been slowed somewhat by earlier proposals for safety-critical Java that are merging with JSR-302, such as the HIJA project [245], the "scalable Java" guidelines from Aonix [246], and Ravenscar-Java [247]. The direction

of JSR-302 is also not universally accepted; a proposal by Schoeberl argues against its approach of subsetting the RTSJ [248, 249].

In the end, however, one of these efforts should culminate in a safety-critical specification for Java that offers performance and resource consumption comparable to C and Ada, satisfies DO-178B Level A[3] certification requirements, and provides the same language syntax and build tools of standard Java.

7.6 Requirements for an Analyzable Real-time Library

Even though a large subset of hard real-time systems falls into this safety-critical category, existing approaches for software libraries are useless in such an environment. Even specially crafted libraries for real-time systems, such as Javolution, suffice only for *soft* real-time systems, where predictability is important but not crucial, and formal analysis is an afterthought. For systems where it is not merely important but must be *guaranteed* to preserve life and limb, the library must do much more to facilitate WCET analysis and other forms of verification. Without this extra support, no strong claims can be made about the safety of the system as a whole.

Based on these observations, a library for hard real-time systems should conform to the restrictions required by safety-critical specifications such as JSR-302. The key requirements of such a library can therefore be refined from the broad goals presented in Section 7.2 into the following:

- All operations in the library must be statically analyzable for worst-case execu-

[3]Level A is the most stringent safety assessment process in DO-178B; it assumes that failure of the system would result in a catastrophic condition.

tion time (WCET). This allows tools such as WCA [135] or Clepsydra [193] to verify the timeliness of the system.

- As a corollary to the previous requirement, the maximum bound of any loop must be known in order to calculate the WCET. Thus, information about all loop bounds must be incorporated into the real-time library. These bounds may be supplied through direct manual annotations [218] or by describing pre- and post-conditions for methods (using a tool such as the Java Modeling Language [250]) and then applying a semi-automated prover (such as KeY [251]).

- All code, even low-level device driver code, must be analyzable. Limited hardware interaction such as reading and writing device registers is allowed, but the methods to do so must have predictable execution time and be visible to the analysis and verification tools.

- The library must not include any support for exceptions. Although exception handling improves code quality by enforcing proper error checking, it also makes the control flow of a program so complex that validating all program paths becomes an intractable problem. Safety-critical certification standards such as the DO-178B specifically forbid reliance on exception-based software due to the lack of pathway predictability that inhibits test coverage analysis (e.g., MC/DC [252], LCSAJ [253], etc.). Likewise, safety-critical software implementations such as SPARK leave out exceptions because they make formal verification much more difficult. They create too many possible exit points for a function, potentially leading to catastrophic failure.[4]

- Dynamic memory (e.g., `new` and `delete`) is unpredictable and very difficult to analyze, so it too is disallowed. Instead, all data structures must be allocated

[4] A real-life example of an exception-handling disaster can be found in the first test flight of the Ariane 5 [254], perhaps the most expensive computer bug in history.

during initialization in order to guarantee that the system never runs out of memory. After initialization, no further allocations are allowed. This restriction enables formal analysis that would otherwise be impossible with today's tools due to the complexity of dynamic memory models.

- The requirement of known execution time has a side-effect on virtual memory. A common feature in operating systems since the late 1980s, virtual memory has simplified application development because it gives the developer practically unlimited memory. This abstraction is not without cost, however: It makes access to memory very unpredictable. A single machine instruction could result in several page faults, each costing hundreds of thousands of clock cycles. Time-predictable execution of tasks therefore requires that virtual memory be disabled. (This is no great loss because a memory-predictable library makes virtual memory irrelevant. The total memory consumption is known ahead of time, and therefore the system can be configured with exactly the amount of memory that the software requires.)

7.7 Canteen: A Prototype for an Analyzable Library

To explore how a software library can meet these requirements, the Volta project (first described in Section 2.4) includes a prototype of a fully analyzable general-purpose library called Canteen[5] [255]. Designed to allow formal analysis for both hard real-time and safety-critical applications, Canteen provides Java implementations of the three most common types of collection interfaces: List, Set, and Map. The implementations

[5]As with all components of the Volta project, Canteen is distributed under an open-source license to invite critical comparison and make the reported results verifiable.

include:

PredictableArrayList A simple random-access sequence that gives precise control over element ordering. It allows multiple entries of the same element, including null. It is implemented as a simple linear array.

PredictableLinkedList Identical to PredictableArrayList but implemented as a linked list. Insertion and removal are much faster, but random access is much slower.

PredictableTreeSet A sorted collection that guards against duplicate elements. As the name implies, it models the mathematical abstraction known as a set. It is implemented as a red-black tree in order to ensure a worst-case running time of $log(n)$ for all operations.

PredictableTreeMap A sorted dictionary-type collection that maps keys to values. Each key can map to at most one value. It is also implemented as a red-black tree for the same reason.

These implementations meet all of the requirements spelled out in Section 7.6. They also offer additional features for convenience, type-checking, and compatibility. To be specific:

All operations statically analyzable for WCET The classes in Canteen are designed to facilitate static timing analysis. For example, recursion is non-existent, and all loops have been annotated to provide a finite bound on their iteration count.

No exceptions thrown after initialization Because exception support makes validation much more difficult, the Canteen classes never deliberately throw unchecked exceptions, and none of their methods declare checked exceptions.

No memory allocation after initialization Following the trend of safety-critical systems, including the latest specifications such as JSR-302, Canteen avoids Java's new operator entirely, relying instead on internal memory pools that recycle objects in constant time. (During initialization, the classes are free to pre-allocate memory from Java's heap for the pools, iterators, and other necessary structures.)

Support for Generics In early versions of Java, collection classes were simply buckets of objects. Nothing prevented a buggy program from putting, say, integer objects into a set meant to contain only strings. Such mistakes resulted in run-time errors that were difficult to isolate. With Java 5, collection classes can now take parameters that indicate the kind of objects a collection may contain. These parameters, known in Java as *generics*, allow the compiler to enforce type-checking rules on collection class operations. The Canteen library fully supports this feature and thus facilitates even stronger validation of correctness.

Compatibility with standard Java interfaces The collection classes in Canteen implement the same List, Set, and Map interfaces declared in Java's standard library. The classes can therefore act as drop-in replacements for the standard collection classes, allowing reuse of existing Java code (with some caveats, such as the mutable object problem outlined in Section 7.10.1).

With these capabilities, an interactive analyzer such as Clepsydra can immediately and automatically display the WCET for each line of code, not only in Canteen itself, but in applications that invoke the library as well. Whereas before only the application code could be analyzed, now the entire software stack is integrated into the analysis process: The WCET of the hardware platform is known, the WCET of the libraries is known, and now the WCET of applications built on top of shared libraries can be also known.

7.8 Prototype Design and Implementation

The collection classes in Canteen are a typical use case that make this prototype a suitable vehicle for research. They are complex enough to demonstrate how timing analysis can be applied to software libraries, yet not so simple as to sidestep the difficult design choices that must be made in order to create a library analyzable for both hard real-time and safety-critical requirements.

In keeping with this role as a stepping stone to future research, Canteen includes an API reference with each method fully documented. It also includes a suite of 72 test cases to verify correct functional behavior of each method. A notable benefit of these test cases is that they can run in "standard" mode, such that new objects are obtained from the heap, instead of Canteen's memory pools, and the standard Java collection classes are used instead of Canteen's time-predictable versions. All test cases pass in both modes, demonstrating that the library can act as a drop-in replacement for the standard Java collection classes.

7.8.1 Analyzable Memory Consumption

The most extreme design element in Canteen is the way it makes memory allocation analyzable. As noted in Section 7.5, dynamic memory allocation from an arbitrary heap, along with a garbage collection thread to clean up unused allocations, is extremely difficult to analyze. Instead, Canteen pre-allocates all of the memory for a collection class instance when it is first initialized. When the collection object requires memory, it retrieves it from this pre-allocated pool and then returns it to the pool when finished.

This classic *memory pool* technique is analyzable because it is much more determin-

istic than dynamic heap allocation. With a dynamic heap, each allocation or deallocation could potentially trigger a rearrangement of the heap, whereas in a memory pool, there is no fragmentation, and blocks are obtained and returned in constant time [256]. The simplicity not only makes the technique comparatively easy to analyze but also offers the predictable performance required for real-time systems.

Memory pools are not a new idea. They are common in real-time operating systems such as IBM's Transaction Processing Facility. They can even be approximated to some degree without changing any library code at all. For instance, a standard Java library class such as ArrayList can be initialized with a user-specified size for its internal array. If this size is at least as large as the number of elements that will ever be added to the list, then the slowdown caused by reallocating the array will never occur. The result is that the list's add operation behaves predictably, without any major spikes in its timing, as shown by the green "statically allocated" data set of Figure 7.3.

The downside, of course, is that pre-allocating all data structures that could potentially be used by a program is an inefficient use of memory. For example, consider a program that allocates two ArrayLists but never needs both lists allocated at the same time. A heap approach could use the same memory for both lists, dynamically allocating and deallocating space as needed. A pre-allocation scheme, on the other hand, would require about twice as much space because it would allocate memory for both lists on initialization and never release it.

A more serious problem is that pre-allocation of a standard library's data structures is often impossible. Unlike ArrayList, most library classes cannot be manipulated by the user to pre-allocate all of the memory that they require, simply because they were never designed to do so. Even when such a feat is possible, knowing how to perform the proper initialization requires intimate knowledge of a library's internal structures,

negating the benefits of data abstraction that the library provides.

Canteen avoids this latter problem by providing direct support for memory pools from within the library itself. It abandons dynamic heap management entirely (except during initialization) and relies instead on memory pools to retrieve pre-allocated elements for adding to a collection. When an element is removed from the collection, Canteen returns it to the collection's pool for later use. This approach provides three distinct advantages:

- Unpredictable delays that would otherwise be caused by garbage collection and heap fragmentation are eliminated.
- The total memory consumption is known immediately after initialization.
- The user of the library does not need to be aware of the library's internal structures in order to pre-allocate memory.

These benefits do not come without cost, however. Just as loop bounds must be known at compile time for WCET analysis, the size of every memory pool must be pre-defined, as well. The developer must supply and maintain this information, making it error-prone, and the effect of accidentally exceeding the size at run-time is undefined.[6] Having to manage memory as static pools rather than as a dynamic heap also conflicts with many standard programming techniques such as iteration and exception handling, as discussed in Section 7.10.1. The Canteen library assumes that developers are willing to accept these limitations in order to gain the benefits of memory pools.

[6]One might consider dealing with memory pool exhaustion by logging a warning and reverting to dynamic allocation. This option is not viable, however, because inserting dynamic allocation into the execution path greatly complicates WCET analysis, which is exactly what memory pools are intended to avoid. Even if analysis is possible, the predicted WCET would be extremely pessimistic because it would have to assume in every case that dynamic allocation could occur even though it rarely would.

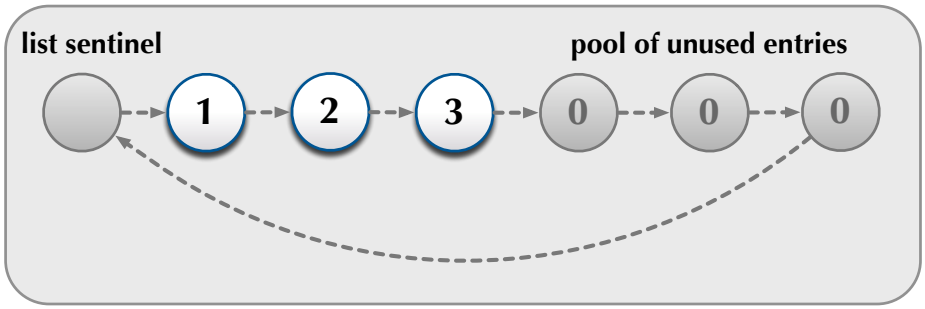

Figure 7.5: The memory pool in Canteen's PredictableLinkedList is attached directly to the list itself. When elements are added to or removed from the list, they are simply swapped between the used and unused portions. Clients are unaware of the pool's existence because all access to the list is protected by the java.util.LinkedList interface.

Implementing memory pools in the List classes is straightforward. For the PredictableArrayList, the memory pool is simply stored at the unused end of the array. Adding a new element increases the boundary between the used and unused portions of the array by one. Removing an element shifts all subsequent elements back by one, and the position that was occupied by the last used element now holds the first element of the memory pool. The memory pool in the PredictableLinkedList class operates in a similar fashion (see Figure 7.5); the only major difference is that it does not need to shift elements during removal.

For the PredicableTreeSet and PredicableTreeMap, however, supporting memory pools is much more involved. These classes are backed by a red-black tree structure, and the memory pool must be maintained across the elaborate rebalancing operations that occur after adding or removing elements in the tree. A simple array cannot act as the pool because adding or removing an element may require shifting elements (a linear-time operation) as well as updating the left and right links of all nodes to account for the shift (another linear-time operation).

To preserve the intended logarithmic time for all operations in Canteen's Set and Map classes (see Table 7.1), a *list-tree hybrid* data structure was developed. It consists of

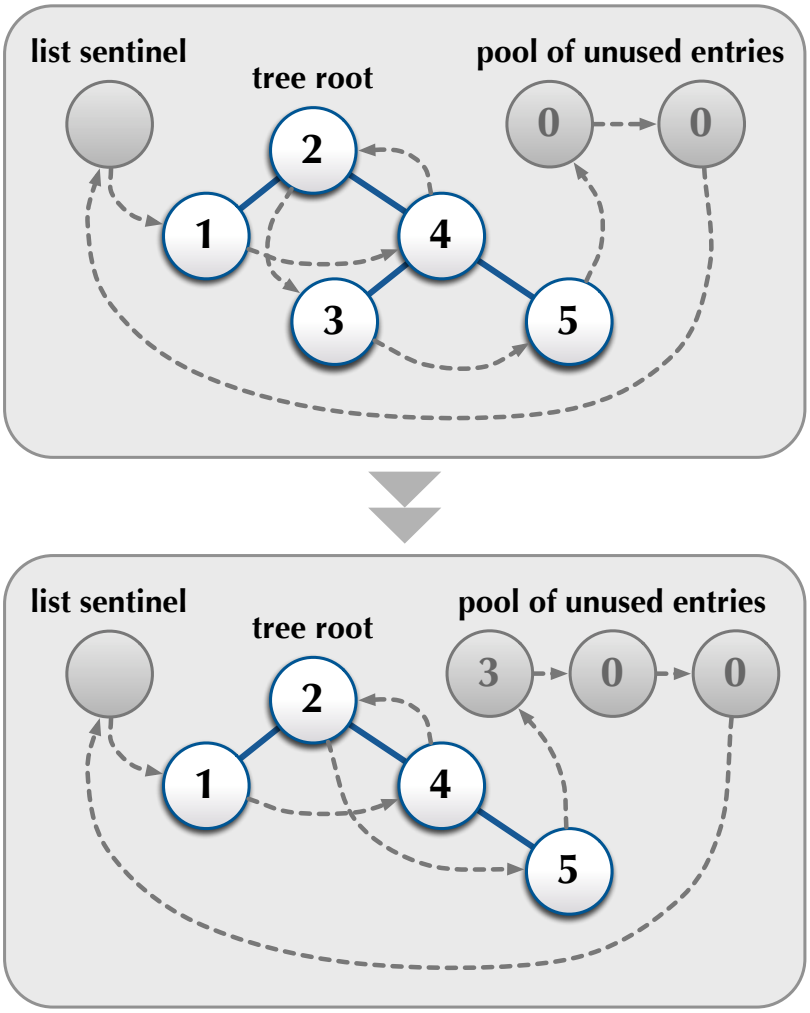

Figure 7.6: This hybrid list-tree data structure allows binary trees to use memory pools. In the diagram, element #3 is being removed from a tree of size 5 and maximum size 7. The element is returned to the pool, and the list pointers are updated accordingly.

a normal binary tree whose entries contain the usual left and right pointers to child nodes. In addition, each entry also contains previous and next pointers to form the doubly-linked list of a memory pool; this allows entries to be removed and returned to the memory pool in constant time. (If the list were singly linked, the removal would degrade to linear time.) The red-black tree algorithms in Canteen were then modified so that these list pointers are properly updated across all rebalancing operations. Figure 7.6 shows an example of how the list-tree hybrid is altered by the removal of an element.

7.8.2 Analyzable Loops

Memory management is not the only complicating factor in making a software library analyzable. Loops also present a problem for analysis tools that try to place an upper bound on execution time. A pure control flow analyzer, such as Cascade, has no way of knowing how many times a loop will execute in the worst case, making the WCET of any loop effectively unbounded.

Canteen solves this problem with a conventional source code annotation approach like the one described in Appendix B. It adopts Java's built-in annotation mechanism and extends it to allow annotations directly on loop constructs. This provides "for free" syntax checking, type safety, and support from existing annotation processors.

The following code snippet shows an example of the annotation syntax:

```
@LoopBound(max=10)
for (int i = 0; i < 10; i++)
  ...
```

For libraries, however, this annotation is insufficient. The maximum number of iterations for loops in a library cannot be fixed to any hard bound; it depends on how the library is used. For example, if two instances of an ArrayList have maximum sizes of 10 and 100, then the loop bound of a search operation for the latter list will be 10 times greater than the former. Therefore, the loop bound annotation of a library method cannot be specified using a simple constant.

To address this issue, loop bound annotations in Canteen are expressed using specially named negative constants. These "magic numbers" tell the WCET analyzer that the true loop bound depends on how the library is used. For example:

```
// Canteen code
class PredictableList ... {
  ...
    public int indexOf(Object o) {
      @LoopBound(max=COLLECTION_BOUND)
      for (int i = 0; i < array.length; i++)
        ...
```

When the WCET analyzer sees this COLLECTION_BOUND value, it knows to look at the code that invoked the method to determine the true loop bound. In Clepsydra, for instance, the default loop bound strategy will search for a class field in the user code that invoked the Canteen method. It will then extract the library's loop bound from the annotation on this field. For example:

```
// User code
@CollectionBound(max=1024)
private PredictableList<Int> list;
...
int index = list.indexOf(...);
```

Here, the analyzer would determine that the value of 1024 should be mapped onto the COLLECTION_BOUND symbol for this particular invocation of indexOf.

Although this technique works for the typical usage of collection classes, it has limitations. If, for example, the field is aliased to some other variable, the WCET analyzer will retrieve the wrong CollectionBound annotation:

```
@CollectionBound(max=1024)
private PredictableList<Int> list;
```

```
@CollectionBound(max=2048)
private PredictableList<Int> other;
...
list = other;
```

In this case, the WCET analyzer will incorrectly use 1024, instead of 2048, as the loop bound. Likewise, the analyzer would be unable to determine the loop bound if the library object is declared as a local variable or method parameter instead of as a class field. In addition, the usual disadvantages of source code annotations still apply: They must be inserted manually and are therefore error-prone.

Fixing these problems would require a sophisticated data flow analysis tool that does not yet exist for Java. For now, source code annotations are a workable alternative that make WCET analysis much faster and simpler.

7.9 Prototype Evaluation

As with any implementation of collection classes, functionality is not the only important attribute. Users must also know the expected running time of the collections in order to choose the correct implementation (e.g., linked list vs. array-based list) and to predict the performance of programs that use them.

Table 7.1 summarizes the time complexity of the basic operations in each Canteen class. Note that all of the complexities shown are asymptotically identical to the standard non-real-time algorithms, demonstrating that performance need not be sacrificed to ensure predictability.

Naturally, asymptotic notation can only describe the overall complexity of an algo-

Table 7.1: Time complexity of the Canteen classes, where n is the element count and $lg(n)$ is the base-two logarithm of n. The index operation refers to retrieving an element based on its position in the list.

	insert	delete	search	index
PredictableArrayList	$O(n)$	$O(n)$	$O(n)$	$O(1)$
PredictableLinkedList	$O(1)$	$O(1)$	$O(n)$	$O(n)$
PredictableTreeSet	$O(lg(n))$	$O(lg(n))$	$O(lg(n))$	N/A
PredictableTreeMap	$O(lg(n))$	$O(lg(n))$	$O(lg(n))$	N/A

rithm. The true performance can vary greatly, even between two implementations of the same basic algorithm. Canteen's performance should therefore be analyzed in more detail.

7.9.1 Performance Analysis

One way to compare the performance of Canteen versus existing collection class libraries is with simple benchmarks. Canteen includes a set of benchmark programs, each of which measures the execution time of a particular operation. Due to the interchangeable nature of Java's collection class interfaces, this same set of benchmarks can be run against other libraries for an objective comparison.

Toward that end, an experiment was conducted with the following libraries:

- Canteen
- Class Library for Java 1.5 [7] (the java.util package)
- Javolution 5.2.6 (the javolution.util package)

The benchmarks were run with each of these libraries on a Sun Netra T2000 (configured with 1.2 GHz UltraSPARC T1 processors) using the Sun Java RTS 2.1 virtual

[7] In this section, the "Class Library for Java" is referred to simply as "Java."

Table 7.2: This table lists the four types of collection classes measured in the benchmarks, along with a corresponding implementation of that type from each library.

Type	Canteen	Java	Javolution
random-access list	PredictableArrayList	ArrayList	FastTable
linked list	PredictableLinkedList	LinkedList	FastList
map	PredictableTreeMap	TreeMap	FastMap
set	PredictableTreeSet	TreeSet	FastSet

machine and the Solaris 10 5/08 operating system. The RTS was chosen because it offers a common platform on which all three libraries can run, thus making the results comparable. (Java processors currently do not support the RTSJ features that Javolution requires.) RTS also provides high-resolution timer services that make the System.nanoTime() call, which the benchmarks rely upon, accurate to around four nanoseconds. To further improve accuracy, the RTS was configured to launch in interpreted mode (-Xint), which disables the just-in-time compiler and the unpredictability it might otherwise create.

For each library, the four collection types in Canteen were measured along with comparable classes for Java and Javolution, as described in Table 7.2. (For Java, more than one map class is available, but the TreeMap was chosen for comparison because it implements the same red-black tree algorithm found in Canteen's PredictableTreeMap.)

For each of the four collection types, four operations were measured:

Append Adds a new element to the end of the collection.

Insert Inserts a new element into the collection at a random location. (For lists, a new element may be inserted into the same location during the same test. For maps and sets, the elements are added in a random order, but never the same element twice.)

Search Retrieves a random element from the collection or, in the case of a set collection, tells whether a random element exists in the set. (The same value may be searched for more than once during same test.)

Delete Removes a random element from the collection. (For lists, an element may be removed from the same location during the same test. For maps and sets, the elements are removed in a random order, but never the same element twice.)

These are by far the most common and most vital operations provided by a collection. Limiting the scope of the measurements to just these four allows the performance evaluation to be nearly comprehensive yet relatively small and simple.

The results can be seen in Figure 7.7. Each chart compares the raw throughput of the libraries for the four operations of a particular collection type. The values represent 10,000 invocations of the operation, averaged across three trials. For the operations that were randomized—insert, search, and delete—the same seed for the random number generator used in a particular benchmark was preserved across all three libraries, thereby ensuring fairness of the results.

For the random-access list, Canteen and Java show about the same performance, although Java beats Canteen in the insertion and deletion tests because it can shift elements more quickly using the System.arraycopy method. Canteen is unable to employ this method because it is native code and effectively unanalyzable for execution time.

Javolution fares worse than either Canteen or Java, especially in the insertion and deletion tests. The poor showing can be attributed to its use of the RTSJ API for memory management which, while making Javolution compatible with the RTSJ's scoped memory model, slows performance considerably whenever the list capacity must change.

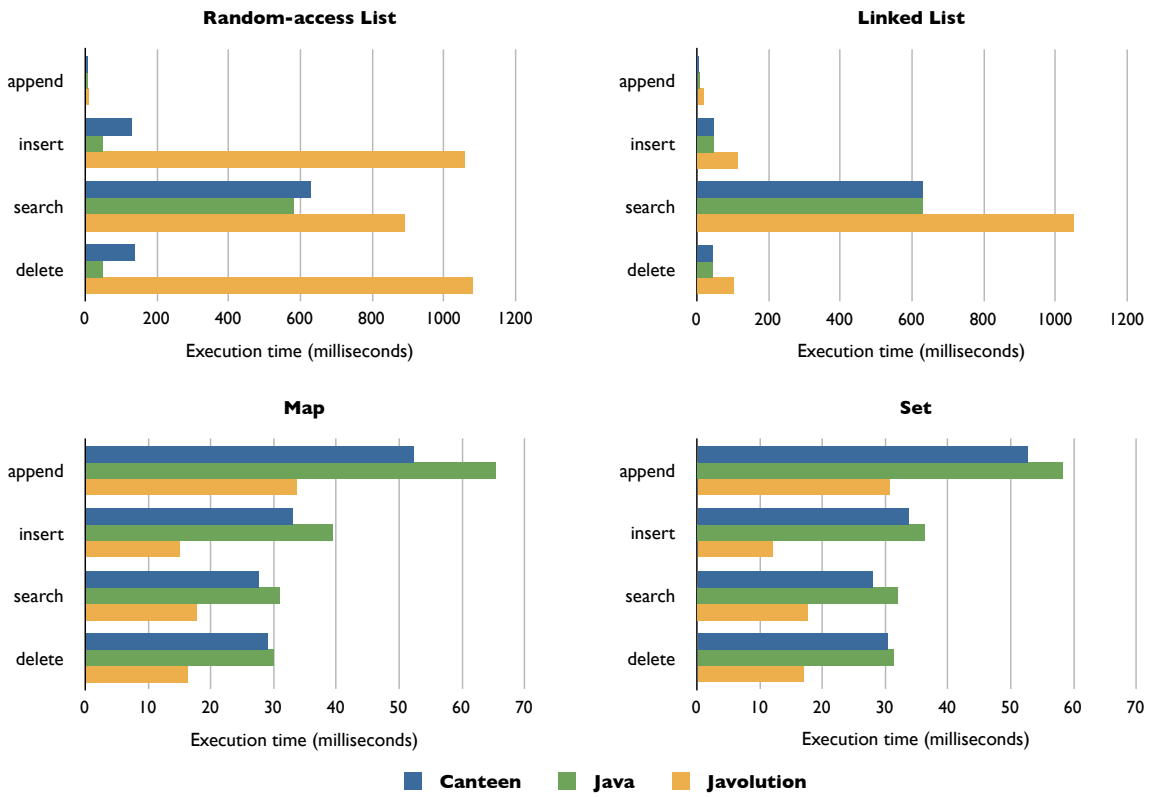

Figure 7.7: These measurements show the time taken for the primary operations—append, insert, search, and delete—of the collection classes in Canteen, Java, and Javolution.

For the linked list, the absence of the arraycopy discrepancy makes the performance of Canteen and Java almost identical. Again, Javolution is a different story. It is much slower because it uses scoped memory to manage the size of the list. Its search operation also suffers because the design expects the JIT to perform method inlining. In this case, however, the benchmarks ran with the JIT disabled, as would be the case for safety-critical systems or a Java processor.

For the map and set collection types, the trends are reversed. Canteen outpaces Java in every test, simply because it pre-allocates collection elements whereas Java allocates them on demand. The two algorithms are otherwise nearly identical. Javolution redeems itself in these benchmarks, but it implements a hash-based algorithm, rather than the tree-based algorithms of Canteen and Java. Although this approach nets a

performance gain, it does so at a cost of predictability, as revealed in the next section.

7.9.2 Predictability Analysis

Since Canteen is intended for hard real-time and safety-critical systems, its predictability is of prime importance. Therefore, in addition to the overall throughput of the library operations, their variability in execution time was also measured. An experiment was conducted, under the same hardware and software conditions of Section 7.9.1, to examine the behavior of the classes shown in Table 7.2. (The set classes are absent from the experiment because each of the three libraries implements sets as maps internally, making the set benchmarks redundant.)

The figures in this section show the results of the experiment. All measurements represent the execution time for an append operation when the collection contains the given number of elements. Other operations could have been measured, but most would yield similar results, and thus the append operation was chosen as a representative sample. (The measurements do not include the first five iterations in order to eliminate the effect of CPU caching in the test system. Such cache effects disappear when running on a Java processor.)

For the random-access list, the results given in Figure 7.8 expose the underlying design of the collection classes. Java and Javolution both experience occasional but extreme delays as their internal array grows to accommodate the size of the collection. Interestingly, Java exhibits a steady geometric expansion of these the delays, owing to the fact that its backing array starts at a size of ten and, when necessary, increases by 50%. The need to dynamically adjust the capacity destroys any semblance of predictability in these two implementations. Canteen does not suffer from this drawback and has a near-constant execution time without any spikes.

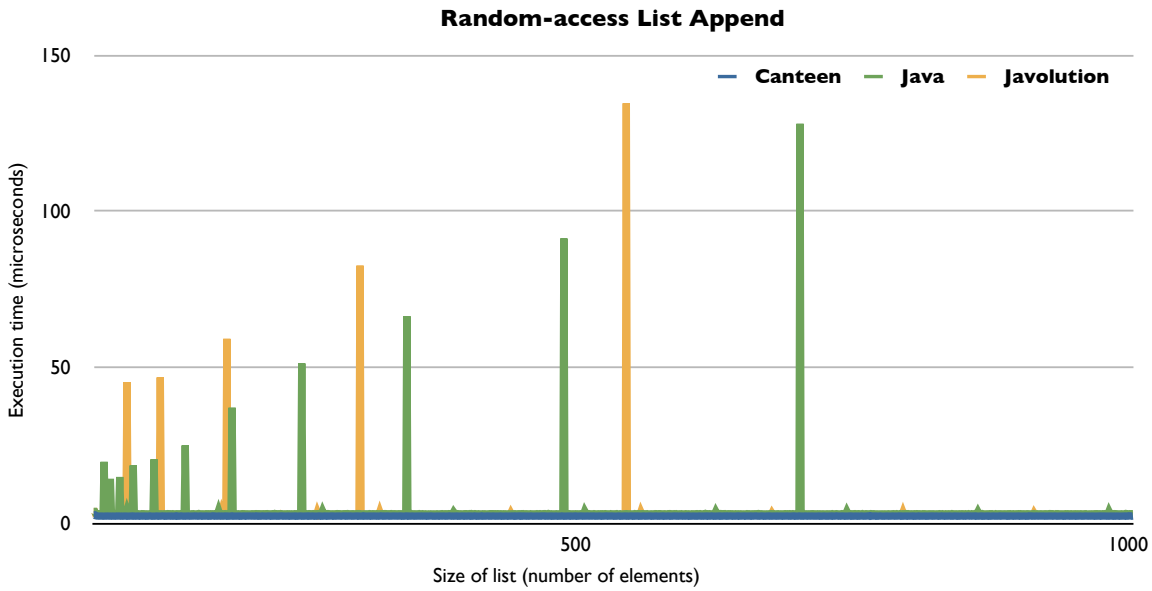

Figure 7.8: These measurements compare the predictability of the append operation for the array-based list classes in Canteen, Java, and Javolution. Each measurement represents the time taken for an append when the list contains the given number of elements.

The results for the linked list, shown in Figure 7.9, provide further evidence for the predictability of Canteen and the unpredictable jitter of Java and Javolution. The latter two libraries dynamically allocate new nodes for the list, causing occasional spikes in execution time. In the case of Javolution, the spikes are shorter and less frequent due to its use of RTSJ's scoped memory feature, which allocates new list nodes in a pre-allocated region of memory. As in the previous benchmark, however, Canteen surpasses both Java and Javolution in speed and predictability. Its execution time is virtually constant throughout all runs.

Finally, the map benchmark reveals some key similarities as well as some fundamental differences among the three libraries. The Java and Canteen libraries, for instance, are remarkably similar in execution time, as illustrated in Figure 7.10. In fact, the patterns are nearly identical because both libraries implement the same red-black tree algorithm. (Java is slightly slower and less deterministic due to its dynamic allocation of new map entries.) The shape of the graph exhibits an unmistakable logarithmic

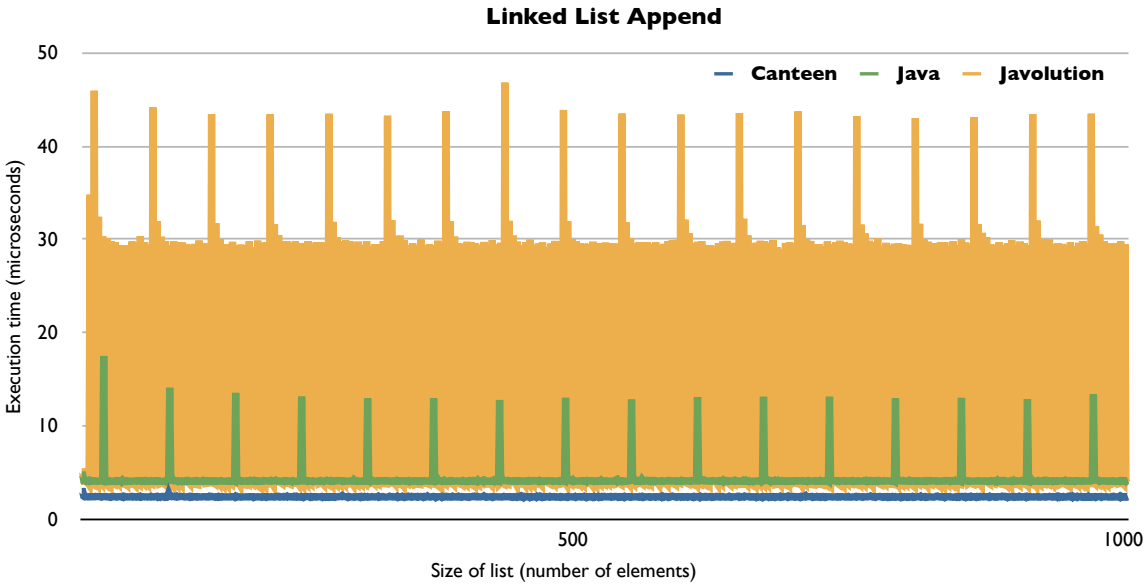

Figure 7.9: These measurements compare the predictability of the append operation for the link-based list classes in Canteen, Java, and Javolution. Each measurement represents the time taken for an append when the list contains the given number of elements.

pattern, just as the algorithm is designed to produce. This deterministic behavior is particularly advantageous for real-time and safety-critical systems.

Javolution is another story entirely. In fact, the results of its map class benchmarks are so different that they had to be presented separately as Figure 7.11. On average, Javolution performs much better than either Java or Canteen, but this is only because it implements its map as a hashtable rather than a red-black tree. In exchange for this improved overall performance, Javolution's map is extremely jittery, at times taking an order of magnitude longer than Java or Canteen to complete the same operation. It provides no guarantee on predictability because an unlucky distribution of hash coding could result in degraded performance. In other words, the performance depends heavily not on the characteristics of the algorithm but on the data supplied to it. Such behavior is extremely difficult to analyze for hard real-time and safety-critical requirements.

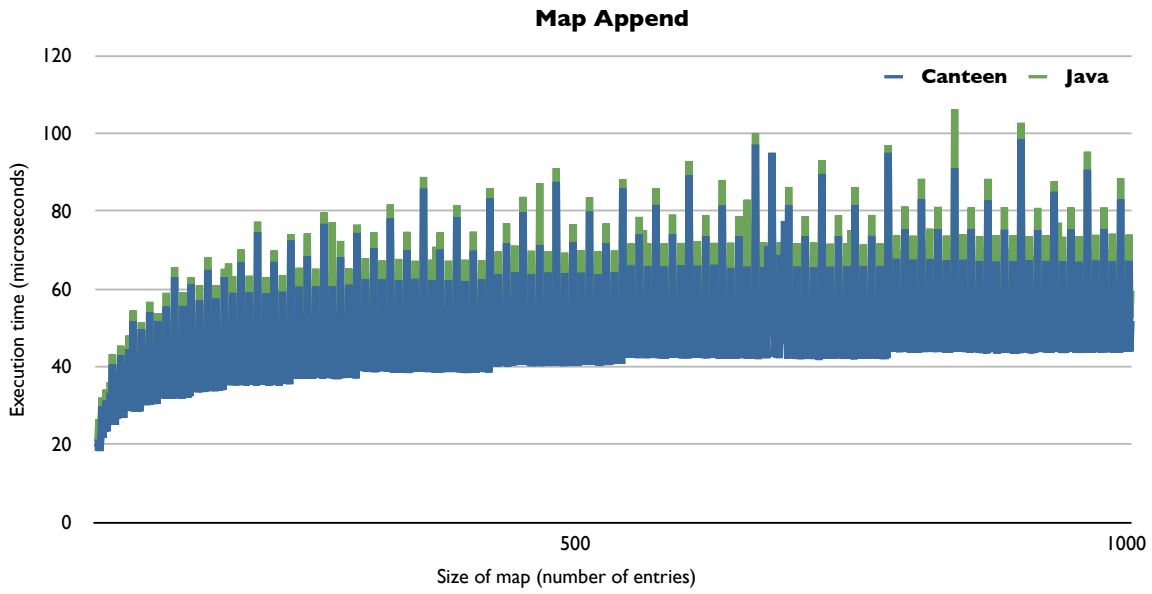

Figure 7.10: These measurements compare the predictability of the append operation for the tree-based map classes in Canteen and Java. (Javolution's measurements are in Figure 7.11.) Each measurement represents the time taken for an append when the map contains the given number of entries.

Javolution also depends on RTSJ's scoped memory management to eliminate the need for a garbage collector. While the absence of a collector should improve overall predictability, there is still no guarantee on the worst-case execution time of Javolution's operations. There are no existing algorithms for static WCET analysis in the presence of scoped memory.

In summary, Java has the highest throughput overall, while Javolution has the highest variance. Canteen is by far the most predictable of the three libraries while still maintaining, and sometimes exceeding, the performance of the others. It is the best solution for obtaining tight estimates on worst-case execution time.

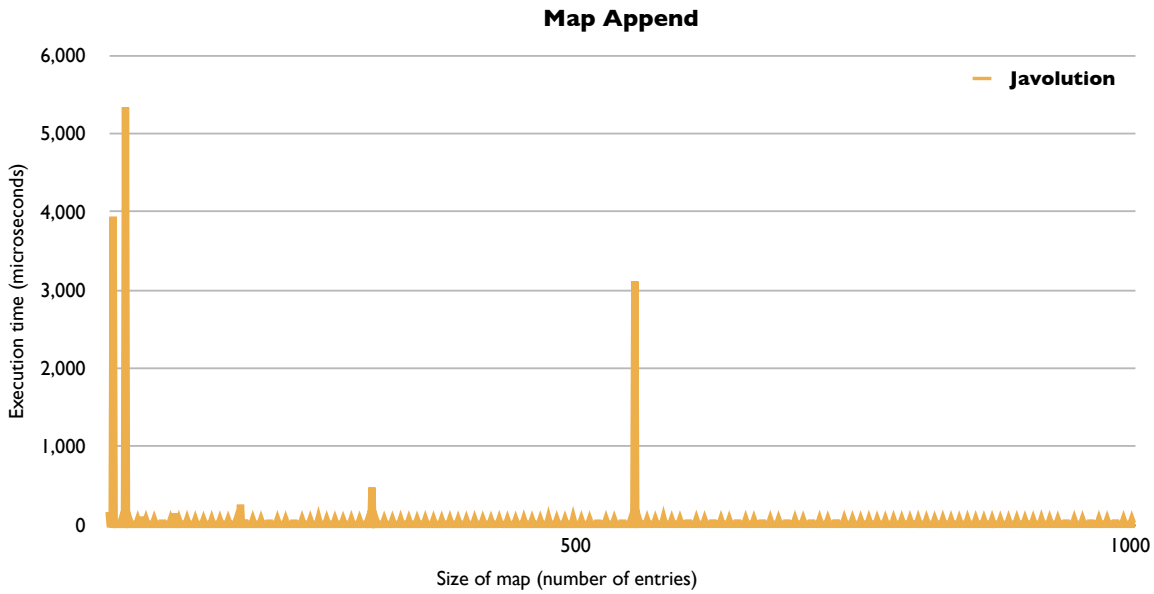

Figure 7.11: These measurements show the predictability of the append operation for the map class in Javolution. (It is presented separately from Figure 7.10's Canteen and Java measurements because of the substantially different results.) Each measurement represents the time taken for an append when the map contains the given number of entries.

7.9.3 Heap Allocation Analysis

In addition to predictable performance, a library for hard real-time systems should also have predictable memory consumption. The safety net of virtual memory cannot be assumed, as discussed in Section 7.6, and therefore a bound must be placed on the amount of physical memory required by the application.

Luckily, the memory pool technique in Canteen (see Section 7.8.1) creates as a side effect a natural bound on memory consumption: The amount of heap memory allocated after initialization is equal to the maximum amount of heap memory that will ever be required by Canteen. Other libraries, including Java and Javolution, offer no such guarantee and are therefore inappropriate for hard real-time and safety-critical systems.

To support these claims, a simple experiment was conducted to examine the mem-

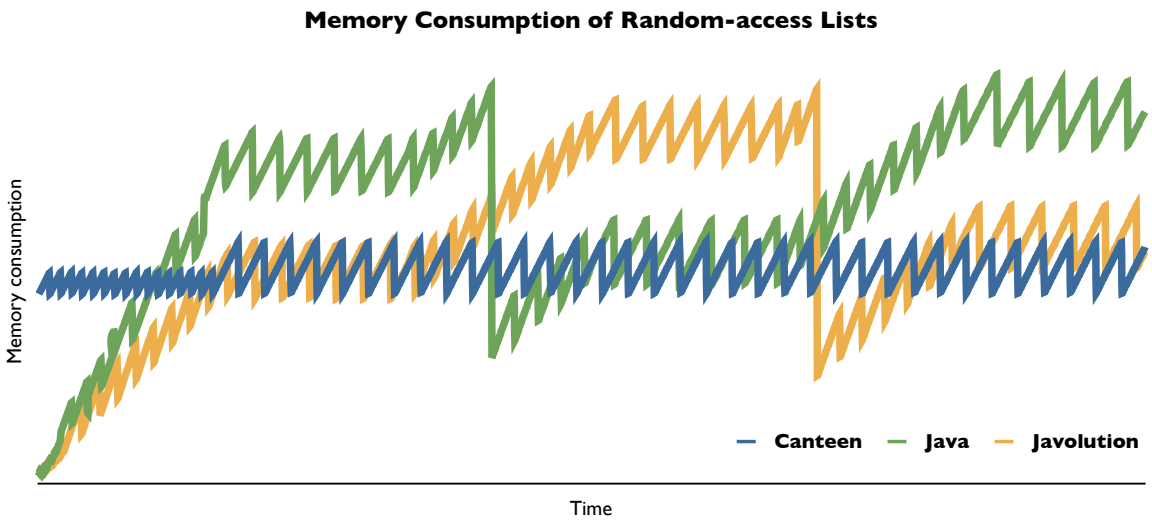

Figure 7.12: These measurements show how memory consumption varies as a random-access list is modified in each of the three libraries. Java and Javolution are erratic, while Canteen's memory usage is predictable and bounded.

ory allocation behavior of the three libraries while running on the Java Standard Edition 1.6.0 virtual machine. The random-access list collection was used as a representative sample. In the experiment, 500,000 elements are added to the list and then removed one by one. This cycle is repeated three times. To collect memory statistics, the MemoryMXBean facility was invoked after each change to the list. To isolate the libraries from each other, the VM was restarted on each run. (The complete source code for this benchmark program is provided in the Volta distribution.)

Figure 7.12 shows the results of this experiment. The most visible characteristic of the measurements is the high-frequency jitter in each line. This sawtooth pattern is merely an artifact of collecting and recording memory statistics during the run. Short-term garbage is created (and then quickly collected) as a byproduct of this process.

The larger variations in the graph are the more important observation. The Java and Javolution benchmarks exhibit large swings in memory usage over time. Surprisingly, the usage increases even when elements are being removed from the list. Both Java

and Javolution fail to reach a steady state in memory consumption even though the same addition and removal cycle is repeated three times. In contrast, Canteen is much more predictable; its memory usage pattern is nearly constant.

7.10 Restrictions of an Analyzable Library

The benefits provided by Canteen are not without cost. It introduces subtle but serious conflicts that arise when memory pools are used in object-oriented environments such as Java. This section summarizes the fundamental issues and major challenges in designing a library specifically for hard real-time and safety-critical systems.

7.10.1 Memory Pool Restrictions

Without dynamic creation of objects, Java becomes a restricted subset of itself. Recent work on real-time garbage collection tries to relax this restriction [257, 258, 259], but the allocation and deallocation rate must still be known so that garbage collection can be scheduled. The memory pool approach of Canteen, provided as a substitute for garbage collection, introduces special problems for object-oriented libraries, particularly in the areas of iterators, object mutation, and element replacement.

Iterators

As required by most safety-critical software specifications, Canteen never allocates memory from the heap after initialization, which greatly simplifies analysis and validation. Instead, it pre-allocates all elements in a memory pool during the start-up phase, up to a user-defined maximum size. When the user needs to add a new ele-

ment to a collection, it can be retrieved from the pool in constant time. When the element is removed, it is recycled back into the pool, again in constant time. (Note that array-based collections may require an additional linear time algorithm in order to perform the insertion or deletion of the element within the array. However, this time can be bounded through static WCET analysis.)

In addition to the elements of a collection, the collection's iterator objects cannot be dynamically allocated; they must be pre-allocated. This presents a problem because, with current analysis tools, the number of iterators to pre-allocate is unknown. Simply disallowing iterators would not be appropriate, given that the iterator design pattern is so prevalent in Java.

Canteen works around this problem by pre-allocating a single iterator per collection object and returning this object whenever an iterator is requested. The shared iterator is reset to its initial state on each request. Although this approach solves the problem in the majority of usage patterns (assuming single-threaded execution), there may be corner cases where an iterator is requested while it is already in use. When this happens, the state of the iterator will be unexpectedly reset, causing run-time errors that are difficult to detect and prevent.

Object Mutation

Another complication of memory pools is that all elements stored in a collection must be mutable. If the elements are immutable, their state will never change, having been allocated and initialized to a default state during the library's initialization phase. As a consequence, none of Java's standard data type wrappers, such as Integer, Float, etc., can be used in Canteen. Users must instead write their own data type wrappers whose state can be changed. Users must also be careful to reset this state when retrieving

an element from a memory pool. (Unlike objects allocated with the new operator, an object in Canteen may have been recycled from a previous use and its fields might not be zero or null.)

Furthermore, objects obtained from a memory pool must always be added back into the collection from which they came. If the client mistakenly adds an element to some other collection, memory leaks will occur. Users must also be careful not to access an object once it has been removed from a collection. It will be recycled back into the memory pool and could therefore be in use by some other part of the code, leading to data corruption through concurrent access.

Element Replacement

Yet another issue in memory pool management is element replacement. The List and Map interfaces allow the user to replace an existing element with some other element. The methods for performing this replacement—get for lists and put for maps—are potentially dangerous in the presence of memory pools. The root of the problem is that replacement elements cannot be created at run-time due to memory allocation restrictions. Instead, the replacement element must come from a memory pool, and pools contain only enough objects for the maximum declared capacity of their corresponding container.

Replacing elements under these circumstances may cause memory leaks and null pointer dereferencing. For example, consider a list containing a single element and a maximum size of two. A user might request an object from the memory pool in order to replace the existing element. If this happens, there will be no more free objects in the memory pool, even though the user may assume that another object remains, given that the list has not yet reached its maximum size.

There are various potential solutions to this problem, although none is satisfactory. One possibility is to perform a static analysis on the code to predict how many times a replacement method is called, then pre-allocate an object in the memory pool for each call. This approach could be very wasteful of memory, however. Another solution is to disallow element replacement altogether, but there are legitimate reasons for replacement, such as swapping two elements in a list. Also, the Map.put method is used not only for replacement; it is also the primary mechanism for adding new elements.

Canteen relies on none of these solutions. Instead, it is based on the observation that element replacement is usually necessary only because the Java objects placed in a container are typically immutable (e.g., Integer, Float, etc.). Since the requirements of our library permit only mutable objects, the need for element replacement is substantially reduced. Therefore, the library simply requires that an object passed to the set method must have been obtained from the list itself and not its memory pool. Likewise, the put method must only be used for swapping elements or adding new ones. Additional support for element replacement is relegated to future work.

7.10.2 Exception Handling Compromises

As discussed in Section 7.6, the Canteen library explicitly avoids exception handling in order to facilitate program verification and timing analysis. During the initialization phase, however, which does not require timing analysis, the container classes are free to throw exceptions. Also, the classes may inadvertently throw unchecked exceptions, such as a null-pointer exception an or array index out-of-bounds exception, if they are initialized or invoked incorrectly.

In certain situations, the lack of exception handling in Canteen means that error codes

must be used (e.g., null object values or negative integer values). Substituting error codes for structured exception handling is error-prone, since the user could forget to check a return value for an error. Worse, errors may sometimes have to be suppressed entirely in order to maintain compatibility with existing collection interfaces. For example, the add(int,E) method in the List interface returns void, relying on thrown exceptions to report errors. Therefore, users of Canteen have no way to check for an error when invoking this method.

A pragmatic but insufficient solution is to pre-allocate the data structures for all possible exceptions in immortal memory and reuse them. A better approach is to avoid exceptions by a defensive programming style. The absence of runtime exceptions can then be proven formally [260].

7.10.3 Unimplemented Methods

In the Java collections API, certain interface methods are declared "optional." For example, a collection that implements an immutable list of objects has no reason to implement the List.set operation. Such optional operations are expected to throw UnsupportedOperationException.

In Canteen, all optional operations are implemented with three exceptions: Collection.addAll, Map.putAll, and Collection.containsAll. The add operations were excluded because adding elements from some arbitrary collection, instead of the collection's own memory pool, could cause memory leaks. Also, the maximum size of the operation's collection parameter must be known for WCET analysis, but current tools cannot detect loop bounds for a polymorphic collection object. The containsAll operation could not be implemented for this reason, as well.

In addition, two *mandatory* operations could not be implemented in Canteen due to the restrictions of analyzability. Specifically, the toArray and subList methods would require allocating memory after initialization, a prohibited act in Canteen. (This restriction could be bypassed by pre-allocating return values for the operations, but this would greatly increase memory requirements, and it could cause dangerously unpredictable results if clients modify the shared return value.)

Chapter 8

Examples of Interactive WCET Analysis

The previous chapters have concentrated on the theory of interactive WCET analysis, largely avoiding a proper discussion of the practical implications. This chapter changes course by focusing exclusively on how the presented ideas can help solve real-world problems. The sections that follow examine two typical scenarios that developers may face when creating real-time system software: selecting a hash function and polling a sensor.

8.1 Hash Functions

A hash function transforms a large piece of data into a smaller representative sample. It is most common in a hash table, where the hash of a search key corresponds to an index. It may also be used to compute a hash sum or, as it is more popularly known, a *checksum*. Checksums detect accidental errors that may occur when transmitting

or storing an arbitrary block of digital data. If the previously saved checksum does not match the recomputed one, data corruption has likely occurred.

Figure 6.3 shows a Java implementation of the simplest possible checksum function. It merely iterates through every byte of the array, adds its value to a running total, and returns the result. The implementation also includes a loop bound annotation so that Clepsydra may analyze it for WCET.

This type of hash function is easy to implement, but it may fail to detect errors that affect many bits at once, such as a change in their order. To address this weakness, a hash may instead be computed with a more complex function called a *cyclic redundancy check*, or CRC. It is especially adept at detecting errors due to noise in a transmission channel. An implementation example of a 16-bit CRC can be found in Figure 6.3 as well.

Naturally, the more complex CRC function would require more execution time than a simple summation, but precisely how much more is not obvious. This uncertainty is a problem because the performance of the hash algorithm has a direct impact on overall throughput. Consider, for example, a communications device that streams data to another device. It would likely implement some kind of hash function in its protocol in order to handle transmission errors. Now imagine that the design specifications required this device to transmit a 64-kilobyte block of data every 50 milliseconds. Which hash function should be used?

With Clepsydra, the answer can be found directly. First, assume that the developer of the communications device has written the code shown in Figure 6.3. Finding the execution time of both hash functions for this particular design would simply require changing the MAX_LENGTH constant to 65,536 (the number of bytes in 64 kilobytes). The Clepsydra plug-in would then re-annotate each line of the code, revealing im-

mediately that the worst-case time of the summation checksum is 26 milliseconds, while the CRC's is 275 milliseconds. (These times assume a 100 MHz JOP as the target processor.) The latter option would then have to be rejected because it exceeds the 50-millisecond deadline. Note that there was no need to run the code or even switch to a separate analysis tool. The decision was made without ever leaving the programming environment.

As a simple test to verify these results, a small utility was created to measure the speed at which the two hash functions execute on a physical processor. Figure 8.1 shows a screenshot of this utility while monitoring a 100 MHz JOP, which was executing the checksum method in an infinite loop. The dial on the left-hand side shows the instantaneous period of each cycle of the loop, while the graph on the right shows the period over time. As expected, the period is about 27 milliseconds, plus an overhead of about 6 milliseconds to account for the communication latency between the JOP and the test machine. The red line in the graph marks the declared deadline, and the measured period is safely underneath it.

In contrast, Figure 8.2 takes the same measurements on the same processor, this time running the CRC algorithm. The period is about 275 milliseconds plus another 6 milliseconds or so due to the communication latency. As predicted, the period exceeds the 50-millisecond deadline, making CRC unusable for this particular application. Again, these measurements are actually unnecessary during a usual development cycle; they merely serve as evidence that Clepsydra correctly predicts worst-case execution time.

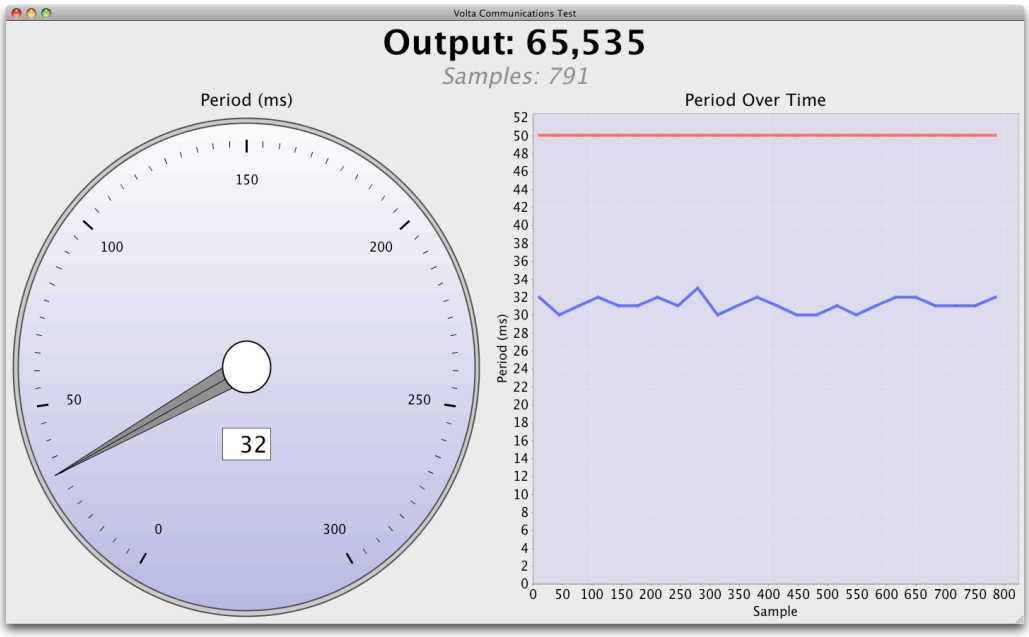

Figure 8.1: This screenshot shows a test program measuring the period between successive invocations of a checksum algorithm running on a physical Java microprocessor.

8.2 Sensor Polling

An extremely common task in the development of a real-time system—and one that is also extremely troublesome and difficult—is to determine how fast the system can interact with its environment. Sensors must be polled, and actuators must be triggered, and the rate at which these actions take place determines whether the system meets its design criteria. For example, a burglar alarm system might be required to monitor a power circuit and switch to a backup battery within 50 milliseconds if the primary voltage drops below a certain threshold. Performing a WCET analysis is the only way to guarantee that the implementation will meet this design requirement.

Even when deadlines are not crucial, finding the WCET may still be necessary when creating a system that interacts with sensors. The developer may need to know the maximum operating frequency of the system, which requires knowing the highest rate at which a sensor can be polled. Finding the answer would traditionally require

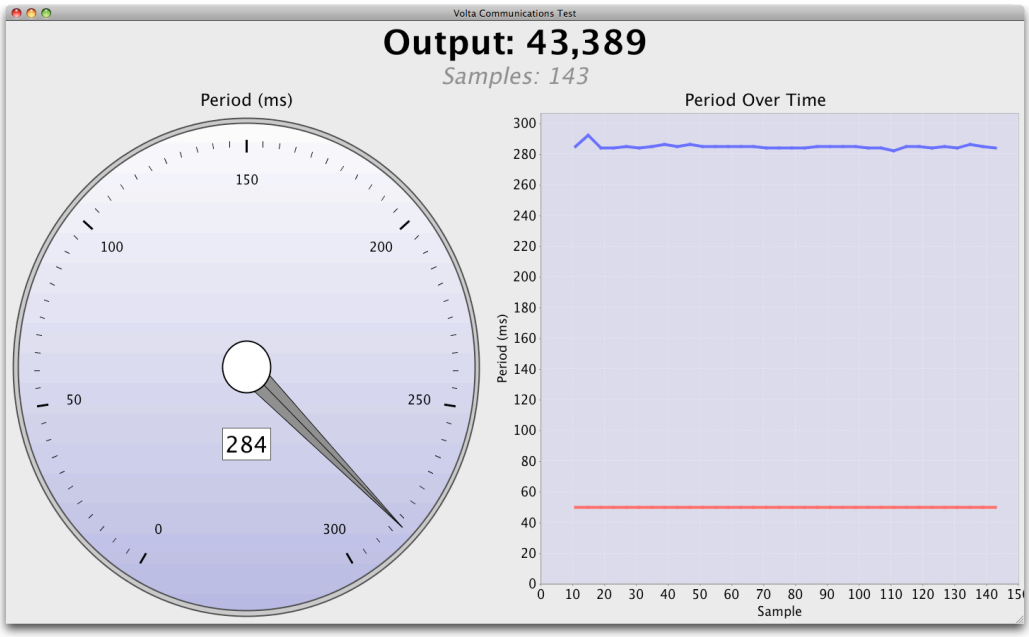

Figure 8.2: This screenshot shows a test program measuring the period between successive invocations of a CRC algorithm running on a physical Java microprocessor.

error-prone and time-consuming system tests and measurements, but an interactive WCET tool such as Clepsydra can help make a guaranteed determination as soon as the code has been written. It can also reevaluate the decision almost instantly, on each change of the code.

As an example, consider the case of a rotary encoder. This type of sensor converts the angular position of a shaft into a digital code. It is an invaluable device for robotics, industrial machines, and any application that needs to know the rotational displacement of an axle. One specific case is a single-axis gimbal that allows an object to rotate up and down along a horizontal axis. A mobile robot may have a LIDAR sensor or similar range finder placed on this gimbal, and as the sensor moves up and down, it takes continuous samples, many times each second, building up a 3D representation of the environment that the robot can then use for obstacle detection and avoidance.

A key property in such a system is the frequency at which the encoder can be polled.

If it is polled too slowly, the data streaming from the range finder will not match the observed angle of its axis. The maximum operating frequency must somehow be determined in order to guarantee that the sensor will be polled at a sufficient rate. Otherwise, the robot's 3D model of the world will become corrupted.

Figure 8.3 shows an example of how Clepsydra can provide an immediate answer in this type of situation. The source code in the screenshot shows how an encoder value might be processed by adding it to a first-in-first-out buffer. If the buffer is full, the oldest element is removed to make space for a new one. (In practice, a circular buffer would probably be used instead for greater efficiency, but this naïve implementation keeps the example simple.)

The screenshot shows two separate implementations of this procedure, one using an array-based list and another using a linked list. (Both lists are imported from the Canteen library to provide predictability and analyzability.) At first, one might expect the array-based implementation to outperform the linked list. Adding an element to a pre-allocated array only requires copying the element and incrementing a size variable, while adding to a linked list requires several pointer copy operations. The WCET annotations inserted by Clepsydra show that this is indeed the case: Adding the encoder position to an array list (line 33) takes 3.06 microseconds in the worst case, but adding it to a linked list (line 44) takes 3.68 microseconds.

Observe, however, that the WCET of each method as a whole (lines 26 and 37) gives the opposite result. At 67.5 milliseconds, the array-based implementation takes more than twice as long to execute in the worst case. The reversal is due to the remove invocations on lines 31 and 42. Removing an element from the beginning of an array requires shifting each of the remaining elements down by one position, whereas a linked list only needs to adjust a few pointers. The analysis tool has thus shown that the maximum operating frequency of the system cannot be obtained with the

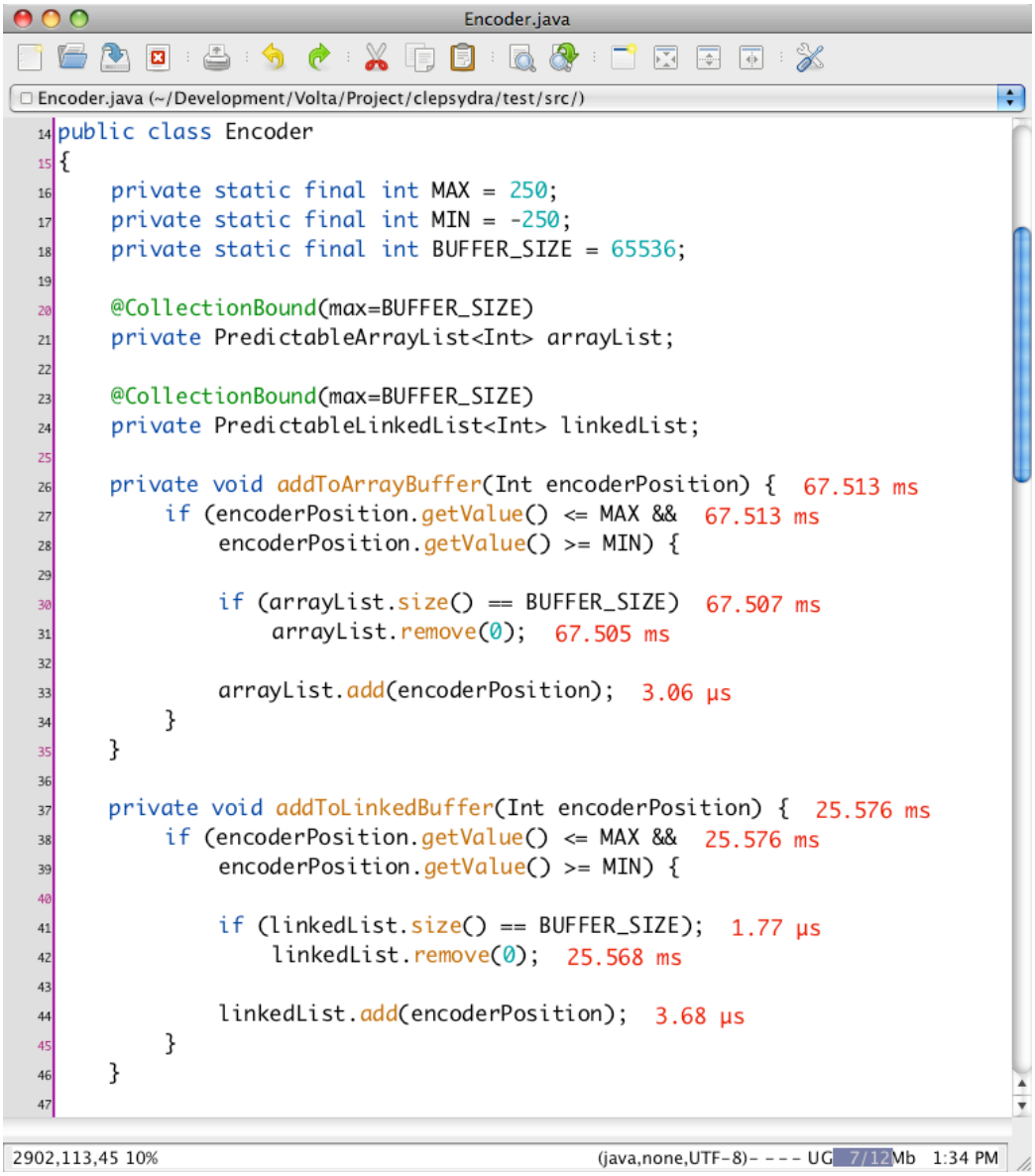

Figure 8.3: This screenshot shows Clepsydra interactively analyzing two implementations of a buffer handling method. The WCET of each implementation, indicated by the red text annotating each line, reveals which version offers better performance.

array-based method, and the linked list version should be used instead.

As a simple verification of these results, the same measurement program introduced in Section 8.1 can be applied once again. Figure 8.4 shows the program measuring the time a Java processor requires to poll a rotary encoder for its position and store the value into a linked list buffer. The average period is about 40 milliseconds:

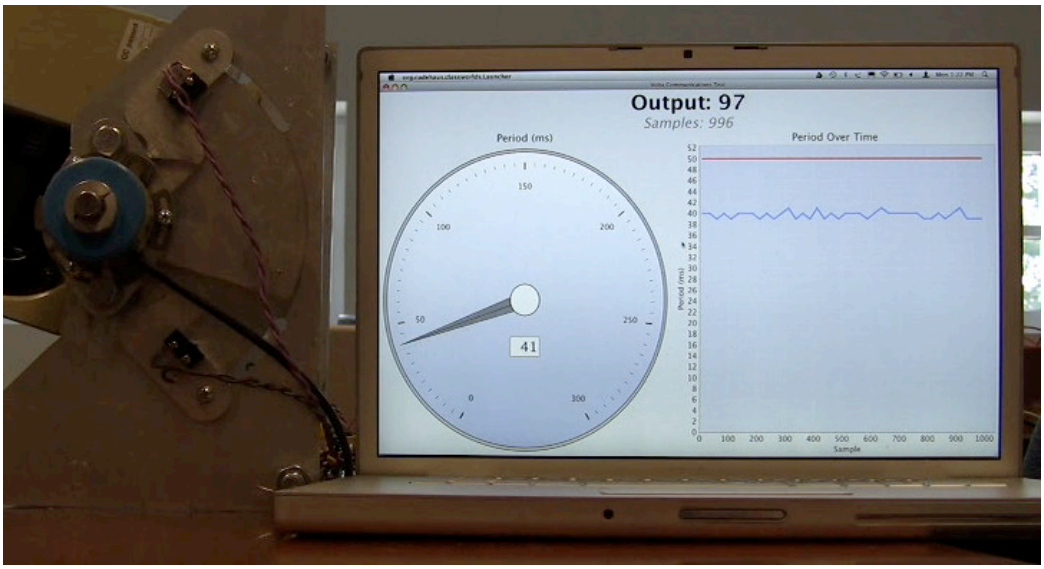

Figure 8.4: A test program measures the period at which a rotary encoder, shown here connected to a gimbal on the left-hand side, can be polled using linked list buffer handling.

25 milliseconds for the buffer processing and another 15 to account for reading the encoder and transmitting the value over a communication link. The total time is within an arbitrary 50-millisecond deadline as denoted by the red line in the graph.

Figure 8.5 shows precisely the same configuration but with an array-based buffer instead of a linked list. Here, the average period is about 82 milliseconds: 67 milliseconds for the buffer processing and, as before, another 15 to account for communication latency. This time the period exceeds the 50-millisecond deadline, as predicted by Clepsydra.

As in Section 8.1, a key point to remember is that the knowledge of which implementation exceeds the deadline can be obtained without performing any live tests. The WCET analyzer can make this determination statically. More importantly, the analyzer employed in this example—Clepsydra—is interactive, so it can adjust continuously for changes in the code. It can pinpoint exactly when an addition, no matter how small, causes the deadline to be exceeded, thereby helping the developer

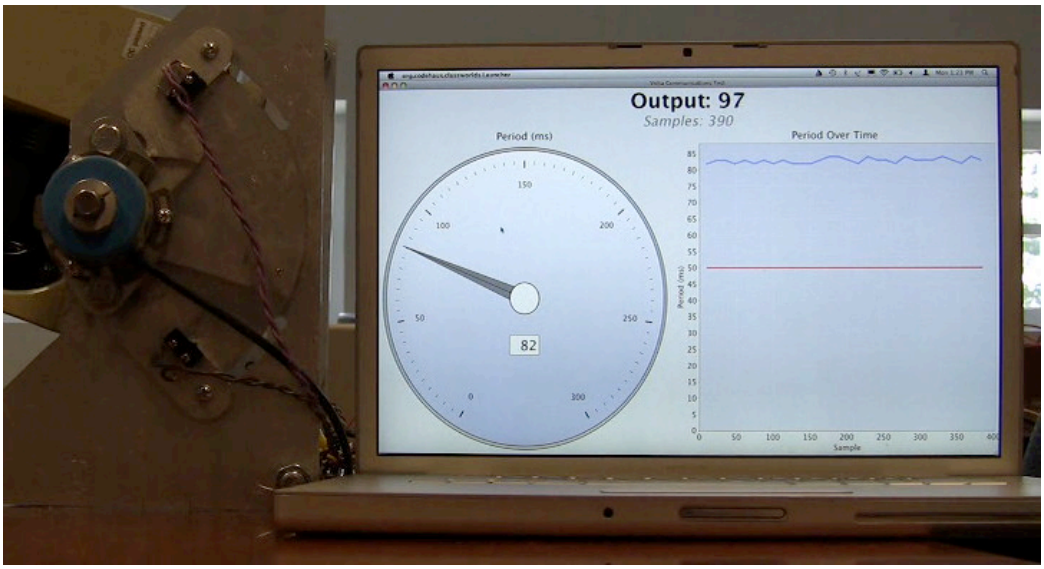

Figure 8.5: A test program measures the period at which a rotary encoder, shown here connected to a gimbal on the left-hand side, can be polled using array-based buffer handling.

fine-tune the code for optimal WCET.

Chapter 9

Conclusions and Future Work

When private spaceflight company SpaceX launched its Falcon 1 rocket in August 2008, the brand-new engine design produced residual thrust for 1.5 seconds longer than expected. The delay resulted in a collision between the first and second stages of the rocket, causing it to plummet into the Pacific Ocean.

Timing failures like the one that downed the Falcon 1 will likely become more frequent as hard real-time and safety-critical systems continue to move beyond the realm of aerospace. Validation of worst-case execution time is therefore a necessity. Without such analysis, the behavior of the system cannot be guaranteed. It could fail catastrophically or, in the case of safety-critical hard real-time systems, could cause injury or even death. While WCET analysis does not prevent failure, it does ensure that the system's deadlines will be met.

Despite this fact, convincing practitioners in the field of the importance of WCET analysis has been enormously difficult. One explanation is that the analysis techniques currently available are too slow, and tool implementations are usually based on C, leading to an overabundance of complexity compared to tools based on more modern,

higher-level languages. As Aristophenes would say: "High thoughts must have a high language."[1]

Chapter 3 argued that WCET analysis based on bytecode languages such as Java would be more attractive to industry. Arbitrary code is extremely difficult to analyze, but the restrictions of Java make the job more tractable while also offering the end user higher productivity and stronger safety guarantees. These reasons have already prompted educators in the real-time systems field to call for an increased use of Java [261].

At the same time, Java brings a number of new complications to the WCET analysis problem. The additional layer of the virtual machine, combined with garbage collection and polymorphism, tends to negate the inherent advantages of Java. However, the advent of Java-specific processors offers a solution by eliminating many sources of unpredictability in both the hardware and software layers. Switching to a fundamentally different architecture may seem drastic, but the idea is not unorthodox. Many researchers have argued in favor of custom hardware for real-time computing, such as general-purpose CPUs with WCET-friendly designs [47, 209, 262].

Some interesting possibilities arose from these assumptions. The first observation is that Java microprocessors make low-level WCET analysis of basic blocks almost trivial, meaning that the largest source of pessimism comes from the longest-path search. However, the dominant search technique, IPET, becomes quite slow as program complexity grows. Chapter 6 demonstrated that the much faster tree-based technique can, with some additional design and implementation effort, attain the same accuracy as IPET even in the presence of method invocations.

This combination of Java, Java microprocessors, and better tree-based analysis en-

[1] This is not a direct quote but a popular paraphrasing of a line from *The Frogs* by Aristophenes.

ables a new paradigm known as *interactive* analysis. It allows tight integration of WCET knowledge throughout the development cycle, providing near-immediate feedback of timing information as the code is written. This approach can help eliminate timing bugs and other bad surprises before they have a chance to infect later phases of the development cycle. Tools supporting this kind of interactivity should therefore be made an integral part of real-time system programming.

As a first step toward this goal, one of the key contributions of this dissertation is Volta, a suite of tools and libraries to enable interactive WCET analysis. It is freely available, open source, and makes the claims verifiable. It supports the ideas of interactive analysis with a true implementation, not just a theory.

Of course, the Volta tools are still in a prototype stage and are by no means complete. They have many known issues and limitations, including a lack of support for recursion, polymorphism, and exception handling. The project is ripe for future work, especially in the area of real-time garbage collection. Volta is ignorant of dynamic memory, but ongoing research on the problem is showing promise [75, 79, 263, 259]. Tools such as Cascade, Clepsydra, and Canteen could serve as a basis for future efforts in this area.

Despite these limitations, the potential benefits are clear. The groundwork has been laid, and perhaps one day an interactive WCET analysis tool will be part of every real-time system developer's toolbox.

Bibliography

[1] J. Ubois, "Sun goes Hollywood," *SunWorld*, November 1995. Available: http://sunsite.uakom.sk/sunworldonline/swol-11-1995/swol-11-pixar.html

[2] M. G. Gouda, Y.-W. Han, E. D. Jensen, W. D. Johnson, and R. Y. Kain, *Distributed Data Processing Technology*. Honeywell Systems and Research Center, September 1977, vol. 4, ch. 3.

[3] F. F.-H. Nah, "A study on tolerable waiting time: How long are Web users willing to wait?" *Behaviour and Information Technology*, vol. 23, no. 3, pp. 153–163, May 2004.

[4] P. C. Dibble, *Real-Time Java Platform Programming*. Prentice Hall, March 2002.

[5] R. Steinmetz and K. Nahrstedt, *Multimedia: Computing, Communications and Applications*. Prentice Hall, 1995.

[6] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[7] *IEEE 1588: Precision clock synchronization protocol for networked measurement and control systems*. IEEE, September 2004.

[8] C. Jones and M. J. Matarić, "Automatic synthesis of communication-based coordinated multi-robot systems," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004)*, vol. 1, September 2004, pp. 381–387.

[9] J. Kaiser and M. A. Livani, "Invocation of real-time objects in a CAN bus-system," in *Proceedings of the 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 1998)*, April 1998, pp. 298–307.

[10] H. Kopetz and G. Grünsteidl, "TTP – a protocol for fault-tolerant real-time systems," *Computer*, vol. 27, no. 1, pp. 14–23, January 1994.

[11] H. Kopetz, "A comparison of TTP/C and FlexRay," Institut für Technische Informatik, Technische Universität Wien, Austria, Tech. Rep., May 2001.

[12] *Common Object Request Broker Architecture: Core Specification*, 3rd ed. Object Management Group, March 2004.

[13] *Real-Time CORBA Specification*, 2nd ed. Object Management Group, January 2005.

[14] D. C. Schmidt and F. Kuhns, "An overview of the Real-Time CORBA specification," *Computer*, vol. 33, no. 6, pp. 56–63, June 2000.

[15] G. Pardo-Castellote, "OMG data-distribution service: Architectural overview," in *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCS 2003)*, May 2003.

[16] E. S. Raymond, *The Art of UNIX Programming*. Addison-Wesley Professional, October 2003.

[17] T. F. Lawrence, "Quality of service (QoS): a model for information," in *Proceedings of the 4th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 1999)*, January 1999, pp. 180–183.

[18] T. V. Vleck, "Software engineering proverbs." Available: http://www.multicians.org/thvv/proverbs.html

[19] B. Rieder, I. Wenzel, K. Steinhammer, and P. Puschner, "Using a runtime measurement device with measurement-based WCET analysis," in *Proceedings of the 2007 International Embedded Systems Symposium (IESS 2007)*, June 2007, pp. 15–26.

[20] J. Hu, S. Gorappa, J. A. Colmenares, and R. Klefstad, "Compadres: A lightweight real-time Java component middleware framework for composing distributed, real-time, embedded systems," in *Middleware 2007*, ser. Lecture Notes in Computer Science, vol. 4834. Springer Berlin, November 2007, pp. 41–59.

[21] S. Bourne, "A conversation with Bruce Lindsay," *Queue*, vol. 2, no. 8, pp. 22–33, November 2004.

[22] J. Gray, "Why do computers stop and what can be done about it?" Tandem Computers, Tech. Rep. 85.7, June 1985.

[23] M. Grottke and K. S. Trivedi, "Software faults, software aging and software rejuvenation," *Journal of the Reliability Engineering Association of Japan*, vol. 27, no. 7, pp. 425–438, 2005.

[24] E. Kligerman and A. D. Stoyenko, "Real-time Euclid: a language for reliable real-time systems," *IEEE Transactions on Software Engineering*, vol. 12, no. 9, pp. 941–949, September 1986.

[25] A. Mohammadi and S. G. Akl, "Scheduling algorithms for real-time systems," Queen's University, Kingston, Ontario, Canada, Tech. Rep. 2005-499, July 2005.

[26] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[27] "Predictable performance for dynamic load and overload: A technology breakthrough," TimeSys Corporation, Tech. Rep., 2004.

[28] K. K. Kim, "Object structures for real-time systems and simulators," *Computer*, vol. 30, no. 8, pp. 62–70, August 1997.

[29] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull, *The Real-Time Specification for Java*, G. Bollella, Ed. Addison Wesley Longman, January 2000.

[30] H. Ramaprasad and F. Mueller, "Bounding worst-case data cache behavior by analytically deriving cache reference patterns," in *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium (RTAS 2005)*. Washington, DC, USA: IEEE Computer Society, March 2005, pp. 148–157.

[31] H. Ramaprasad and F. Mueller, "Bounding preemption delay within data cache reference patterns for real-time tasks," in *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)*. Los Alamitos, CA, USA: IEEE Computer Society, April 2006, pp. 71–80.

[32] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, "Reliable and precise WCET determination for a real-life processor," in *Proceedings of the First International Workshop on Embedded Software (EMSOFT 2001)*, ser. Lecture Notes in Computer Science, vol. 2211. London, UK: Springer-Verlag, October 2001, pp. 469–485.

[33] Y.-T. S. Li, S. Malik, and A. Wolfe, "Performance estimation of embedded software with instruction cache modeling," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, no. 3, pp. 257–279, July 1999.

[34] C. Ferdinand, R. Heckmann, and H. Theiling, "Convenient user annotations for a WCET tool," in *Proceedings of the Third International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*, July 2003, pp. 17–20.

[35] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper, "Applying static WCET analysis to automotive communication software," in *Proceedings of the Seventeenth Euromicro Conference on Real-Time Systems (ECRTS 2005)*. Washington, DC, USA: IEEE Computer Society, July 2005, pp. 249–258.

[36] L. Tan, "The worst case execution time tool challenge 2006: The external test," in *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2006)*, November 2006, pp. 241–248.

[37] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, 2nd ed. Microsoft Press, June 2004.

[38] "DO-178B, software considerations in airborne systems and equipment certification," RTCA, Inc., December 1992.

[39] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.

[40] M. Bordin and T. Vardanega, "Correctness by construction for high-integrity real-time systems: A metamodel-driven approach," in *Proceedings of the Twelfth Ada-Europe International Conference on Reliable Software Technologies*, ser. Lecture Notes in Computer Science, vol. 4498. Springer Berlin, June 2007.

[41] P. Amey, "Correctness by construction: Better can also be cheaper," *CrossTalk: The Journal of Defense Software Engineering*, vol. 15, no. 3, pp. 24–28, March 2002.

[42] J. Gustafsson and A. Ermedahl, "Experiences from applying WCET analysis in industrial settings," in *Proceedings of the Tenth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2007)*. Washington, DC, USA: IEEE Computer Society, May 2007, pp. 382–392.

[43] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, January 2003.

[44] E. A. Lee and Y. Zhao, "Reinventing computing for real time," in *Proceedings of the Monterey Workshop 2006*, ser. Lecture Notes in Computer Science, vol. 4322. Springer Berlin / Heidelberg, September 2006, pp. 1–25.

[45] P. P. Puschner and A. V. Schedl, "Computing maximum task execution times — a graph-based approach," *Real-Time Systems*, vol. 13, no. 1, pp. 67–91, 1997.

[46] R. Kirner and P. Puschner, "Discussion of misconceptions about WCET analysis," in *Proceedings of the Third International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*, July 2003, pp. 61–64.

[47] P. Puschner, "Is worst-case execution-time analysis a non-problem? — Towards new software and hardware architectures," in *Proceedings of the Second International Workshop on Worst-Case Execution Time Analysis (WCET 2002)*, June 2002, pp. 57–60.

[48] P. Puschner, "Experiments with WCET-oriented programming and the single-path architecture," in *Proceedings of the Tenth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2005)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 205–210.

[49] S. Petersson, A. Ermedahl, A. Pettersson, D. Sundmark, and N. Holsti, "Using a WCET analysis tool in real-time systems education," in *Proceedings of the Fifth International Workshop on Worst-Case Execution Time Analysis (WCET 2005)*, R. Wilhelm, Ed., Dagstuhl, Germany, 2005.

[50] G. Cato, "Commentary: Java is ready for real time," *EE Times*, December 2006.

[51] D. Geer, "The Unmanned Little Bird project," *SERVO Magazine*, pp. 10–12, February 2007.

[52] J. W. Grenning, "Why are you still using C?" *Embedded.com*, April 2003.

[53] D. M. Ritchie, "The development of the C language," *SIGPLAN Notices*, vol. 28, no. 3, pp. 201–208, March 1993.

[54] D. Engler, "Weird things that surprise academics trying to commercialize a static checking tool," 2005. Available: http://www.stanford.edu/~engler/spin05-coverity.pdf

[55] B. Meyer, "Approaches to portability," *Journal of Object-Oriented Programming*, vol. 11, no. 6, pp. 93–95, 1998.

[56] D. C. Schmidt, D. L. Levine, and S. Mungee, "The design of the TAO real-time object request broker," *Computer Communications*, vol. 21, no. 4, pp. 294–324, April 1998.

[57] B. Stroustrup, "Bjarne Stroustrup's FAQ." Available: http://www.research.att.com/~bs/bs_faq.html

[58] E. G. Benowitz and A. F. Niessner, "Experiences in adopting real-time Java for flight-like software," in *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, ser. Lecture Notes in Computer Science, vol. 2889. Springer Berlin, October 2003, pp. 490–496.

[59] C. A. R. Hoare, "The emperor's old clothes," *Communications of the ACM*, vol. 24, no. 2, pp. 75–83, February 1981.

[60] D. Lammers, "REAL-TIME JAVA: Reliability quest fuels RT Java projects," *EE Times*, March 2005. Available: http://www.eetimes.com/showArticle.jhtml?articleID=159905424

[61] M. R. Elliott, "The real-time specification for Java," Presentation slides, June 2007. Available: http://www.uuasc.org/rtsj.pdf

[62] Y. Chen, R. Dios, A. Mili, L. Wu, and K. Wang, "An empirical study of programming language trends," *IEEE Software*, vol. 22, no. 3, pp. 72–79, May/June 2005.

[63] A. T. Murray and M. Shahabuddin, "OO techniques applied to a real-time, embedded, spaceborne application," in *Companion to the Twenty-First ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. New York, NY, USA: ACM, October 2006, pp. 830–838.

[64] G. Bernat, A. Burns, and A. Wellings, "Portable worst-case execution time analysis using Java byte code," in *Proceedings of the 12th Euromicro Conference on Real-Time Systems (Euromicro-RTS 2000)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2000, pp. 81–88.

[65] B. Boyes, "Why use Java?" Systronix White Paper. Available: http://www.practicalembeddedjava.com/language/WhyUseJava.pdf

[66] E. Y.-S. Hu, G. Bernat, and A. Wellings, "Addressing dynamic dispatching issues in WCET analysis for object-oriented hard real-time systems," in *Proceedings of the Fifth IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2002)*. Los Alamitos, CA, USA: IEEE Computer Society, April 2002, pp. 109–116.

[67] M. A. Wehrmeister, C. E. Pereira, and L. B. Becker, "Optimizing the generation of object-oriented real-time embedded applications based on the real-time specification for Java," in *Proceedings of the Seventh Conference on Design, Automation and Test in Europe (DATE 2006)*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 806–811.

[68] M. Paleczny, C. Vick, and C. Click, "The Java HotSpot server compiler," in *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM 2001)*. Berkeley, CA, USA: USENIX Association, April 2001.

[69] J. Child, "Real-time Java takes aim at embedded control," *RTC*, August 2004.

[70] K. D. Nilsen, "Issues in the design and implementation of real-time Java," *Java Developer's Journal*, vol. 1, no. 1, November 1996.

[71] K. Nilsen, "Adding real-time capabilities to Java," *Communications of the ACM*, vol. 41, no. 6, pp. 49–56, June 1998.

[72] G. Bollella and J. Gosling, "The real-time specification for Java," *Computer*, vol. 33, no. 6, pp. 47–54, June 2000.

[73] D. S. Hardin, "Real-time objects on the bare metal: An efficient hardware realization of the Java virtual machine," *Proceedings of the Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, pp. 53–59, May 2001.

[74] D. C. Sharp, E. Pla, and K. R. Luecke, "Evaluating mission critical large-scale embedded system performance in real-time Java," in *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS 2003)*. Washington, DC, USA: IEEE Computer Society, December 2003, pp. 362–365.

[75] F. Siebert, "Hard real-time garbage-collection in the Jamaica virtual machine," in *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA 1999)*. Washington, DC, USA: IEEE Computer Society, December 1999, pp. 96–102.

[76] J. McEnery, D. Hickey, and M. Boubekeur, "Empirical evaluation of two mainstream RTSJ implementations," in *Proceedings of the Fifth International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*. New York, NY, USA: ACM, September 2007, pp. 47–54.

[77] A. Corsaro and D. C. Schmidt, "The design and performance of the jRate real-time Java implementation," in *Proceedings of the Fourth International Symposium on Distributed Objects and Applications (DOA 2002)*, ser. Lecture Notes in Computer Science, vol. 2519. London, UK: Springer-Verlag, October 2002, pp. 900–921.

[78] J. Baker, A. Cunei, C. Flack, F. Pizlo, M. Prochazka, J. Vitek, A. Armbruster, E. Pla, and D. Holmes, "A real-time Java virtual machine for avionics: An experience report," in *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002)*. Los Alamitos, CA, USA: IEEE Computer Society, April 2006, pp. 384–396.

[79] D. F. Bacon, P. Cheng, and V. T. Rajan, "The Metronome: A simpler approach to garbage collection in real-time systems," in *On The Move to Meaningful Internet Systems: OTM 2003 Workshops*, ser. Lecture Notes in Computer Science, R. Meersman and Z. Tari, Eds., vol. 2889. Springer Berlin, November 2003, pp. 466–478.

[80] S. G. Robertz, R. Henriksson, K. Nilsson, A. Blomdell, and I. Tarasov, "Using real-time Java for industrial robot control," in *Proceedings of the Fifth International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2007)*. New York, NY, USA: ACM, 2007, pp. 104–110.

[81] J. Auerbach, D. F. Bacon, B. Blainey, P. Cheng, M. Dawson, M. Fulton, D. Grove, D. Hart, and M. Stoodley, "Design and implementation of a comprehensive real-time Java virtual machine," in *Proceedings of the Seventh ACM and IEEE International Conference on Embedded Software (EMSOFT 2007)*. New York, NY, USA: ACM, September 2007, pp. 249–258.

[82] "L-3 telemetry selects Aonix PERC VM for upgrade of Java real-time data acquisition system," Press release, April 2006. Available: http://www.aonix.com/pr_04.04.06a.html

[83] J. Auerbach, D. F. Bacon, D. T. Iercan, C. M. Kirsch, V. T. Rajan, H. Roeck, and R. Trummer, "Java takes flight: Time-portable real-time programming with exotasks," in *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2007)*, vol. 42, no. 7. New York, NY, USA: ACM, 2007, pp. 51–62.

[84] J. Martin, "Sun's James Gosling shows what Java can do," *CNET News*, May 2007. Available: http://www.news.com/2300-1012_3-6183339.html

[85] T. Carmel-Veilleux and M. Morissette, "SONIA 2007: Exploring the depths with ease," École de technologie supérieure, Tech. Rep., 2007.

[86] "National Oilwell Varco selects Aonix PERC for Java-based robotic drilling," Press release, September 2006. Available: http://www.aonix.com/pr_09.25.06c.html

[87] "Linux and real-time Java power German traffic lights," August 2007. Available: http://www.linuxdevices.com/news/NS2015665496.html

[88] A. Corsaro and D. C. Schmidt, "Evaluating real-time Java features and performance for real-time embedded systems," in *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002)*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 90–100.

[89] P. Mikhalenko, "Real-time Java: An introduction," *ONJava.com*, May 2006. Available: http://www.onjava.com/pub/a/onjava/2006/05/10/real-time-java-introduction.html

[90] G. Bollella, B. Delsart, R. Guider, C. Lizzi, and F. Parain, "Mackinac: Making HotSpot real-time," in *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, May 2005, pp. 45–54.

[91] N. Zhang, A. Burns, and M. Nicholson, "Pipelined processors and worst case execution times," *Real-Time Systems*, vol. 5, no. 4, pp. 319–343, October 1993.

[92] N. Williams, B. Marre, P. Mouy, and M. Roger, "PathCrawler: Automatic generation of path tests by combining static and dynamic analysis," in *Proceedings of the Fifth European Dependable Computing Conference (EDCC 2005)*, ser. Lecture Notes in Computer Science, vol. 3463, April 2005, pp. 281–292.

[93] C. Im, "A hybrid approach for derivation of tight execution time bounds of program-segments and service time bounds of simple object methods in real-time distributed computing systems," Ph.D. dissertation, University of California, Irvine, 2005.

[94] A. Zerzelidis and A. J. Wellings, "Requirements for a real-time .NET framework," *SIGPLAN Notices*, vol. 40, no. 2, pp. 41–50, February 2005.

[95] M. Schoeberl, "A time predictable instruction cache for a Java processor," in *On the Move to Meaningful Internet Systems 2004*, ser. Lecture Notes in Computer Science, R. Meersman and Z. Tari, Eds., vol. 3292, January 2004, pp. 371–382.

[96] M. Schoeberl, "A time predictable Java processor," in *Design, Automation, and Test in Europe (DATE 2006)*. 3001 Leuven, Belgium: European Design and Automation Association, March 2006, pp. 800–805.

[97] M. Schoeberl, "A Java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, June 2007.

[98] R. Kirner, "The programming language wcetC," Technische Universität Wien, Institut für Technische Informatik, Research Report 2/2002, May 2002.

[99] R. Wilhelm, J. Engblom, S. Thesing, and D. Whalley, "Industrial requirements for WCET tools: Answers to the ARTIST questionnaire," in *Proceedings of the Third International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*, July 2003, pp. 39–43.

[100] J. M. O'Connor and M. Tremblay, "picoJava-I: the Java virtual machine in hardware," *IEEE Micro*, vol. 17, no. 2, pp. 45–53, March/April 1997.

[101] G. Lawton, "Moving Java into mobile phones," *Computer*, vol. 35, no. 6, pp. 17–20, June 2002.

[102] C. Porthouse, "PRODUCT HOW-TO: Use ARM DBX hardware extensions to accelerate Java in space-constrained embedded apps," *Embedded.com*, October 2007. Available: http://www.embedded.com/products/softwaretools/202402546

[103] R. B. Smith, "SPOTWorld and the Sun SPOT," in *Proceedings of the Sixth International Conference on Information Processing in Sensor Networks (IPSN 2007)*. New York, NY, USA: ACM, April 2007, pp. 565–566.

[104] D. Loomis, *The TINI Specification and Developer's Guide*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[105] "Velocity Semiconductor debuts first in series of mixed-language microcontrollers," Press release, May 2003. Available: http://www.velocitysemi.com/pr001.htm

[106] Z. Liang, J. Plosila, and K. Sere, "Asynchronous Java accelerator for embedded Java virtual machine," in *Proceedings of the IEEE Sixth Circuits and Systems Symposium on Emerging Technologies: Frontiers of Mobile and Wireless Communication*, May 2004, pp. 253–256.

[107] C. Holland, "Moon2 improves Java applications," *Embedded.com*, April 2002. Available: http://www.embedded.com/23901577

[108] J. Kreuzinger, R. Marston, T. Ungerer, U. Brinkschulte, and C. Krakowski, "The Komodo Project: Thread-based event handling supported by a multithreaded Java microcontroller," vol. 2. Los Alamitos, CA, USA: IEEE Computer Society, September 1999, p. 2122.

[109] J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and T. Ungerer, "Real-time event-handling and scheduling on a multithreaded Java microcontroller," *Microprocessors and Microsystems*, vol. 27, no. 1, pp. 19–31, February 2003.

[110] U. Brinkschulte and M. Pacher, "Improving the real-time behaviour of a multithreaded Java microcontroller by control theory and model based latency prediction," in *Tenth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. Los Alamitos, CA, USA: IEEE Computer Society, February 2005, pp. 82–96.

[111] S. A. Ito, L. Carro, and R. P. Jacobi, "Making Java work for microcontroller applications," *IEEE Design and Test of Computers*, vol. 18, no. 5, pp. 100–110, September-October 2001.

[112] A. C. S. Beck and L. Carro, "A VLIW low power Java processor for embedded applications," in *Proceedings of the Seventeenth Symposium on Integrated Circuits and System Design (SBCCI 2004)*. New York, NY, USA: ACM, 2004, pp. 157–162.

[113] D. J. Newman, "Embedded Java controllers," *Circuit Cellar*, no. 166, pp. 16–21, May 2004.

[114] T. R. Halfhill, "Imsys hedges bets on Java," *Microprocessor Report*, August 2000.

[115] *IM1101C Technical Reference Manual*, 0th ed., Imsys Technologies, October 2004. Available: http://www.imsys.se/documentation/manuals/tr-CjipTechref.pdf

[116] B. Bose, M. E. Tuna, and J. M. Nagy, "LavaCORE: configurable Java processor core," in *Proceedings of the IEEE Aerospace Conference*, vol. 4, March 2002.

[117] M. Zabel, T. B. Preuber, P. Reichel, and R. G. Spallek, "Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture," in *Proceedings of the Tenth Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*. Washington, DC, USA: IEEE Computer Society, August 2007, pp. 59–62.

[118] T. Yiyu, L. W. Yiu, Y. C. Hang, R. Li, and A. S. Fong, "A Java processor with hardware-support object-oriented instructions," *Microprocessors and Microsystems*, vol. 30, no. 8, p. 469, December 2006.

[119] F. Gruian and M. Westmijze, "BlueJEP: a flexible and high-performance Java embedded processor," in *Proceedings of the Fifth International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2007)*. New York, NY, USA: ACM, September 2007, pp. 222–229.

[120] M. Schoeberl, "JOP: A Java optimized processor for embedded real-time systems," Ph.D. dissertation, Vienna University of Technology, Vienna, Austria, January 2005.

[121] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1038–1054, July 2003.

[122] K. Hansen, "Bluetooth API for JOP: Implementation of the Java specification request 82," Master's thesis, Copenhagen Business School, February 2007.

[123] A. J. Perlis, "Epigrams on programming," *SIGPLAN Notices*, vol. 17, no. 9, pp. 7–13, September 1982.

[124] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001)*. New York, NY, USA: ACM, June 2001, pp. 54–61.

[125] T. P. Jensen, D. L. Metayer, and T. Thorn, "Verification of control flow based security properties," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, May 1999, pp. 89–103.

[126] R. Pawlak, "Spoon: Annotation-driven program transformation — the AOP case," in *Proceedings of the First Workshop on Aspect-Oriented Middleware Development (AOMD 2005)*. New York, NY, USA: ACM Press, 2005.

[127] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1999)*. IBM Press, November 1999, p. 13.

[128] J. E. Shaw, "Visualization tools for optimizing compilers," Master's thesis, McGill University, August 2005.

[129] M. C. Rinard *et al.*, "Flex compiler infrastructure." Available: http://flex-compiler.csail.mit.edu/

[130] H. Agrawal, "Dominators, super blocks, and program coverage," in *Proceedings of the Twenty-first ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1994)*. New York, NY, USA: ACM, January 1994, pp. 25–34.

[131] M. Eichberg, "BAT$_2$XML: XML-based Java bytecode representation," in *Proceedings of the First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2005)*, ser. Electronic Notes in Theoretical Computer Science, vol. 141, no. 1. Elsevier B.V., December 2005, pp. 93–107.

[132] E. M. Gagnon and L. J. Hendren, "SableCC, an object-oriented compiler framework," in *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 1998)*. Washington, DC, USA: IEEE Computer Society, August 1998, p. 140.

[133] M. Cortés, M. Fontoura, and C. Lucena, "Framework evolution tool," *Journal of Object Technology*, vol. 5, no. 8, pp. 101–124, November-December 2006.

[134] B. Bokowski and A. Spiegel, "Barat — a front–end for Java," Freie Universität Berlin, Tech. Rep. B-98-09, December 1998.

[135] M. Schoeberl and R. Pedersen, "WCET analysis for a Java processor," in *Proceedings of the Fourth International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, October 2006.

[136] M. Dahm, "Byte code engineering with the BCEL API," Freie Universität Berlin, Tech. Rep. B-17-98, April 2001.

[137] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: a code manipulation tool to implement adaptable systems," in *Adaptable and extensible component systems (Systèmes à composants adaptables et extensibles)*, October 2002.

[138] A. A. Evstiougov-Babaev, "Call graph and control flow graph visualization for developers of embedded applications," in *Software Visualization*, ser. Lecture Notes in Computer Science, vol. 2269. Springer Berlin, 2002, pp. 611–614.

[139] B. L. Titzer, "Avrora: The AVR simulation and analysis framework," Master's thesis, University of California, Los Angeles, 2004.

[140] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Software: Practice and Experience*, vol. 25, no. 7, pp. 811–829, July 1995.

[141] M. V. Emmerik and T. Waddington, "Using a decompiler for real-world source recovery," in *Proceedings of the Eleventh Working Conference on Reverse Engineering (WCRE 2004)*. Washington, DC, USA: IEEE Computer Society, November 2004, pp. 27–36.

[142] L. Ramshaw, "Eliminating go to's while preserving program structure," *Journal of the ACM*, vol. 35, no. 4, pp. 893–920, October 1988.

[143] C. Cifuentes, "Reverse compilation techniques," Ph.D. dissertation, Queensland University of Technology, July 1994.

[144] H. van Vliet, "Mocha, the Java decompiler." Available: http://www.brouhaha.com/~eric/software/mocha/

[145] M. Batchelder and L. Hendren, "Obfuscating Java: The most pain for the least gain," in *Proceedings of the Sixteenth International Conference on Compiler Construction (CC 2007)*, ser. Lecture Notes in Computer Science, vol. 4420. Springer Berlin, March 2007, pp. 96–110.

[146] T. Hou, H. Chen, and M. Tsai, "Three control flow obfuscation methods for Java software," *IEE Proceedings Software*, vol. 153, no. 2, pp. 80–86, April 2006.

[147] L. Ertaul and S. Venkatesh, "JHide — a tool kit for code obfuscation," in *Proceedings of the Eighth IASTED International Conference on Software Engineering and Applications (SEA 2004)*, November 2004, pp. 133–138.

[148] T. A. Proebsting and S. A. Watterson, "Krakatoa: Decompilation in Java (does bytecode reveal source?)," in *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS 1997)*. Berkeley, CA, USA: USENIX Association, June 1997, pp. 185–197.

[149] J. Miecznikowski and L. J. Hendren, "Decompiling Java bytecode: Problems, traps and pitfalls," in *Proceedings of the Eleventh International Conference on Compiler Construction (CC 2002)*, ser. Lecture Notes in Computer Science, vol. 2304. London, UK: Springer-Verlag, March 2002, pp. 111–127.

[150] N. A. Naeem and L. Hendren, "Programmer-friendly decompiled Java," in *Proceedings of the Fourteenth IEEE International Conference on Program Comprehension (ICPC 2006)*. Washington, DC, USA: IEEE Computer Society, June 2006, pp. 327–336.

[151] J. Hoenicke, "Java Optimize and Decompile Environment (JODE)." Available: http://jode.sourceforge.net/

[152] K. Kumar, "JReversePro." Available: http://jrevpro.sourceforge.net/

[153] B. Jasik, "dis - a fast Java disassembler." Available: http://www.cs.princeton.edu/~benjasik/dis/index.html

[154] J. Meyer and D. Reynaud, "Jasmin." Available: http://jasmin.sourceforge.net/

[155] S. Dambalkar, "javap - the Java class file disassembler." Available: http://java.sun.com/javase/6/docs/technotes/tools/solaris/javap.html

[156] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Software: Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, September 2000.

[157] I. G. Tollis, G. D. Battista, P. Eades, and R. Tamassia, *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, July 1998.

[158] N. Mathis, *Storm Warning: The Story of a Killer Tornado*. Touchstone, March 2007.

[159] W. E. Wong and J. Li, "An integrated solution for testing and analyzing Java applications in an industrial setting," in *Proceedings of the Twelfth Asia-Pacific Software Engineering Conference (APSEC 2005)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 576–583.

[160] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, December 1976.

[161] S. Pestov, "jEdit — programmer's text editor." Available: http://www.jedit.org/

[162] E. Gamma and K. Beck, *Contributing to Eclipse: Principles, Patterns, and Plugins*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2003.

[163] T. Boudreau, J. Glick, and V. Spurlin, *NetBeans: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002.

[164] J. Gustedt, O. A. Mæhle, and J. A. Telle, "The treewidth of Java programs," in *The Fourth International Workshop on Algorithm Engineering and Experiments (ALENEX 2002)*, ser. Lecture Notes in Computer Science, vol. 2409. London, UK: Springer-Verlag, 2002, pp. 86–97.

[165] R. C. Martin, *Java and C++: A Critical Comparison*, ser. Sigs Reference Library. New York, NY, USA: Cambridge University Press, February 1998, no. 10, pp. 51–68.

[166] C. W. Probst, "Modular control flow analysis for libraries," in *Proceedings of the Ninth International Symposium on Static Analysis (SAS 2002)*, ser. Lecture Notes in Computer Science, vol. 2477. London, UK: Springer-Verlag, September 2002, pp. 165–179.

[167] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani, "A study of devirtualization techniques for a Java just-in-time compiler," in *Proceedings of the Fifteenth ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000)*. New York, NY, USA: ACM, October 2000, pp. 294–310.

[168] N. Walkinshaw, M. Roper, and M. Wood, "The Java system dependence graph," in *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, September 2003, pp. 55–64.

[169] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," in *Proceedings of the London Mathematical Society*, ser. 2, vol. 42. London Mathematical Society, November 1936, pp. 230–265.

[170] J. Gustafsson, "Analyzing execution-time of object-oriented programs using abstract interpretation," Ph.D. dissertation, Mälardalen University, Västerås, Sweden, May 2000.

[171] J. Gustafsson, "Worst case execution time analysis of object-oriented programs," *Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pp. 71–76, 2002.

[172] P. Puschner and A. Burns, "Guest editorial: A review of worst-case execution-time analysis," *Real-Time Systems*, vol. 18, no. 2-3, pp. 115–128, May 2000.

[173] J. Engblom, A. Ermedahl, M. Sjoedin, J. Gustafsson, and H. Hansson, "Worst-case execution-time analysis for embedded real-time systems," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 4, pp. 437–455, August 2003.

[174] C. Stoif, "A survey of the research on analysis of the worst-case execution-time (WCET)," Master's thesis, University of Technology, Vienna, August 2006.

[175] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution time problem—Overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, April 2008.

[176] G. Bernat, A. Burns, and M. Newby, "Probabilistic timing analysis: An approach using copulas," *Journal of Embedded Computing*, vol. 1, no. 2, pp. 179–194, April 2005.

[177] I. Wenzel, "Measurement-based timing analysis of superscalar processors," Ph.D. dissertation, Technische Universität Wien, Vienna, Austria, November 2006.

[178] P. A. Guedes and S. V. Cavalcante, "On the design of an extensible platform for flow analysis of Java using abstract interpretation," in *Proceedings of the Third International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*, July 2003, pp. 47–50.

[179] J. Gustafsson, B. Lisper, R. Kirner, and P. Puschner, "Code analysis for temporal predictability," *Real-Time Systems*, vol. 32, no. 3, pp. 253–277, March 2006.

[180] J. Engblom, "Processor pipelines and static worst-case execution time analysis," Ph.D. dissertation, Uppsala University, April 2002.

[181] A. C. Shaw, "Reasoning about time in higher-level language software," *IEEE Transactions on Software Engineering*, vol. 15, no. 7, pp. 875–889, July 1989.

[182] M. Schoeberl, "Design and implementation of an efficient stack machine," in *Proceedings of the Nineteenth IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*. Washington, DC, USA: IEEE Computer Society, April 2005.

[183] F. Stappert, A. Ermedahl, and J. Engblom, "Efficient longest executable path search for programs with complex flows and pipeline effects," in *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2001)*. New York, NY, USA: ACM, November 2001, pp. 132–140.

[184] A. Ermedahl, F. Stappert, and J. Engblom, "Clustered worst-case execution-time calculation," *IEEE Transactions on Computers*, vol. 54, no. 9, pp. 1104–1122, September 2005.

[185] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *Real-Time Systems*, vol. 1, no. 2, pp. 159–176, September 1989.

[186] A. Colin and I. Puaut, "A modular and retargetable framework for tree-based WCET analysis," in *Proceedings of the Thirteenth Euromicro Conference on Real-Time Systems (ECRTS 2001)*. Washington, DC, USA: IEEE Computer Society, June 2001, pp. 37–44.

[187] P. Altenbernd, "On the false path problem in hard real-time programs," in *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems (EUR-WRTS 2006)*. Los Alamitos, CA, USA: IEEE Computer Society, June 1996, pp. 102–107.

[188] A. Colin and G. Bernat, "Scope-tree: A program representation for symbolic worst-case execution time analysis," in *Proceedings of the Fourteenth Euromicro Conference on Real-Time Systems (ECRTS 2002)*. Washington, DC, USA: IEEE Computer Society, June 2002, pp. 50–59.

[189] F. Stappert and P. Altenbernd, "Complete worst-case execution time analysis of straight-line hard real-time programs," *Journal of Systems Architecture*, vol. 46, no. 4, pp. 339–355, February 2000.

[190] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 12, pp. 1477–1487, December 1997.

[191] Y.-T. S. Li, S. Malik, and A. Wolfe, "Efficient microarchitecture modeling and path analysis for real-time software," in *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium (RTSS 1995)*. Washington, DC, USA: IEEE Computer Society, 1995, p. 298.

[192] "lp_solve." Available: http://lpsolve.sourceforge.net/

[193] T. Harmon and R. Klefstad, "Interactive back-annotation of worst-case execution time analysis for Java microprocessors," in *Proceedings of the Thirteenth IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, August 2007, pp. 209–216.

[194] R. A. Quinnell, "Static analysis stomps on bugs," *EE Times*, March 2008. Available: http://www.eetimes.com/showArticle.jhtml?articleID=206902140

[195] R. Jetley and P. Anderson, "Using static analysis to evaluate software in medical devices," *Embedded.com*, April 2008.

[196] A. Prantl, "TuBound - a tool for worst-case execution time analysis," in *Proceedings of the Eighth International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, July 2008.

[197] A. Ermedahl, "A modular tool architecture for worst-case execution time analysis," Ph.D. dissertation, Uppsala University, Uppsala, Sweden, June 2003.

[198] H. Cassé and P. Sainrat, "OTAWA, a framework for experimenting WCET computations," in *Proceedings of the Third Embedded Real-Time Software Conference (ERTS 2006)*, January 2006.

[199] X. Li, Y. Liang, T. Mitra, and A. Roychoudhur, "Chronos: A timing analyzer for embedded software," *Science of Computer Programming*, vol. 69, pp. 56–67, December 2007.

[200] N. Holsti and S. Saarinen, "Status of the Bound-T WCET tool," in *Proceedings of the Second International Workshop on Worst-Case Execution Time Analysis (WCET 2002)*, June 2002.

[201] G. Bernat and M. Bennett, "Identifying opportunities for worst-case execution time reduction in an avionics system," in *Proceedings of the Twelveth International Conference on Reliable Software Technologies (Ada-Europe 2007)*, June 2007.

[202] W. Zhao, D. Whalley, C. Healy, and F. Mueller, "Improving WCET by applying a WC code-positioning optimization," *ACM Transactions on Architecture and Code Optimization*, vol. 2, no. 4, pp. 335–365, 2005.

[203] L. Ko, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M. Harmon, "Supporting the specification and analysis of timing constraints," in *Proceedings of the Second IEEE Real-Time Technology and Applications Symposium (RTAS 1996)*, June 1996, pp. 170–178.

[204] J. R. P. Ribeiro, N. C. da Silva, and C. E. Morón, "A visual environment for the development of parallel real-time programs," in *Proceedings of the 12th International Parallel Processing Symposium / Ninth Symposium on Parallel and Distributed Processing (IPPS/SPDP 1998)*, ser. Lecture Notes in Computer Science, vol. 1388. Springer Berlin, March 1998, pp. 994–1014.

[205] R. Kirner, "Consideration of optimizing compilers in the context of WCET analysis," in *Proceedings of the Deutsche Informatiktage 2000*, October 2000, pp. 123–126.

[206] P. Persson and G. Hedin, "An interactive environment for real-time software development," in *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 2000)*. Washington, DC, USA: IEEE Computer Society, June 2000, pp. 57–68.

[207] J. Fauster, R. Kirner, and P. Puschner, "Intelligent editor for writing worst-case-execution-time-oriented programs," in *Proceedings of the Third International Conference on Embedded Software (EMSOFT 2003)*, ser. Lecture Notes in Computer Science, vol. 2855. Springer-Verlag, 2003, pp. 190–205.

[208] W. Zhao, P. Kulkarni, D. Whalley, C. Healy, F. Mueller, and G.-R. Uh, "Timing the WCET of embedded applications," in *Proceedings of the Tenth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)*, May 2004, pp. 472–481.

[209] P. Yu and T. Mitra, "Satisfying real-time constraints with custom instructions," in *Proceedings of the Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2005)*. New York, NY, USA: ACM, September 2005, pp. 166–171.

[210] W. Zhao, W. Kreahling, D. Whalley, C. Healy, and F. Mueller, "Improving WCET by optimizing worst-case paths," in *Proceedings of the Eleventh IEEE Real Time on Embedded Technology and Applications Symposium (RTAS 2005)*. Washington, DC, USA: IEEE Computer Society, March 2005, pp. 138–147.

[211] M. Nolin, J. Mäki-Turja, and K. Hänninen, "Achieving industrial strength timing predictions of embedded system behavior," in *The 2008 International Conference on Embedded Systems and Applications (ESA 2008)*, Las Vegas, Nevada, USA, July 2008.

[212] T. Harmon, R. Kirner, M. Schoeberl, and R. Klefstad, "A modular worst-case execution time analysis tool for Java processors," in *Proceedings of the Fourteenth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, April 2008, pp. 47–57.

[213] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[214] B. Cole, "Embedded programmer shortage: A problem, but what kind? How bad? And how to solve it?" *Embedded.com*, June 2007. Available: http://www.embedded.com/news/embeddedindustry/200000893

[215] E. Y.-S. Hu, A. Wellings, and G. Bernat, "XRTJ: An extensible distributed high-integrity real-time Java environment," in *Proceedings of the Ninth International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2003)*, ser. Lecture Notes in Computer Science, vol. 2968. Springer Berlin, February 2003, pp. 208–228.

[216] R. Kirner and P. Puschner, "Classification of code annotations and discussion of compiler-support for worst-case execution time analysis," in *Proceedings of the Fifth International Workshop on Worst-Case Execution Time Analysis (WCET 2005)*, July 2005.

[217] R. Kirner, A. Kadlec, P. Puschner, A. Prantl, M. Schordan, and J. Knoop, "Towards a common WCET annotation language: Essential ingredients," in *Proceedings of the Eighth International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, July 2008.

[218] T. Harmon and R. Klefstad, "Toward a unified standard for worst-case execution time annotations in real-time Java," in *Proceedings of the Fifteenth International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2007)*. IEEE Computer Society, March 2007.

[219] B. Lisper, "Fully automatic, parametric worst-case execution time analysis," in *Proceedings of the Third International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*, J. Gustafsson, Ed., July 2003, pp. 77–80.

[220] D. Kazakov and I. Bate, "Towards new methods for developing real-time systems: Automatically deriving loop bounds using machine learning," in *Proceedings of the Eleventh International Conference on Emerging Technologies and Factory Automation (ETFA 2006)*, September 2006.

[221] J. J. Hunt, F. B. Siebert, P. H. Schmitt, and I. Tonin, "Provably correct loops bounds for realtime Java programs," in *Proceedings of the Fourth International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*. New York, NY, USA: ACM Press, October 2006, pp. 162–169.

[222] R. Kirner and M. Schoeberl, "Modeling the function cache for worst-case execution time analysis," in *Proceedings of the Forty-Third Design Automation Conference (DAC 2007)*. San Diego, CA, USA: ACM, June 2007.

[223] W. Puffitsch, "picoJava-II in an FPGA," Master's thesis, Technischen Universität Wien, November 2007.

[224] "GNU Linear Programming Kit." Available: http://www.gnu.org/software/glpk/

[225] J. Gustafsson, "The worst case execution time tool challenge 2006," in *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006)*, November 2006, pp. 233–240.

[226] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. D. Michiel, "PapaBench: A free real-time benchmark," in *Proceedings of the Sixth International Workshop on Worst-Case Execution Time Analysis (WCET 2006)*, July 2006.

[227] A. A. Avižienis, *The Methodology of N-Version Programming*. John Wiley and Sons, 1995, ch. 2.

[228] S. S. Brilliant, J. C. Knight, and N. G. Leveson, "Analysis of faults in an N-version software experiment," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 238–247, February 1990.

[229] L. von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum, "re-CAPTCHA: Human-based character recognition via web security measures," *Science*, August 2008.

[230] P. Puschner and R. Nossal, "Testing the results of static worst-case execution-time analysis," in *Proceedings of the Nineteenth IEEE Real-Time Systems Symposium (RTSS 1998)*. Washington, DC, USA: IEEE Computer Society, December 1998, pp. 134–143.

[231] P. E. Ceruzzi, *A History of Modern Computing*. MIT Press, 2003.

[232] E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly, 2001.

[233] E. R. Harold, "Java's new math, part 1: Real numbers," October 2008. Available: http://www.ibm.com/developerworks/java/library/j-math1/index.html

[234] T. Veldhuizen, "What is a library?" Talk given at the Dagstuhl workshop "Software Libraries: Design and Evaluation", March 2005.

[235] K. Raman, Y. Zhang, M. Panahi, J. A. Colmenares, R. Klefstad, and T. Harmon, "RTZen: Highly predictable, real-time Java middleware for distributed and embedded systems," in *Middleware*, ser. Lecture Notes in Computer Science. Springer Berlin, November 2005, pp. 225–248.

[236] R. Kirner, M. Grössing, and P. Puschner, "Comparing WCET and resource demands of trigonometric functions implemented as iterative calculations vs. table-lookup," in *Proceedings of the Sixth International Workshop on Worst-Case Execution Time Analysis (WCET 2006)*, F. Mueller, Ed., 2006.

[237] J.-M. Dautelle, "Fully time deterministic Java," in *Proceedings of the AIAA SPACE 2007 Conference and Exposition*. American Institute of Aeronautics and Astronautics, Inc., September 2007.

[238] "The SPARK examiner," Praxis High Integrity Systems. Available: http://www.praxis-his.com/sparkada/examiner.asp

[239] "SCADE suite," Esterel Technologies. Available: http://www.esterel-technologies.com/products/scade-suite/

[240] J. Kwon, A. Wellings, and S. King, "Assessment of the Java programming language for use in high integrity systems," *SIGPLAN Notices*, vol. 38, no. 4, pp. 34–46, April 2003.

[241] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Professional, April 2003.

[242] A. Burns, "The Ravenscar profile," *ACM SIGAda Ada Letters*, vol. XIX, no. 4, pp. 49–52, 1999.

[243] N. Hamilton, "The A-Z of programming languages: Ada," *Computerworld*, April 2008.

[244] "JSR 302: Safety critical Java technology." Available: http://jcp.org/en/jsr/detail?id=302

[245] E. Y.-S. Hu, E. Jenn, N. Valot, and A. Alonso, "Safety critical applications and hard real-time profile for Java: a case study in avionics," in *Proceedings of the Fourth International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*. New York, NY, USA: ACM Press, October 2006, pp. 125–134.

[246] K. Nilsen, "Guidelines for scalable Java development of real-time systems," March 2006.

[247] J. Kwon, A. Wellings, and S. King, "Ravenscar-Java: A high integrity profile for real-time Java," in *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande (JGI 2002)*. New York, NY, USA: ACM, November 2002, pp. 131–140.

[248] M. Schoeberl, H. Sondergaard, B. Thomsen, and A. P. Ravn, "A profile for safety critical Java," in *Proceedings of the Tenth IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2007)*, May 2007.

[249] M. Schoeberl, "Mission modes for safety critical Java," in *Proceedings of the Fifth IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2007)*, May 2007.

[250] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman, *JML Reference Manual*, Iowa State University, May 2008.

[251] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt, "The KeY tool," *Software and Systems Modeling*, vol. 4, no. 1, pp. 32–54, February 2005.

[252] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, September 1994.

[253] M. R. Woodward, D. Hedley, and M. A. Hennell, "Experience with path analysis and testing of programs," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, pp. 278–286, May 1980.

[254] "Inquiry board traces Ariane 5 failure to overflow error," *SIAM News*, vol. 29, no. 8, October 1996.

[255] T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad, "Toward libraries for real-time Java," in *Proceedings of the Eleventh IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC 2008)*, May 2008, pp. 458–462.

[256] Q. Li, *Real-Time Concepts for Embedded Systems*. CMP Books, July 2003, ch. 13.

[257] F. Siebert, "Real-time garbage collection in multi-threaded systems on a single processor," in *Proceedings of the Twentieth IEEE Real-Time Systems Symposium (RTSS 1999)*, Phoenix, Arizona, December 1999, pp. 277–278.

[258] R. Henriksson, "Scheduling garbage collection in embedded systems," Ph.D. dissertation, Lund Institute of Technology, 1998. Available: Henriksson1998.pdf

[259] M. Schoeberl and J. Vitek, "Garbage collection for safety critical Java," in *Proceedings of the Fifth International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*. New York, NY, USA: ACM, September 2007, pp. 85–93.

[260] F. Siebert, "Proving the absence of RTSJ related runtime errors through data flow analysis," in *Proceedings of the Fourth International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2006)*. New York, NY, USA: ACM Press, October 2006, pp. 152–161.

[261] S. Smith, S. W. Lawson, and A. Lawson, "Can real-time software engineering be taught to Java programmers?" in *Proceedings of the Seventeenth Conference on Software Engineering Education and Training (CSEET 2004)*. Washington, DC, USA: IEEE Computer Society, March 2004, pp. 124–129.

[262] J. Yan and W. Zhang, "A time-predictable VLIW processor and its compiler support," *Real-Time Systems*, vol. 38, no. 1, pp. 67–84, January 2008.

[263] M. Pfeffer, T. Ungerer, S. Fuhrmann, J. Kreuzinger, and U. Brinkschulte, "Real-time garbage collection for a multithreaded Java microcontroller," *Real-Time Systems*, vol. 26, no. 1, pp. 89–106, January 2004.

[264] T. Harmon and R. Klefstad, "A survey of worst-case execution time analysis for real-time Java," in *Proceedings of the Ninth International Workshop on Java and Components for Parallelism, Distribution and Concurrency (JAVAPDC 2007)*. IEEE Computer Society, March 2007.

[265] P. Puschner and G. Bernat, "WCET analysis of reusable portable code," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS 2001)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 45–52.

[266] E. Y.-S. Hu, G. Bernat, and A. Wellings, "A static timing analysis environment using Java architecture for safety critical real-time systems," in *Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*. Los Alamitos, CA, USA: IEEE Computer Society, January 2002, pp. 77–84.

[267] J. J. Hunt, I. Tonin, and F. B. Siebert, "Using global data flow analysis on bytecode to aid worst case execution time analysis for realtime java programs," in *Proceedings of the Sixth International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2008)*. New York, NY, USA: ACM, September 2008, pp. 97–105.

[268] J. Ventura, F. Siebert, A. Walter, and J. Hunt, "HIDOORS - a high integrity distributed deterministic Java environment," in *Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*. Los Alamitos, CA, USA: IEEE Computer Society, January 2002, pp. 113–118. Available: http://www.hidoors.org/

[269] I. Bate, G. Bernat, G. Murphy, and P. Puschner, "Low-level analysis of a portable Java byte code WCET analysis framework," in *Proceedings of the Seventh International Conference on Real-Time Computing Systems and Applications (RTCSA 2000)*. Los Alamitos, CA, USA: IEEE Computer Society, December 2000, pp. 39–48.

[270] I. Bate, G. Bernat, and P. Puschner, "Java virtual-machine support for portable worst-case execution-time analysis," in *Proceedings of the Fifth IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2002)*, April 2002, pp. 83–90.

[271] J. Consortium, "Real-time core extensions," September 2000.

[272] E. Y.-S. Hu, A. Wellings, and G. Bernat, "Deriving Java machine timing models for portable worst-case execution time analysis," in *On the Move to Meaningfull Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems*, ser. Lecture Notes in Computer Science, vol. 2889. Springer, November 2003, pp. 411–424.

[273] Z. Chai, Z. Tang, L. Wang, and S. Tu, "An effective instruction optimization method for embedded real-time Java processor," in *2005 International Conference on Parallel Processing Workshops (ICPPW 2005)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2005, pp. 225–231.

[274] E. Y.-S. Hu, A. Wellings, and G. Bernat, "Gain time reclaiming in high performance real-time Java systems," in *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2003)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2003, pp. 249–256.

[275] A. Corsaro and C. Santoro, "Optimizing JVM object operations to improve WCET predictability," in *Proceedings of the Fourth International Workshop on Worst-Case Execution Time Analysis (WCET 2004)*, June 2004, pp. 15–18.

[276] Z. Chai, W. Chen, Z. Tang, Z. Chen, and S. Tu, "Asynchronous transfer of control in the RTSJ-compliant Java processor," in *The Fifth International Conference on Computer and Information Technology (CIT 2005)*. IEEE Computer Society, September 2005, pp. 764–770.

[277] D. R. Cok and J. R. Kiniry, "ESC/Java2: Uniting ESC/Java and JML," in *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2005)*, ser. Lecture Notes in Computer Science, vol. 3362. Springer Berlin / Heidelberg, March 2005, pp. 108–128.

[278] C. Walls and N. Richards, *XDoclet in Action*. Manning Publications, December 2003.

[279] L. G. DeMichiel, L. Ümit Yalçinalp, and S. Krishnan, "Enterprise JavaBeans specification," August 2001.

[280] J. Gosling, B. Joy, G. L. S. Jr., and G. Bracha, *The Java Language Specification*, 3rd ed., ser. The Java Series. Boston, Massachusetts: Addison-Wesley Professional, June 2005.

[281] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed., ser. The Java Series. Boston, Massachusetts: Addison-Wesley Professional, April 1999.

[282] "Annogen." Available: http://annogen.codehaus.org/

[283] S. Chiba, "Javassist — a reflection-based programming wizard for Java," in *Proceedings of the ACM OOPSLA 1998 Workshop on Reflective Programming in C++ and Java*, October 1998.

[284] "JSR 269: Pluggable annotation processing API." Available: http://jcp.org/en/jsr/detail?id=269

[285] M. M. Papi and M. D. Ernst, "Annotations on Java types," November 2006. Available: http://jcp.org/en/jsr/detail?id=308

[286] "OpenJDK." Available: http://openjdk.java.net/

Appendices

A A Survey of Worst-Case Execution Time Analysis for Java

While the theory of worst-case execution time has been addressed by hundreds of research papers over the last two decades,[2] fewer than twenty publications spanning only six years have tackled the problem of Java in WCET analysis. Experimental tools for performing WCET analysis in Java are even rarer, and commercial products simply do not exist [264].

Yet the research accomplished thus far shows promise. As discussed in Chapter 3, Java is an attractive platform for WCET analysis, and it offers a number of new avenues of research. Advancements in this area are increasingly important as interest in using Java for real-time systems grows.

As a complement to the existing work, this appendix provides a comprehensive survey of efforts in combining WCET analysis with the Java domain. It is a snapshot of the current state of the art and provides a convenient summary for future researchers in this field.

[2] Kligerman's and Stoyenko's 1986 paper [24] is generally considered the first publication to address the problem of WCET.

Each contribution has been categorized into one of four basic groups: 1) bytecode as an intermediate representation, 2) high-level WCET analysis, 3) low-level WCET analysis, and 4) miscellaneous work, a catch-all category for research that does not fit cleanly into the first three. Where appropriate, a discussion of the strengths and weaknesses of a technique or tool is also included.

A.1 Bytecode as an Intermediate Representation

The earliest known work to combine the Java domain with WCET analysis was Bernat's proposal [64] to use Java bytecode in WCET tools. Noting that WCET analysis was not being adopted by industry practitioners, Bernat suggested that a lack of portability in WCET tools was the cause. Existing software for WCET analysis was normally restricted to a single source language, a specific compiler, and a unique configuration of processor, memory, and clock speed. As a result of these constraints, the usual industry practice was to forgo these tools and rely instead on *ad hoc* measurement, a more flexible but inefficient and error-prone technique that often leads to overly optimistic WCET bounds.

To address this problem, Bernat proposed that Java bytecode could serve as an intermediate representation for WCET tools, analogous to register transfer languages acting as intermediate representations in compilers. The assumption was that if WCET tools were to standardize on bytecode, they would be more portable, versatile, and thus more attractive to industry.

The emphasis, then, was not on Java as a real-time programming language but rather as a catalyst. For example, real-time programs written in C or Ada could be translated to Java bytecode, then translated from bytecode to machine code. This multi-stage process would, in theory, allow a single WCET tool designed only for bytecode to

analyze both C and Ada programs without knowledge of either language.

Compared to most instruction sets, Java bytecode contains enough high-level information to perform a full WCET analysis, but this benefit does not come for free. The move to bytecode brings new challenges, such as how to pass WCET annotations from an arbitrary source language to bytecode and how to integrate knowledge of the target hardware into the bytecode analysis.

Bernat offered a somewhat awkward solution to the first problem: Programmers would be required to invoke methods in a predefined class whenever a WCET annotation was required. For instance, the following Ada statement would indicate a maximum loop bound of 10:

```
WCETAn.Loopcount(10);
```

Bernat also created a prototype tool called Javelin to demonstrate these ideas. It was able to parse Java class files, analyze control flow, and extract loop bound annotations. A diagram of Javelin's operation is shown in Figure A.1.

Discussion

Compared to established techniques, this style of annotation mingles non-functional metadata—that is, the WCET information—with the normal source code statements, making the program more difficult to read. In addition, tools for compiling arbitrary C[3] or Ada[4] source to Java bytecode remain primitive. Most such tools have not progressed beyond the prototype stage or have serious limitations that prevent their use in day-to-day operations. Jazillian, for example, a C-to-Java translator, makes

[3]See `http://www.jazillian.com/competition.html` for a comparison of C-to-Java translators.
[4]Examples of Ada-to-Java translators include AppletMagic, ObjectAda, and JGNAT.

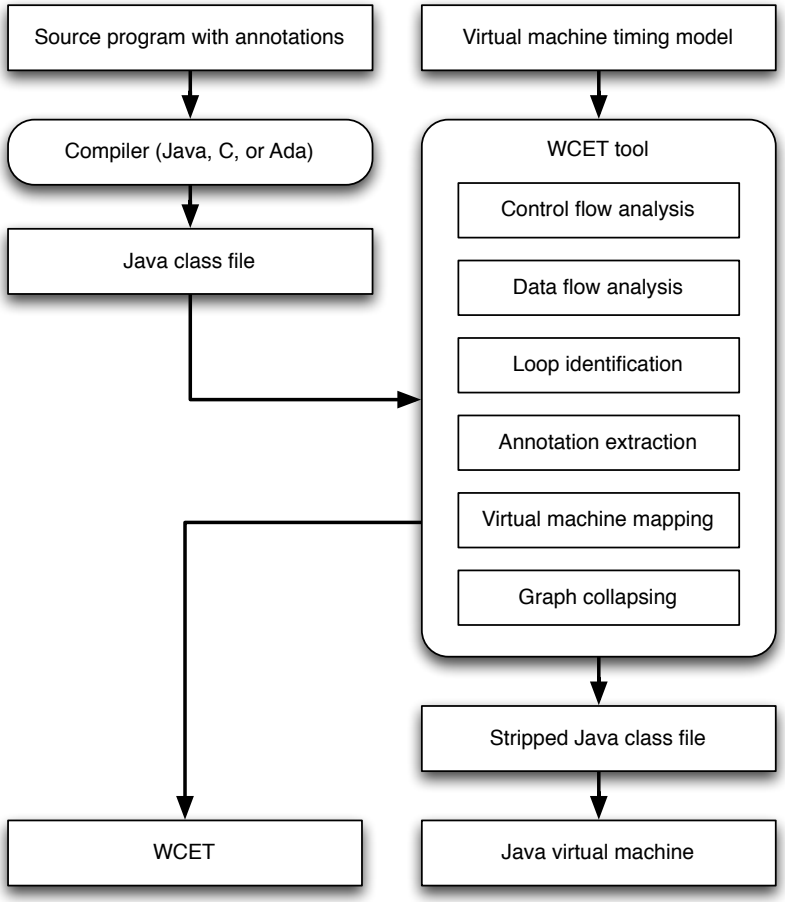

Figure A.1: The basic operation of the Javelin tool.

no guarantee that the resulting Java code can even be compiled.

For these reasons, the notion of bytecode as an intermediate representation for WCET tools has failed to reach industry practice and has not progressed beyond Bernat's initial research.

A.2 High-level Analysis for the Java Language

In contrast to low-level analysis, high-level (meaning above bytecode level) WCET analysis for Java is relatively simpler. The language is similar to existing ALGOL-like imperative languages, so it builds upon a vast body of existing work in compilers and

high-level WCET theory. Research in this area is therefore more mature because it has received attention from a greater number of researchers.

The earliest work in high-level WCET analysis for Java comes from Puschner [265]. He noted that significant effort had been expended on making the *functionality* of code portable, but there were no mechanisms for porting or distributing information about the *execution time* of code. To solve this problem, he centered on the idea of "abstract" timing information. The goal was to collect and store as much information as possible about timing, such as the control flow and loop bounds, without knowing the concrete details of the processor, the cache, and so on. This abstract information could then be ported to any processor, saving the work of running a complete analysis for each target architecture. Figure A.2 provides a diagram of this process.

As a convenient side-effect, this solution also addresses the problem of WCET in third-party libraries. Such libraries make developers more efficient by providing standard functionality—encryption, networking, or graphics processing, for example—in a reusable package. Unfortunately, developers of real-time systems are often cut off from such benefits because these libraries provide no WCET information. End users usually do not wish to perform this analysis themselves, and even if the vendors are willing to perform a WCET analysis and add the necessary source code annotations, they may not wish to expose this code to the outside world. The abstract timing approach advocated by Puschner solves these problems by allowing vendors to bundle WCET information with their code in a portable, reusable format that keeps source code private.

Like Puschner, Hu [266] also developed techniques for making WCET information more portable. Instead of performing an analysis, however, Hu focused on WCET source code annotations, introducing a new format called XAC, or Extensible Annotation Class. Similar in scope to Bernat's WCETAn technique [64], XAC encodes timing

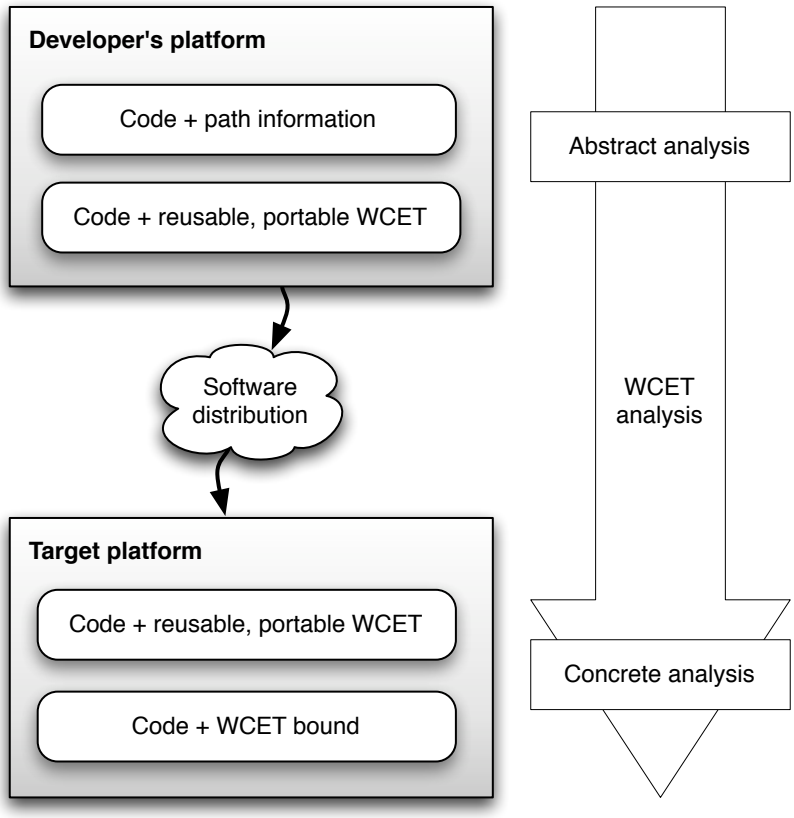

Figure A.2: Puschner proposed using Java bytecode as a layer of abstraction in WCET analysis, whereby the timing information propagates from the abstract to the concrete.

hints as source code comments rather than explicit method calls. This improvement over WCETAn eliminates the relatively complex task of re-writing the bytecode to remove the method calls (in order to eliminate their performance penalty). Hu also specified an extensible file format for bundling WCET annotations with their corresponding class files.

Later that same year, Hu extended his XAC format to handle the problem of dynamic dispatch [66]. Typical in object-oriented programs, dynamic dispatch of method invocations (i.e., polymorphism) is simply disallowed in most WCET tools. Such tools are normally designed for procedural languages, such as C, where dynamic dispatch is much less common. In Java, however, almost every method call requires dynamic

```
class A
{
  //@ Label(A.m1())
  public void m1()
  { ... }
}

class B extends A
{
  //@ Label(B.m1())
  public void m1()
  { ... }
}

A a = new A();
B b = new B();
...
for (int i = 0; i < 5; i++)
{
  if (i == 2)
    a = b;
  //@ 2*UseWCET(A.m1)+3*UseWCET(B.m1))
  a.m1();
}
```

Figure A.3: Hu's Extensible Annotation Class technology addressed the problem of WCET analysis in the presence of polymorphism. Note how the annotation inside the loop body disambiguates the method invocation.

dispatch. Hu addressed this problem by providing new WCET annotation types designed for class hierarchies. For example, the programmer could specify a subset of child classes that are valid for a particular method invocation on a base class (see Figure A.3). This simplistic approach dumps most of the work in the programmer's lap, relying entirely on manual annotations to tighten the WCET bounds of dynamic dispatch.

In stark contrast with Hu's style, Guedes [178] dispensed with annotations altogether, explaining how Gustafsson's "abstract interpretation" technique [170] could be applied to Java. The goal was to remove the need for annotations as much as possible, saving the trouble of having to provide WCET parameters (loop bounds in particular)

in many cases. This very preliminary work was largely theoretical and has not been pursued.

Hunt offered a similar approach for provably correct loop bounds in real-time Java [221, 267]. It combined data flow analysis with formal deductive verification to arrive at a final bound. Although this technique still relies on source code annotations supplied by the developer, Hunt claims that they are much simpler than functional annotations (e.g., loop invariants for correctness proofs) and can be verified for correctness. Figure A.4 shows an example of Hunt's proposed loop annotation syntax.

The remaining work in high-level analysis comes from a European initiative to advance the role of Java in real-time systems. Dubbed HIDOORS (High Integrity Distributed Object-Oriented Realtime Systems) [268], it had many goals: a real-time garbage collector, a graphical UML-based modeling tool, a distributed real-time event manager, and a WCET analysis tool. While the real-time garbage collector has seen new life as part of the Jamaica Virtual Machine [75], the WCET tool never progressed beyond the specification stage.

A.3 Low-level WCET Analysis for Java Bytecode

WCET analysis of bytecode is only a partial solution. Computation of the actual WCET requires low-level analysis that takes into account the particular timing characteristics of a target processor.

Toward that end, Bate expanded on Bernat's work by developing a framework for low-level WCET analysis of Java bytecode [269]. To remain portable among processors, the framework differs from traditional approaches: Instead of calculating the WCET of each basic block (which is impossible at the bytecode level), it calculates bytecode

```
/*@ private normal_behavior
  @ requires legs1!=null && legs1.length>0 &&
  @ legs2!=null && legs2.length>0 &&
  @ i3>0 && i3<legs2.length ;
  @ assignable i1, i2;
  @ ensures true;
  @*/
private void findIntersection(Leg[] legs1, Leg[] legs2, int i3)
{
  i1 = legs1.length - 1;
  i2 = 0;
  int j = 0;
  int k = i3;
  Leg leg;

  /*@ assignable leg, j, k, i1, i2;
    @ decreases (legs1.length - j);
    @*/
  while (j < legs1.length)
  {
    leg = legs1[j];
    if (leg instanceof KFixLeg)
    {
      k = i3;
      /*@ assignable k, i1, i2;
        @ decreases (k + 1);
        @*/
      while (k >= 0)
      {
        if (leg == legs2[k])
        {
          i1 = j;
          i2 = k + 1;
          return;
        }
        k--;
      }
    }
    j++;
  }
}
```

Figure A.4: An example of Hunt's proposed annotations for loop bound verification.

frequencies. When a particular target architecture is known, the frequency vectors can then be mapped to a concrete timing model. Two years later, Bate integrated this approach into a single framework [270] that combines the high-level [64] and

low-level [269] techniques in one vertical package.

Instead of concentrating on a portability solution for WCET analysis tools, Hu targeted the Java platform itself. Citing the growing interest in pure, 100% Java real-time specifications, such as the RTSJ [29] and the Real-time Core Extensions [271],[5] Hu observed that none offered any mechanism for WCET analysis. In addition, existing analysis techniques were exclusive to procedural programming languages, ignoring the dynamic dispatching features of Java.

Hu therefore adapted Bernat's existing WCET framework for the needs of Java, rebranding it the XRTJ (eXtended Real-Time Java) [215]. Essentially a refinement of this existing framework, it added one notable new feature for low-level WCET analysis. Specifically, XRTJ prescribed a measurement-based technique for deriving a timing model of an arbitrary Java virtual machine [272]. This timing model is simply a performance profile, a benchmark of the target processor's ability to interpret bytecodes in the presence of an operating system and virtual machine. The resulting WCET is therefore an estimate and does not provide the hard guarantee of a static analysis. However, it works across all Java systems and requires no modifications to the virtual machine.

Discussion

These efforts are the only published work on low-level WCET analysis for Java bytecode. This raises the question of why other groups have not pursued the same challenge. The explanation is manifold:

- Even today, the concept of real-time Java is relatively new. Reliable implemen-

[5]The RTSJ and the RTCE were two competing real-time specifications for Java. Although they were largely similar, the RTSJ had the support of Sun. As a result, all development of RTCE has ceased, and the J Consortium has disbanded.

tations of the RTSJ, for example, became available only in the last few years. Despite new large-scale projects [60], acceptance of Java for real-time systems is still limited, even among the research community.

- Low-level analysis of bytecode is extremely difficult. Mapping a non-Java language to bytecode is a formidable challenge by itself. In addition, the bytecode must be translated to an arbitrary target architecture, all the while maintaining tight WCET bounds. This requires a detailed analysis to account for pipeline and cache effects, not to mention the overhead of the operating system and the virtual machine. As a result of this complexity, the WCET analysis is often pessimistic, counteracting the benefits that bytecode portability brings.

- The multiple layers of OS, VM, and processor complicate low-level bytecode analysis. One way to mitigate this problem is to adopt a Java-native processor such as Schoeberl's JOP [120] or aJile System's aJ-100 [73]. These processors collapse the vertical stack, removing the OS and VM layers entirely and greatly simplifying low-level analysis. Until recently, however, viable Java-native processors such as these were unavailable, making them even less prevalent in the research community than real-time Java. In addition, restricting low-level analysis to these processors limits the portability, and thus the acceptance, of any analysis technique.

These issues have substantially slowed research in low-level bytecode analysis.

A.4 WCET Analysis for Java-specific Processors

Modern CPU architecture is the WCET researcher's worst nightmare. Large pipelines, branch prediction, and sophisticated multi-level caching have greatly improved aver-

age throughput, but not without cost. Providing a tight guarantee on worst-case execution time is horrendously difficult on these superscalar processors.

As a result, new processor architectures have emerged that are designed specifically for real-time systems, making them an easier target for WCET analysis. An example from the Java domain is JOP, or Java Optimized Processor [120], a WCET-aware CPU that executes bytecode natively without the need for an OS or virtual machine.

JOP offers three key features that allow bytecode execution time to be predicted tightly:

- JOP translates bytecodes into microcode instructions, each of which executes in a single cycle. And because there are no dependencies between bytecodes, calculating WCET of basic blocks is a simple matter of summing the cycle count of each bytecode.
- JOP has a short four-stage pipeline, allowing branch prediction logic (which complicates WCET analysis) to be discarded with minimal performance loss.
- JOP provides a unique instruction cache specially designed for WCET analysis in Java. It is based on the observation that no branch instructions in Java jump outside of a method; therefore, the "method cache" in JOP [135] is based on whole methods rather than small cache lines. Consequently, hit and miss detection occurs only during method invocation and return, allowing WCET analysis of the cache to be ignored entirely during the execution of individual methods.

To demonstrate JOP's ability to yield tight WCET bounds, a basic IPET-based analyzer called WCA was created [135]. It understands JOP's method cache architecture

and supports WCET analysis across method invocations. (Section 3.4 describes the JOP in more detail, and Section 5.1.2 explains its method cache.)

In other work on Java-specific processors, Chai [273] developed a technique for pre-processing class files that were destined for embedded systems. Noting that such systems normally prohibit dynamic class loading and garbage collection, Chai relies on these assumptions to replace certain bytecodes with "optimized" variants. These altered bytecodes exchange flexibility for fewer cycles per instruction, leading to a reduced WCET. As such, Chai's proposal is better described as a speed optimization rather than a WCET analysis technique.

A.5 Other Work in WCET Analysis for Java

Portability, low-level analysis, and high-level analysis are where most WCET research for Java has been applied. This section presents work in WCET analysis that does not fall cleanly into one of these three main categories.

Persson describes a development environment for real-time Java that incorporates WCET information [206]. Called Skånerost, it displays the WCET of a particular method in the margins of its source code editor. The WCET value is updated continuously, as the source code changes, to provide feedback to the developer, as shown in Figure A.5. (Persson does not describe exactly how this WCET value is obtained; an analysis tool and the appropriate annotations are assumed to be available.)

Hu developed a "gain time" reclamation framework for hard real-time Java [274]. Based on the assumption that real-time tasks often do not follow the worst-case path at run-time, the goal is to reclaim this "gain time" by detecting when a task has completed before its predicted worst-case time. A lower-priority task can then be

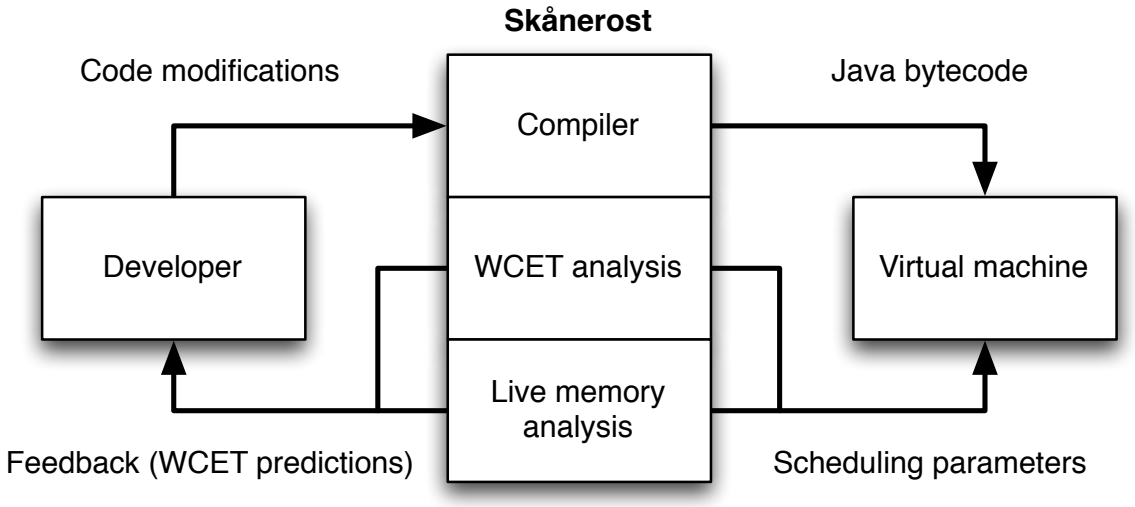

Figure A.5: The flow of WCET information in the Skånerost system puts the developer "in the loop" to support interactive WCET analysis.

executed, increasing overall CPU utilization. The novelty of Hu's approach lies in the ability to track object types as they change (via a so-called Object Type Lifetime Graph), thus yielding tighter WCET bounds than would otherwise be possible in dynamic dispatch languages like Java.

Corsaro addressed the problem of obtaining tight WCET bounds on memory allocations in Java [275]. Borrowing principles from UNIX file systems, the approach gains predictable allocation time at a cost of wasted space. The basic idea is to permit fragmentation of memory chunks. Allocation and deallocation of the chunks can then be accomplished in linear time, thereby improving the WCET of memory operations.

Finally, Lei focused on tightening the WCET of RTSJ's asynchronous transfer of control (ATC) mechanism [276]. Conventional ATC implementations rely on a recursive procedure to locate the appropriate catch class, making WCET analysis difficult. Lei solves this problem by performing class resolution and linking at compile-time rather than run-time (if certain assumptions about the run-time environment can be made). ATC then reduces to a simple comparison, and its WCET is more predictable.

A.6 Conclusion

Given that the entire collective work in WCET analysis for Java can be summarized so briefly, much remains to be done. Most research projects have not progressed beyond the prototype stage, and several open problems persist:

- How can dynamic dispatch be handled in an automatic way (without a total dependency on manual annotations [66])?
- Given the difficulty of performing a fully static analysis on the Java stack (including OS and VM layers), could measurement-based or probabilistic approaches be a sufficient replacement?
- How can garbage collection latencies be incorporated into a WCET analysis?

Performing WCET analysis in the presence of dynamic dispatch, garbage collection, and multiple layers of abstraction is still very much an open problem. While it may be possible to adapt existing commercial tools for C, such as RapiTime, for measurement-based analysis of Java, the best solution remains unclear, and many challenges still lie on the horizon.

B WCET Annotations in Java

WCET analysis is far from trivial. The analysis algorithm must determine not only *when* a program finishes but also *whether* it will ever finish at all. This problem of non-termination has been known since 1936 when Alan Turing proved that, given an arbitrary program, a decision as to whether it will finish or will run forever does not exist [169].

```
1  public class BubbleSort
2  {
3      public static void main(String[] args)
4      {
5          int n = Integer.parseInt(args[0]);
6          double a[] = new double[n];
7
8          // Fill the array with random numbers
9          for (int i = 0; i < n; i++)
10             a[i] = Math.random();
11
12         BubbleSort.sort(a);
13     }
14
15     // The standard bubblesort algorithm
16     private static void sort(double[] a)
17     {
18         // @WCET loopMax=9
19         for (int i = 0; i < a.length - 1; i++)
20         {
21             // @WCET loopMax=9
22             for (int j = 0; j < a.length - 1 - i; j++)
23             {
24                 if (a[j + 1] < a[j])
25                 {
26                     double tmp = a[j];
27                     a[j] = a[j + 1];
28                     a[j + 1] = tmp;
29                 }
30             }
31         }
32     }
33 }
```

Figure B.6: WCET tools require knowledge about the constraints under which a program will run. Here, the developer "knows" the input size will never be larger than 10 and has inserted WCET annotations accordingly.

The consequence for WCET analysis is that no tool can examine an arbitrary real-time Java program and derive its worst-case bound. As shown in Figure B.6, the WCET may depend on knowledge that only exists at run-time, stifling any attempt at static analysis.

A bound on WCET may be undecidable in general, but real-time systems are hardly the arbitrary programs described by Turing. In a real-time environment, the developer

is normally able to (and, in many cases, must) exercise careful control over input parameters and program complexity. As a result, certain assumptions can be provided to a WCET analysis tool that make its job tractable. These assumptions usually take the form of *loop annotations*.[6]

For example, turning again to Figure B.6, the developer knows something about the run-time environment and is able to make an assumption: The maximum array size (that is, the `args[0]` parameter) for any execution of this program will never be larger than 10. The developer was thus able to *annotate* the source code on lines 18 and 21 with the maximum possible iteration of each loop. By propagating this special knowledge from developer to analysis tool, annotations provide much tighter WCET bounds than would otherwise be possible.

B.1 Prior Work in WCET Annotations for Java

Given that WCET annotations have been used for many years in real-time C and Ada software, one might expect a standard, or at least a *de facto* convention, to materialize. In reality, a number of competing styles of annotation have been developed for real-time Java. Each one embodies slightly different mechanisms for syntax, storage, and specification, making them all mutually incompatible.

This section provides a brief survey of these annotation styles for Java. It includes only the research literature and does not consider commercial tools and other annotation styles that may be found in industry.

The earliest work in WCET annotations for Java comes from Bernat, who proposed a portable WCET analysis tool [64]. In this case, *portable* refers to language portability:

[6]Loop annotations are just one of many different types of WCET annotations. They belong to a broader category known as *program semantics* annotations [216] or *program-specific* annotations [217], which include recursion bounds, variable value restrictions, and others.

Table B.1: A sample of WCET annotation styles in real-time Java

Tool	Examples	Comments

WCETAn [64]	`WCETAn.Loopcount(20);`
	`WCETAn.Define_Path(l);`
	`WCETAn.Scope S =`
	`  new WCETAn.Scope();`
		Implemented as method calls instead of comments for binary portability.

XAC [266]	`//@ Loopcount(100)`
	`//@ Mode(Quick_Mode)`
	`//@ UseWCET(`
	`  AirTempSen.AccessSensor(V))`
		Traditional comment-based mechanism. Has been extended to handle polymorphic method calls.

Skånerost [206]	`/*$ loop-bound 100 */`
	`/*$ time-bound 25ms */`
	`/*$ path-bound 10 */`
		Departs from the popular convention of using `@` as the start token in annotations, opting for `$` instead.

WCA [135]	`//@WCA loop=10`	Currently supports loop bound constants only.

The tool was designed for analyzing Java, C, Ada, and any other language that could be translated to Java bytecode. To achieve such portability, the source code must invoke methods in a predefined class whenever a WCET annotation is required. (See Table B.1 for an example.) Compared to traditional annotations, this style mingles non-functional metadata—that is, the WCET information—with the normal source code statements, making programs more difficult to read.

The XRTJ project [215] implemented WCET annotations in a more traditional way. All annotations appear as comments with the characters `//@` for single lines and `/*@ ... @*/` for multiple lines [266]. The XRTJ compiler parses these lines and writes them to an XAC (Extensible Annotation Class) file, an XML-like text file that is paired with its class in a real-time Java program. The XRTJ analyzer then reads each XAC file to determine loop bounds, timing modes, and other details necessary to derive the WCET. Hu later extended the XAC format to capture dynamic dispatch (i.e., polymorphism) semantics [66]. XRTJ is also notable as the first research project

to suggest the idea of embedding annotations in Java class files [274].

The Java development environment Skånerost also relies on annotations to obtain information that is difficult for a tool to deduce but often obvious to a programmer. As usual, the annotations are expressed as source code comments adhering to a special syntax that can be identified by an analysis tool but ignored by the Java compiler. Skånerost's approach differs from most others by providing annotations not only for loop bounds and path constraints but also the size and shape of data structures. This information is vital to the live memory analysis of Skånerost's real-time garbage collector.

In more recent work, Schoeberl and Pedersen implemented a WCET analyzer for JOP [135]. This tool introduced yet another syntax for annotations. Although it still used the same `//@` starting token, the syntax of the annotations differed from the XRTJ project, making the tools incompatible with each other.

Despite these prior research efforts, neither of the industry standards for real-time Java, RTSJ [29] nor RTCE [271],[7] provides any mechanism whatsoever for annotations. The lack of support for annotations is puzzling, especially given the importance of WCET as outlined in Section 2.1.

B.2 A Lack of Standards

Clearly, no single convention for WCET annotations has taken hold, resulting in a contradiction of Java's mantra: "Write once, run anywhere." Real-time Java programs designed for one WCET analysis tool must be rewritten for other tools. The absence of a single annotation standard also makes the tools themselves more difficult to implement. Even if they all agreed on the same starting token (e.g., `//@`), several

[7] RTCE is now defunct, supplanted by the RTSJ.

unresolved issues remain:

- What are the syntax and semantics for the string following the starting token?
- After an annotation is parsed, where is the information stored, and how do lower-level tools retrieve it?
- Is there a formal specification to ensure compatibility? Who ratifies it and who maintains it?

The end result is that each new WCET analysis tool must reinvent the wheel when it comes to annotations. Even if a common syntax were chosen, there are no open, reusable libraries for parsing, storing, and extracting the annotations. For instance, a high-level WCET analysis tool may write annotation data to a file format that a low-level tool cannot understand. These obstacles prevent interoperability among tools, lengthen their development cycles, and impede the overall progress of research into real-time systems.

B.3 A Standard for WCET Annotations in Java

In the non-real-time Java domain, a similar situation had already transpired. The Java Modeling Language [250], the ESC/Java2 [277] static checker, XDoclet [278], and various other software offered custom, incompatible annotation systems for Java source code. In addition, Java's frameworks for server software development, such as Enterprise JavaBeans [279], had become increasingly complex due to an explosion of metadata. Deploying components to a web server, for example, required maintenance of separate (and unwieldy) XML files to describe remote interfaces, database-to-object mappings, and so on.

To solve both problems, the Java community proposed a standard framework for annotations. The basic idea was to encapsulate common code patterns into single statements—annotations—embedded directly into the source code. Rather than manage source code and metadata separately, taking pains to keep them in sync, programmers could attach annotations directly to the source code constructs they describe. This approach increases the power and convenience of annotations. It enables, for example, the canonical getter/setter methods:

```
private int sensorReading;

public int getSensorReading() {
  return sensorReading;
}

public void setSensorReading(int sensorReading) {
  this.sensorReading = sensorReading;
}
```

to be condensed to:

```
@property int sensorReading;
```

This style is a significant shift because, unlike the techniques described in Section B.1, the annotations are no longer jury-rigged as comments. Instead, they are first-class objects in the Java language. They can have parameters and must conform to type-checking rules. Such a substantial change to the language, which also demanded special compiler support, led to formal review under the Java Community Process (JCP): a public, cooperative system for adopting new technologies as official Java specifications.

The proposed annotation framework was submitted to the JCP as JSR-175[8] and met final approval in September 2004. That same month, Sun released Java 5, Standard Edition, which included support for JSR-175 annotations. A new edition of the Java Language Specification [280] was published the following year to formalize these annotations and ensure that tools and libraries using them would remain compatible with each other. Today, annotations are a standard, well-defined part of the Java platform.

Storing Java Annotations

One of the advantages of Java annotations,[9] and a key departure from many existing WCET annotation frameworks, is that the annotation data is stored directly in class files. Embedding annotations within classes simplifies code management because a separate file format for metadata need not be maintained. Also, existing mechanisms for storing, distributing, and deploying class files can readily be used for annotation data. For instance, a build script that packages a class library into a JAR file (Java ARchive) will automatically package annotations as well.

The annotation standard achieves this simplicity by building upon an existing Java mechanism for bundling metadata with class files. Known as *class file attributes*, this mechanism has been part of the Java virtual machine specification since its inception. It defines an area at the end of the class file for storing any kind of structured data, such as the line number table, the source file name, and even the Java bytecode itself. (For a complete description of class file attributes, refer to Section 4.7 of the Java Virtual Machine Specification [281].)

[8] JCP proposals begin life as a numbered Java Specification Request, or JSR, and are available from `http://jcp.org/`.

[9] From this point forward, "Java annotations" refers to the JSR-175 standard.

Java compilers and other code generators are permitted to emit class files containing new attributes, and Java virtual machine implementations are prohibited from refusing to load class files simply because of the presence of some new attribute. Thus, class file attributes are *extensible* and may support new attributes at any time without sacrificing backward compatibility.

The JSR-175 specification takes advantage of this extensibility by using it to store annotations. As illustrated in Figure B.7, the spec defines several new class file attributes for holding annotation data. It reserves their names to avoid conflicting with other attributes, and it formalizes their structure so that third-party tools know how to access them. For instance, the RuntimeInvisibleAnnotations attribute represents annotations that Java's reflection API should ignore.

Creating Java Annotations

Conveniently, programmers need not write to these attributes directly. The Java compiler generates them automatically when compiling Java source code containing annotations. For example, Java 5 includes a few built-in annotation types such as SuppressWarnings:

```
@SuppressWarnings({"deprecation"})
public void myMethod() { ... }
```

Compiling this code results in a class file with a RuntimeInvisibleAnnotations attribute pointing to the SuppressWarnings annotation type. As expected, compilers will not print deprecation warnings for methods annotated with this type. Furthermore, the "invisible" specifier tells virtual machines not to load the attribute into memory because it is useful only at compile time.

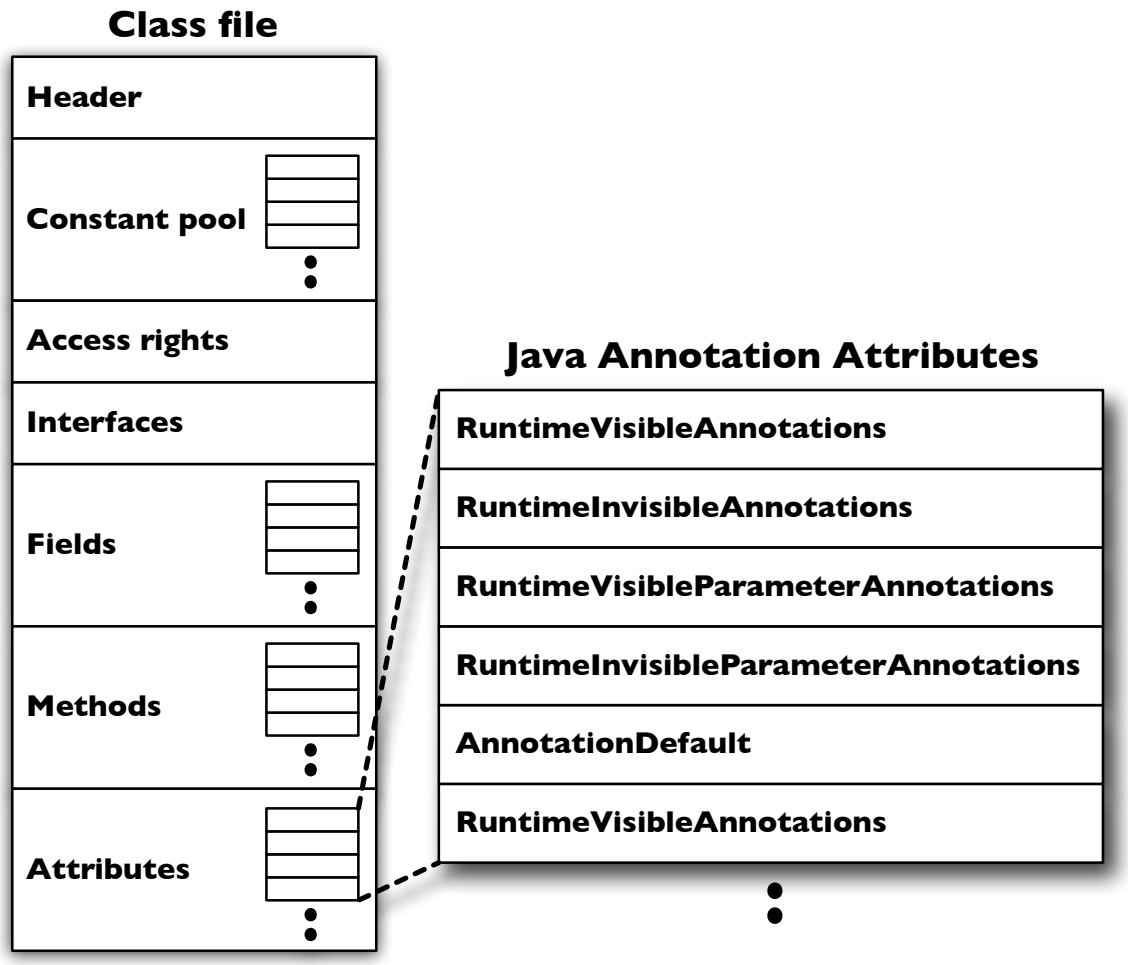

Figure B.7: This diagram of Java's class file format shows how JSR-175 annotations are stored as attributes at the end of the file.

Custom annotations can be created easily, as well. Imagine a real-time control system that must respond to a sensor change and actuate a motor within fifty milliseconds. This requirement could be encoded as an annotation, perhaps called MaxAllowableWCET, and attached directly to the very method that handles the response. WCET tools analyzing the method could then print a warning if the calculated WCET exceeds the specified WCET.

Creating this sort of annotation requires defining an *annotation type* that resembles a traditional Java interface:

```
@Target(ElementType.METHOD)
public @interface MaxAllowableWCET {
  double seconds();
}
```

Here, the seconds method declares a parameter for the annotation type, while the Target annotation (actually a meta-annotation, since it annotates an annotation) tells the compiler that this annotation type is only permissible on method declarations.

After compiling the annotation type, using it is a simple matter:

```
@MaxAllowableWCET(seconds=0.05)
public void actuateMotor() { ... }
```

Reading Java Annotations

Tools such as WCET analyzers need to read the annotations programmers have written, and Java provides a facility for this. Java 5 includes a new package, java.lang.annotation, and an expanded Reflection API to access annotations. Figure B.8 shows a simple example of how MaxAllowableWCET annotations can be detected using this API. (For the example to work, the MaxAllowableWCET annotation type must have a retention policy of RUNTIME so that it is visible to the virtual machine. See Table B.2 for a list of retention policies.)

In addition to this built-in mechanism, a number of third-party frameworks have been developed that can assist in reading and manipulating Java annotations: Annogen [282], ASM [137], and Javassist [283], to name a few. All of these work with the

```
import java.lang.reflect.*;

public class CheckWCETAnnotations
{
   public static void main(String[] args) throws Exception
   {
      for (Method m : Class.forName(args[0]).getDeclaredMethods())
      {
         if ( !m.isAnnotationPresent(MaxAllowableWCET.class) )
            System.out.println("Error: A WCET annotation is missing.");
      }
   }
}
```

Figure B.8: Annotations are stored as low-level class attributes, but high-level tools can read them using Java's Reflection API. This example code uses the API to verify that all methods in a given class are annotated with the MaxAllowableWCET type described in Section B.3.

Table B.2: Java annotation retention policies

Policy	**Description**
SOURCE	The annotation is not stored in the class file; it is simply discarded at compile-time.
CLASS	The annotation is stored in the class file but is not loaded into memory.
RUNTIME	The annotation is stored in the class file and loaded into memory at run-time so that it is visible to Java's Reflection API.

same Java annotation standard, so unlike the current situation with WCET analysis, annotations created by one tool can be accessed by another.

Weaknesses of Java Annotations

While standardization, portability, and validation are the greatest strengths in Java's annotation facility, it also has some serious drawbacks. Chief among them is a restriction on where annotations can be placed in the source code: They can act only as declaration modifiers. They cannot annotate method calls, loops, and other program

elements. For example, the following code is illegal:

```
@LoopBound(max=10)
while (!bufferEmpty) { ... }
```

An explanation for this restriction comes from Appendix II of the JSR-175 specification:

> ***Why can't you annotate arbitrary program elements such as blocks and individual statements?***
>
> *This would greatly complicate the annotation syntax: We would have to sacrifice the simplicity of saying that annotations are simply modifiers, which can be used on declarations.*

In other words, bolting a totally new annotation mechanism onto an already mature and established language like Java is no small task. It requires extensive cross-cutting changes to the virtual machine, class file format, and compiler. Restricting the scope of the changes to declarations made the implementation tractable and allowed Java to support a limited annotation syntax immediately after the proposal's ratification, rather than wait years for a fully functional and comprehensive mechanism to evolve.

Another limitation is that Java's Reflection API can only read annotations that have been stored in class files. Source code with annotations must first be compiled before the annotations can be read. A more serious implication is that if an annotation type has a SOURCE retention policy, it cannot be accessed at all without third-party tools that are able to parse Java source code.

The Java community has recognized these weaknesses and responded with a series of proposals submitted to the JCP. For example, JSR-269 [284] provides a standard API

for interacting with annotation processors such as Annogen. It also provides an interface for processing annotations in source files. The proposal was approved and implemented in Java 6.

Another proposal, JSR-308 [285], allows annotations to be placed where types are used, not just where they are declared. For instance:

```
myString = (@NonNull String) myObject;
```

Another example:

```
int size() @Readonly { ... }
```

This relaxation allows verification tools and defect finders to work with standard Java annotations. The proposal has been approved and will likely be implemented in Java 7.

Table B.3 summarizes these current and former proposals for improving Java's annotation facility. Some have already been implemented in Java 6, while others have been approved but not yet scheduled for implementation until Java 7.

B.4 Applying Java's Annotation Standard to WCET

Despite the progress in enabling Java annotations to support modeling and checking tools, there has been virtually no attention given to making them suitable for WCET analysis tools. The lack of emphasis is unfortunate because, compared to the existing comment-based systems, Java's built-in annotations offer an attractive alternative. In particular:

Table B.3: Proposed specifications for improving Java's annotation facility

Proposal ID	Description
JSR-181	Defines a set of annotations for web services. The proposal was implemented in Java 6.
JSR-250	Defines a set of annotations for general use. The proposal was implemented in Java 6.
JSR-269	Provides a standard API for interacting with annotation processors such as Annogen. Also provides an interface for processing annotations in source files. The proposal was implemented in Java 6.
JSR-303	Defines a set of annotations for validating JavaBeans. The proposal was approved and will likely be implemented in Java 7.
JSR-305	Defines a set of annotations for defect-detection tools such as FindBugs. The proposal was approved and will likely be implemented in Java 7.
JSR-308	Allows annotations to be placed where types are used, not just where they are declared. For example: myString = (@NonNull String)myObject; The proposal was approved and will likely be implemented in Java 7.

Standards-based Java's annotation mechanism solves the standards issues raised in Section B.2.

Validation Unlike most comment-based systems, Java annotations are highly structured and type-safe. For instance, the Java compiler can guarantee that an annotation designed for methods is not accidentally used to annotate other source code constructs.

Convenience With WCET annotations implemented as Java annotations, timing information and code can be bundled into the same class file. This simplifies code management and distribution. For example, a class library for real-time systems can include timing information in the library itself. No extra files and no new file formats are necessary.

Tool support Because Java's annotation mechanism is relatively mature and ships with every copy of the Java run-time, it enjoys broad support from a variety of tools and APIs for manipulating annotations. WCET analysis tools are therefore easier to implement since they can be built upon these existing mechanisms.

Comment-based WCET annotations described in the literature do not have these features due to their proprietary nature and usually *ad hoc* implementation. WCET analysis tools should instead adopt Java's annotation standard. Doing so would likely shorten the development cycle of such tools, leading to more prototypes and an overall improvement in real-time research.

Unfortunately, Java's standard is not yet a drop-in replacement for existing annotation frameworks. One of the issues is a concern for embedded systems. (Embedded systems are not directly related to WCET analysis, but real-time systems are often embedded, so the domains overlap.) These systems normally have stringent resource requirements, thus the Java class files deployed on them must be as small as possible. However, Java annotations are stored in these class files, taking up valuable storage space even though the annotation information is only required at design time.

One way to solve this problem is to change the retention policy of the annotation to SOURCE. This prevents the annotation from being stored in the class file, but it also destroys the benefit of bundling annotation data with the class. A preferable workaround is to leave the annotation at the default CLASS retention level but use a tool to strip the annotations from the class file before deploying it to the embedded device. This removal bears no risk to violating WCET guarantees because, unlike other approaches [64], the annotations are stored as class file attributes, not in the executable code itself.

A more critical failing of Java's annotation standard cannot be mitigated so easily. Namely, the restriction on where annotations can be placed is a major impediment toward using standard Java annotations in WCET tools. As a result, none of the existing WCET annotation styles listed in Table B.1 can be fully translated to the current system of Java annotations.

In certain scenarios, however, an approximation is possible. For example, Figure B.9 shows a test method from the JOP project [135]. The original comment-based annotations have been removed and subsequently replaced with the following Java annotation type:

```
public @interface LoopBound
{
    int max();
}
```

As a consequence of this change, the loop bounds no longer appear adjacent to the for loop constructs but rather at the loop variable declarations (lines 3, 8, and 14). A WCET analysis tool can still run successfully with these modifications, but restricting loop bounds to variable declarations means that the bound may be placed far from the loop itself. The artificial constraint is also unintuitive because it forces programmers to write loop constructs in an atypical and awkward style.

Other researchers, working independently, have encountered the same obstacles in Java annotations. In fact, JSR-308 was born as a result of these very obstacles. Its primary goal was to support annotations on types, but its secondary goal was to loosen other restrictions on where annotations may be placed, thereby enabling new types of analysis tools. Ultimately, the proposal did not relax them far enough for the purpose of WCET analysis tools, which would require, at a minimum, placement

```
1  public int measure(boolean b, int val)
2  {
3      @LoopBound(max=10) int i;
4      for (i = 0; i < 10; i++)
5      {
6          if (b)
7          {
8              @LoopBound(max=4) int j;
9              for (j = 0; j < 4; j++)
10                 val *= val;
11         }
12         else
13         {
14             @LoopBound(max=7) int k;
15             for (k = 0; k < 7; k++)
16                 val += val;
17         }
18     }
19
20     return val;
21 }
```

Figure B.9: This altered program segment from JOP's test suite shows how WCET annotations might be implemented without changing the current Java standard.

of annotations directly on loop constructs (if, while, and do/while).

Although Java annotations are not currently suitable as WCET annotations, the standard is not far from becoming a virtually ideal replacement for the existing approaches presented in Table B.1. Two changes are necessary for this to happen.

First, the standard should be modified so that annotations can be placed on loops and basic blocks. JSR-308 has already moved the standard in this direction. Indeed, the original proposal commented that such a change would be useful for concurrency and atomicity (although it made no mention of WCET analysis). Furthermore, members of the Expert Group who voted in favor of the proposal agreed that relaxing the standard in this manner would be beneficial. For instance, Intel Corporation added the following comment to its vote:

This note confirms our understanding that the scope of the JSR includes providing for annotations on loops and blocks if the Expert Group decides to include that after evaluation. The JSR itself should be updated to make it clear this is in the scope.

Nortel also added:

Completely aligned with the comments by Intel, it would have been extremely useful to extend the annotations.

Second, type definitions and a naming convention for WCET annotations must be established. Java annotations would do little good if one tool recognized `@LoopBound` while another tool expected, say, `@LoopMax`. Establishing these conventions should be straightforward. JSRs 181, 250, 303, and 305 have already walked this path for other domains; a proposal for WCET annotations would merely follow in their footsteps.

B.5 A Java Compiler for WCET Annotations

Until such changes have been formally proposed and actually implemented in the official Java platform, the Volta project provides an interim solution. It includes a fully functional and ready-to-use Java compiler, based on Sun's OpenJDK source code [286], that allows annotations on loop statements. With this modified `javac`, the following code becomes legal:

```
@LoopBound(max=100)
for (int i = 0; i < percent; i++) { ... }
```

The new annotation support requires no changes to the virtual machine or class file format, only some compiler modifications and a few new class file attributes. In effect, it changes Java's language grammar for loop statements to the following non-ambiguous production:

```
LoopStatement:
  Annotations LoopStatement
  ... // current body of ''LoopStatement'' grammar production
```

The 25 kilobyte patch to javac to allow this production touches the relevant syntax tree classes: JCWhileLoop, JCEnhancedForLoop, etc. It also changes the compiler to emit class file attributes that describe the statement annotations: RuntimeVisibleStatementAnnotations and RuntimeInvisibleStatementAnnotations. These attributes are attached to the class's Code attribute and are identical to the existing RuntimeVisibleAnnotations and RuntimeInvisibleAnnotations attributes, except that they include one additional field:

```
u4 pc;  // Code offset to the start of the loop
```

This field uniquely identifies the annotation and indicates to which loop it refers. Tools can retrieve its value by loading the class annotations using BCEL or a similar utility. For example, the Clepsydra tool from the Volta project includes a built-in annotation reader for accessing this style of annotations for WCET analysis.

To verify that the modified compiler works correctly, Volta also includes a utility for dumping a class's annotations to the console. It displays them in the same struct-like format as the Java virtual machine specification, making comparison to other annotation formats easier. For instance, the utility reveals that compiling the code

```
RuntimeInvisibleStatementAnnotations_attribute {
    u2 attribute_name_index = 30 (RuntimeInvisibleStatementAnnotations);
    u2 attribute_length = 15;
    u2 num_annotations = 1;
    annotations =
        {
            u2 type_index = 31 (LLoopBound;);
            u4 pc = 3;
            u2 num_element_value_pairs = 1;
            element_value_pairs =
                {
                    u2 element_name_index = 32 (max);
                    element_value value = {
                        u1 tag = 73 (I);
                        u2 const_value_index = 8 (64);
                    }
                }
        }
}
```

Figure B.10: The Volta project includes a utility for viewing the structure of Java annotations. Here, the utility reveals the structure of the annotation from Figure 4.6.

from Figure 4.6 would produce the statement annotation given in Figure B.10.

B.6 Conclusion

Java annotations in their present form are not suitable as a full replacement for existing WCET annotation frameworks. However, proposals such as JSR-308 indicate that the Java platform is evolving to a point where such annotations will eventually become possible. They could then be integrated into existing real-time Java frameworks, all of which would benefit from improved WCET analysis. The potential advantages, including syntax checking, type safety, and tool support, are too great to ignore.