FAKULTÄT FÜR !NFORMATIK

Worst-Case Execution Time Analysis for Real-Time Java

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Benedikt Huber
Matrikelnummer 0060387

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: o.Univ.-Prof. Dipl.-Ing. Dr.techn. Herbert Grünbacher
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Martin Schöberl

Wien, 01.03.2009

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Benedikt Huber
Vitusgasse 8/10
A-1130 Wien

"Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit einschließlich Tabellen, Karten und Abbildungen , die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Benedikt Huber Wien, 23. März 2009

Abstract

Real-time systems are applications which have to meet time constraints, ensuring that periodic tasks are scheduled in time, and events are handled within a permitted delay. In order to show that the system behaves correctly, it is necessary to verify that tasks complete within a given time span. Worst Case Execution Time (WCET) analysis, the static prediction of the time needed to execute a task, therefore plays a central role for safety-critical real-time systems.

In this thesis, we discuss the WCET analysis of real-time Java applications, and present a tool analyzing tasks executed on the Java Optimized Processor (JOP). JOP is an implementation of the Java Virtual Machine in hardware, and uses a cache fetching all instructions of a method at once, which needs to be taken into account.

Two techniques for calculating a WCET bound have been implemented, the *Implicit Path Enumeration Technique* (IPET), translating the calculation to a maximum cost circulation problem, and a dynamic approach using the timed automata model checker UPPAAL. The model checking based approach is computationally more expensive, but capable of modeling timings which depend on the execution history, and therefore has the potential to yield more accurate WCET bounds than the static IPET method. JOP's variable block method cache was included in both timing models, reducing the gap between the actual and the calculated WCET.

Both approaches have been integrated in the analysis tool, and we are thus able to directly compare analysis times and the quality of the calculated WCET bound. Experimental results indicate that timed automata model checking using UPPAAL is at least suitable for the analysis of single methods and smaller tasks, although large loop bounds lead to a significant increase of the time needed for the analysis. IPET on the other hand scales very well, but delivered slightly worse results for the cache approximation. As the model checking approach is not established yet, there seem to be plenty of opportunities for optimizations and new applications, leaving room for future research.

Kurzfassung

Die Korrektheit eines Echtzeitsystems ist nicht nur von der Programmlogik, sondern auch von zeitlichen Faktoren abhänging. So muss beispielsweise ein periodisch aufgerufener Programmteil rechtzeitig ausgeführt und auf Ereignisse innerhalb einer festgelegten Frist reagiert werden. Um die Einhaltung dieser Anforderungen zu garantieren, ist eine *Worst Case Execution-Time* (WCET) Analyse notwendig, um festzustellen, wieviel Zeit die Ausführung eines Programmfragments benötigt. Für sicherheitskritische Systeme ist man im Besonderen an der Berechnung einer sicheren oberen Schranke der maximalen Laufzeit interessiert.

Diese Diplomarbeit behandelt die Laufzeitanalyse der sequentiellen Programmteile sicherheitskritischer *Java* Echtzeitanwendungen. In diesem Rahmen wurde eine Applikation zur Laufzeitanalyse für den Java Prozessor *JOP* entwickelt und zwei unterschiedliche Techniken zur Berechnung der maximalen Laufzeit implementiert. Zum einen die häufig angewandte Technik der impliziten Pfadenumeration (*Implicit Path Enumeration Technique* (IPET)), welche die Laufzeitberechnung in ein Netzwerkfluss-Problem überführt. Der zweite Ansatz modelliert Programme als ein Netzwerk von *timed automata*, eine Erweiterung endlicher Automaten um zeitabhängige Systeme zu modellieren. Dabei wird der *model checker* UPPAAL verwendet, um im durch die Automaten implizit gegebenen Zustandsraum nach dem Ausführungspfad mit der maximalen Laufzeit zu suchen.

Die Berechnung mit Hilfe von UPPAAL ist zwar teurer, ermöglicht aber im Gegenzug, Abhängigkeiten des Zeitverhaltens vom bisher aufgeführten Pfad zu modellieren, und damit eine genauere Approximation der maximalen Laufzeit zu erhalten. In beiden der oben skizzierten Ansätze wurde JOP's *Methoden Cache* berücksichtigt, um die Schranke zu verbessern.

Da beide Techniken in dem Analysewerkzeug integriert wurden, konnten sie direkt miteinander verglichen werden. In den durchgeführten Experimenten stellte sich der *Timed Automata* Ansatz als für kleinere Programme geeignet heraus, wobei allerdings eine hohe Zahl an Schleifeniterationen die Analysezeit signifikant verlängerte. Die IPET Methode wiederum skalierte sehr gut, lieferte aber im Gegenzug etwas schlechtere Ergebnisse mit der von uns verwendeten Cache-Approximation. Für die Berechnung via *model checking* scheinen noch zahlreiche Optimierungen möglich, um die Analysezeit zu verringern. Außerdem ergeben sich hiermit weitere, neue Anwendungsgebiete, wie etwa modell-basierte *Scheduling* Analyse, so dass es lohnenswert erscheint, diesen Ansatz auch in Zukunft weiterzuverfolgen.

Contents

Chapter 1

Introduction

For safety-critical real-time systems, one not only wants to minimize the risk for errors due to faulty program logic, but also prove that no time constraints will be violated at runtime. This is only possible if there are means to compute an upper bound for the time needed to execute one task. As the execution time is influenced by a multitude of factors, such as external input and internal processor state, it is often intractable to obtain a safe bound using measurements only. Static analysis techniques for predicting the Worst-Case Execution Time (WCET) on the other hand provide an estimate which is known to be safe. Because resources are limited, a wasteful over-approximation of the WCET is unlikely to be tolerated. So not only the safety is important, but also minimizing the difference between the actual worst-case execution time, and the one calculated statically.

The timing analysis is usually divided into two phases: obtaining a timing model of the processor and the application, and calculating the WCET from those models. A simple processor timing model simplifies the calculation, as it is often impossible to precisely simulate features of modern processors, such as out-of-order execution and branch prediction. Nevertheless, instruction and data caches for example are quite common, and need to be taken into account. A suitable representation of the necessarily finite set of execution paths has to be extracted from the program code, which is only manageable if the language is restricted to an analyzable subset, possibly extended with annotations providing additional flow information.

It is usually intractable to explicitly enumerate all execution paths, so more efficient techniques for calculating the WCET are needed. The simplest possibility is to operate on the syntax tree of the program, and calculate the execution time recursively. This method has several shortcomings, not supporting arbitrary control flow or most forms of timing and execution dependencies. A more general solution is to label the control flow graphs with execution costs and execution frequency constraints, and view the WCET calculation as a network flow problem. Another,

quite different approach is to model the execution path set using finite automata, and explore their state space searching for the worst-case path. The latter two approaches will be discussed, and have been implemented for the WCET analysis of real-time Java.

The focus of this thesis is on the calculation of the WCET given models for the processor and Java methods, while the extraction of models from the hardware and program code will not be considered in great detail. Consequently, the evaluation of the developed tool focuses on analysis times and the quality of the cache approximation, and to a lesser extend on differences due to a conservative over-approximation of the execution path set. While issues related to concurrent systems, such as scheduling analysis, preemption and synchronization will not be discussed, work in these areas depends on an effective WCET analysis.

Outline Chapter 2 starts with an introduction to real-time systems, providing an overview and introducing important concepts. After a brief introduction to the Java programming language, the *The Real Time Specification for Java* is presented, which adopts Java to address the challenges posed by real time systems. We then discuss the requirements of high-integrity real-time applications, and review two high-integrity profiles, restricted subsets of Java, which should facilitate the use of Java for safety-critical applications.

Chapter 3 deals with worst-case execution time analysis in general, and issues specific to Real-Time Java. First, we introduce concepts of control flow analysis, a necessary prerequisite to determine possible execution paths. Then high-level WCET analysis, the extraction and modeling of feasible execution paths, is discussed. Next, the Java Optimized Processor (JOP), a Java processor for real-time applications, is introduced. We investigate low-level WCET analysis in the context of JOP, including analysis techniques for JOP's instruction cache. At the end of the chapter, the IPET technique for calculating WCET bounds is presented.

Chapter 4 discusses the calculation of WCET bounds using model checkers for timed automata. The theory of timed automata, and the UPPAAL model checker are explained first. It is shown how to use UPPAAL for WCET calculation, presenting a detailed translation algorithm. Finally, we discuss how to model different variants of JOP's method cache.

The tool developed in course of this thesis is presented in Chapter 5. After a description of the tool's architecture, some experimental results obtained with the tool are presented.

Finally this thesis concludes with a summary and future prospectives.

Chapter 2

Real-Time Java

The outstanding characteristic of real-time systems is that the correctness of an application depends on whether it meets given time constraints [SR94]. Usually, the interaction with an *external environment* plays an important role, and, as many real-time applications are safety critical, *reliability* is crucial as well.

In *hard real-time systems*, failing to satisfy time constraints may result in critical failures, possibly endangering precious resources. A control system which has to shut down a nuclear power plant in case of an accident is a classical example for a system with hard real-time constraints. Missing a deadline in *soft real-time systems* on the other hand results in degraded performance, but is acceptable to some degree. For example, failing to display a frame of streamed video in time will result in a lower quality, but is usually tolerable.

In this chapter we will discuss adoptions of the Java programming language for safety-critical, hard real-time systems. After a short introduction to real-time systems, and to the Java platform in general, the Real-Time Specification for Java (RTSJ) is reviewed. Finally, two high-integrity profiles for real-time Java, subsets of the RTSJ adopted to the needs of predictable hard real-time systems, are discussed.

2.1 Introduction to Real Time Systems

A real-time system executes possibly cooperating and dependent *tasks*, which can be classified as either *periodic* or *aperiodic*. Periodic tasks are run at regular time intervals, while aperiodic tasks are invoked to react to sporadic events. *Time critical tasks* have to *meet deadlines*, i. e., they need to complete within a certain time interval.

Deadlines are classified as *hard*, *firm* or *soft*. Hard deadlines have to be met under

all circumstances, and failing to do so results in a critical system failure. If a task misses a firm deadline, this does not result in a critical failure, but there is no use in completing the task either and it can be abandoned. Finally, missing a soft deadline lowers the quality of the system's service, but completing the task is still useful. Depending on the application, one or more kinds of deadlines may be present. Additionally, there may be non time-critical tasks as well.

It is crucial that the application designer is able to predict *a priori* whether deadlines will be met. There are different kinds of predictability: for hard deadlines, it has to be shown that they will certainly be met. As the environment interacting with the system will usually be non deterministic, certain assumptions regarding the frequency of events or faults have to be made. For firm and soft deadlines, weaker guarantees, for example in terms of the probability of missing a deadline, can be sufficient. Finally, deadline violations can be monitored, taking appropriate actions in case a deadline is missed.

Scheduling Scheduling is the problem of deciding which tasks should be executed when, and, in the case of multiprocessor systems, on which processor. While schedulers of, e. g., desktop operating system try to achieve a good overall resource utilization, a real-time system's scheduler's highest priority is to assign processing time in such a way, that time constraints are not violated.

When the scheduling is *preemptive*, running tasks may be suspended and resumed again, whereas in a non-preemptive system, a task, once started, can not be interrupted but only give away control voluntarily.

The task schedule can either be determined statically or decided at runtime. The former only works for a fixed set of periodic tasks, while for the latter a scheduling algorithm dynamically decides which tasks should be run. Scheduling algorithms can be further classified by distinguishing the kind of deadlines (hard, firm or soft) they are dealing with, and whether they are designed for periodic or aperiodic tasks.

One of the simplest scheduling techniques is the cyclic executive model [BS88]. A controller (the cyclic executive) explicitly starts tasks and delays execution according to a fixed pattern. Tasks are manually divided into smaller subtasks, providing a limited form of preemption. This approach has been a popular choice for hard real-time systems in the past, because it is reliable and easy to predict. The major drawback of the cyclic-executive approach is its inflexibility and the complicated programming model. The application designer needs to define appropriate subtasks, taking timing constraints and synchronization into account. Furthermore, events need to be handled by periodic tasks, which is rather counterintuitive.

Fixed priority scheduling requires each task to have a fixed (initial) priority. Whenever two tasks are ready for execution, the one with the higher priority is scheduled.

When the scheduling is preemptive, and the currently running task has a lower priority than another one ready for execution, the running task is suspended and the higher priority task is executed (*context switch*). Fixed priority schedulers allow to predict precisely whether tasks will meet their deadlines.

Tasks need means to synchronize access to *shared resources*, leading to additional complications [BDV03]. *Priority Inversion* denotes a situation where a low priority tasks holds a resource required by a high priority task. The high priority task is blocked waiting for the low priority tasks, whereas the latter may not get scheduled due to its low priority. A related problem is *Deadlock*, where tasks block each other by holding some resources the other tasks needs to proceed. When a fixed priority scheduler is used, those problems can be avoided using a priority ceiling protocol, raising the priority of a task when it acquires a lock on a shared resource. Finally, as synchronization delays execution, unpredicted synchronization may cause deadlines to be missed.

High-integrity real-time systems are real-time systems which interact with a safety critical environment. Failure of such a system may endanger human lives or cause environmental or economical disasters. In [KWK02], the authors summarize the software requirements for such a system: Predictable performance under specified conditions (*reliability*), ability to cope with abnormal situations (*robustness*), a transparent mapping from source to object code (*traceability*) and decreased likelihood that updates to the software introduce errors (*maintainability*). Reliable software allows to predict various aspects of runtime behavior, e.g execution and response times, memory consumption and possible control or data flow.

2.2 The Java Programming Language

Java [GJSB05] is an object-oriented programming language, designed for being portable (write once, run everywhere), and dynamic (loading of classes at runtime), while reducing the risk for runtime errors (static type safety, runtime checks, garbage collector). Its built-in support for concurrency (multithreading, synchronization) makes it attractive for concurrent systems.

Though originally developed for an embedded device, it became popular as a language for both desktop and server applications, and famous for remote execution of code in a safe runtime environment (*Applets*).

The *Java platform* not only consists of the language itself, but also defines the standard library and the execution environment. The *Standard Edition* of the Java platform, intended for desktop and server use, has been extended for multitiered server applications, resulting in the *Server Edition*, and restricted in order to execute Java on mobile devices (*Micro Edition*).

The Java Virtual Machine In Java, portability is achieved by compiling Java programs to a stack-based intermediate language, *Java bytecode*. The bytecode is in turn executed on a particular target architecture by an implementation of the Java Virtual Machine (JVM), an abstract stack machine. Before execution, the bytecode is verified to prevent malicious code from compromising the integrity of the execution environment.

The Java Virtual Machine Specification [LY99] defines the file format for compiled Java code (*class files*), the bytecode instruction set and the abstract machine for executing bytecode.

Classes form the compilation units of Java applications, and are compiled separately, resulting in corresponding class files. Each class has an associated constant pool, which defines symbolic names for numeric constants, string literals, fields, methods, interfaces and classes. References to entities defined in other compilation units are resolved by the class loader at runtime (*dynamic class loading*).

Each thread is associated with its own *frame stack*, a stack of execution contexts. The *current frame* holds the execution context for the currently active method of a thread, and is the only one used in the scope of that method.

The local data of a frame comprises an *operand stack* and an array of *local variables*. Instructions load data from local variables or memory onto the operand stack, and store it back into memory or the local variable array. Arithmetic instructions and conditional branches, however, operate on the stack only. [1]

Objects and arrays are allocated on the *heap*, which is shared among all threads. The runtime system's garbage collector repeatedly scans the heap for objects which are no longer used, and frees the space allocated for those objects.

The JVM distinguishes primitive types, including signed integers, unicode characters, floating point values and references to arrays and objects (Table 2.1). Instructions which inspect data values are typed, i. e., there a different instructions for different operand and result types.

Every bytecode instruction is encoded using 8 bits, though some instructions need to be followed by operand bytes, which provide constants or indices into the constant pool.

Stack management instructions rearrange values on the operand stack. The most common ones are `pop`, removing the stack's topmost value and `dup`, duplicating the topmost value.

Type conversion instructions are used to convert one numeric type into another one. The conversions from `byte`, `short` and `char` to `int` are implicit, and therefore not represented by a type conversion instruction. This implicit coercion suggests

[1] With the exception of `iinc`, used for incrementing local variables.

Figure 2.1: JVM types

JVM Type	Mnemonic	Description
`byte`	b	8-bit signed integer
`short`	s	16-bit signed integer
`int`	i	32-bit signed integer
`long`	l	64-bit signed integer
`char`	c	16-bit unsigned unicode character
`float`	f	single precision floating point
`double`	d	double precision floating point
`reference`	a	reference to object or array

that implementations of the JVM use 32-bit signed integers to represent all integral values with less than 32 bit precision.

Arithmetic instructions take one or two values from the stack, perform an arithmetic operation and push to result back onto the stack. Because at most 256 instructions can be encoded using a single byte, arithmetic instructions operate on `int`, `long`, `float` and `double` only. To support arithmetic operations on other integral types, the compiler has to insert appropriate type conversion instructions narrowing the result of an arithmetic operation.

Control transfer instructions unconditionally or conditionally change the next instruction to be executed, within the current method. They include support for conditional branches, unconditional jumps, switch statements and local subroutines.

Load and store instructions load local variables and constants onto the stack, or move values from the operand stack into local variables.

Array instructions are used to create and access arrays. The JVM supports typed arrays of booleans, primitive types and objects, and provides an instruction to query the length of an array. Except for those specific instructions, arrays behave as objects, and are passed by reference.

Object creation and manipulation instructions are used to build new class instances and access class and instance fields. Additionally, the instruction set supports subtype checks and dynamic type casts.

Instructions for method invocations cause the JVM to change control to a different method. Invoke instructions create a new stack frame and transfer control to the beginning of the invoked method, while return instructions transfer control back to the invoking method. For non-private instance methods, the actual method implementation executed depends on the type of the receiver and is resolved at runtime (*dynamic binding*).

Exceptions are used for non-local transfer of control, from the point the exception was thrown to the beginning of some code handling the exception. Exception handlers define the beginning of the code block to handle an exception, the code region for which they are active, and the `Throwable` subclass they handle. Exceptions are either generated by the runtime system, or by the `athrow` instruction. When an exception is thrown, the runtime system looks up the closest handler capable of handling the exception type. All stack frames above the method defining the exception handler are removed, and control is transfered to the beginning of the handler code. If no such handler is found, the program terminates.

Synchronization is supported by a built-in monitor mechanism. The `monitorenter` instruction acquires a lock on the given object, and `monitorexit` releases the lock. Additionally, methods can be declared as synchronized, implying that a lock on the receiving object has to be acquired before executing the method.

For a comprehensive description of the JVM's instruction set, the reader is referred to [LY99].

Java and Real-Time Systems Java, as defined in the *The Java Language Specification* and *The Java Virtual Machine Specification* isn't particularly well suited for real-time programming. In addition to leaving some aspects implementation defined, there are some features of Java that make it hard to predict its timing behavior [Sch04a]:

- The behavior of the runtime system's scheduler is not specified rigorously enough. It need not respect priorities or deal with priority inversion.
- The Java platform does not define high-resolution clocks, but for real-time systems, millisecond resolution is not sufficient.
- Java has an inflexible memory management model. All objects have to be allocated on the garbage collected heap. It is sometimes desirable or necessary to have threads not depend on the garbage collector, especially as real-time garbage collectors are still an active area of research.
- Dynamic class loading is hard to predict, both because it is difficult to estimate timing characteristics of the class loader and verifier, and because dynamically loaded code need not be known at compile time.
- It is very hard to give accurate timing characteristics for modern JVMs using *Just-In-Time compilation*. On the other hand, simple interpreters often deliver a poor performance. Solutions include dedicated Java processors (section 3.3) or Ahead-Of-Time compilation.

To make it feasible to use Java for real-time computing, several attempts have been made to address this limitations, which will be discussed in the remaining of this chapter.

The *Real-Time Specification for Java* (RTSJ) [BGB+] tries to conserve as many of Java's features as possible, creating an expressive and flexible execution environment for real-time computing (section 2.3). On the downside, it is rather complex, and not well suited for embedded devices. It also is not ideal for high-integrity applications, because it leaves some important aspects open and requires some hard to predict features to be present.

Attempts to provide a Java specification for use in high-integrity applications resulted in several specifications, so-called *profiles* (section 2.4). *Ravenscar-Java* targets hard real-time systems, by defining a compatible subset of the RTSJ. The *JOP real-time Java Profile*, later refined to the *Profile for Safety-Critical Java* [SSTR07], is a minimal specification for safety critical applications, which requires fewer resources to be implemented. Though this profile is not a subset of the RTSJ, Ravenscar-Java can be implemented on top of it.

2.3 The Real Time Specification for Java

The RTSJ addresses several areas in order to make Java a better platform for real-time applications. The most important of them deal with scheduling, memory management and synchronization issues.

Because of its complexity, the RTSJ is probably targeted at similar environments as the Java Platform Standard Edition, and not suitable for small embedded devices. Nevertheless, many concepts of RTSJ are used in the high-integrity profiles presented later, and therefore the specification will be reviewed in greater detail.

Time, Clocks and Timers Real-time systems need high-resolution real-time clocks and timers, and the notion of absolute and relative points in time. For this purpose, the RTSJ defines the concept of clocks (`Clock`), timers (`Timer`) and points in time (`HighResolutionTime`).

There is at least one clock in the system, the system's real-time clock. It provides monotonic, non-decreasing time values, but need not be synchronized with the external world. It is argued that having more than one local clock is useful when different time resolutions are needed, e. g., when some events have a period of hours and others fire every few microseconds.

A timer is associated with a specific clock and fires an event when a certain amount of time has passed on that clock. A `OneShotTimer` fires once after being activated, while a `PeriodicTimer` fires repeatedly, given a start time and a time interval.

Finally, the subclasses of `HighResolutionTime` allow to represent time with nanosecond precision. `AbsoluteTime` represents an absolute time value with respect to a `Clock`, while instances of `RelativeTime` represent time intervals. Absolute points in time and time intervals are conceptually quite similar, and differ mainly in their relation to clocks. In both cases time values are relative to some clock and are represented as normalized pairs of 32-bit nanosecond and 64-bit millisecond integers.

Threads and Scheduling The RTSJ requires a fixed priority, *preemptive* scheduler with at least 28 priority levels. *Schedulable objects* include real time threads (`RealTimeThread`) and handlers for asynchronous events (`AsyncEventHandler`). The state of a schedulable object is either *Executing*, *Blocked* or *Eligible-For-Execution*. If a schedulable object is blocked, it cannot be executed and is waiting for some event to make it eligible for execution (written as *Blocked-for-Event*). The scheduler has to ensure that the schedulable object with the highest active priority is running, if it isn't blocked. Objects which have the same active priority are maintained in a first-in first-out (FIFO) queue, i. e., within one priority level, the object which becomes eligible for execution first, is scheduled first.

A *release event* for an schedulable object marks it as eligible for execution if it is *Blocked-For-Release*. The time of the next release event is called release time. If a task completes, it switches from *Executing* to *Blocked-For-Release* state.

Periodic threads are released at fixed intervals. In addition to ordinary real-time threads (`RealTimeThread`), the thread model includes the class `NoHeapRealtimeThread`, suitable for high priority threads which do not need garbage collection and therefore may preempt the garbage collector at any time. A periodic thread completes by calling `waitForNextPeriod()`, which transfers control to the scheduler.

Periodic parameters control the execution of periodic objects. The *start time* and the *period* fix the intended release times of the object. The *deadline* specifies the time interval that is allowed to pass between the release time and the completion of a periodic object, and needs to be less than or equal to the period.

Additionally, the scheduler monitors deadlines and execution costs. A cost overrun occurs if a thread consumes more processing time than it was expected to consume. On a deadline miss or cost overrun, handlers can be invoked, taking appropriate actions.

Finally there is also the concept of process groups, groups of schedulable objects whose combined execution has further timing constraints. Conceptually, a server is monitoring the execution of schedulable objects in one process group, and monitors deadline misses and cost overruns of the combined execution.

Asynchrony The RTSJ provides mechanisms to handle asynchronous events (`AsyncEvent`), which might be triggered by client side code, the run-time system (for example on a deadline miss) or by external events (called *happenings*). Events are handled by asynchronous event handlers (`AsyncEventHandler`), which are schedulable objects similar to real-time threads, but released when certain events fire, rather than on a regular basis. They can be bound to a dedicated real-time thread (`BoundAsyncEventHandler`), but this isn't mandatory and not advised in applications with a very large number of asynchronous event handlers.

The scheduler maintains an event queue for each event handler, with a fixed initial size. When the event queue is full, depending on the policy the event is either ignored, an exception is raised, it replaces some yet unprocessed event, or the size of the event queue is increased.

For event handlers, the arrival time of the corresponding event corresponds to the release time of the handler, in case the event is accepted. If the event is rejected, the corresponding handler is not released. The *minimum inter-arrival time* (MIT) specifies the shortest permissible time interval between two consecutive events of the same kind. If another event occurs before that time passed, either *Arrival-time Regulation* or *Execution-time Regulation* is performed. The former either ignores the event, raises an exception, or, under certain circumstances, replaces a pending event. The latter on the other hand delays the processing of the event, making sure that only one event per MIT is processed, but does not drop it.

The RTSJ further provides mechanisms for *Asynchronous Transfer of Control*, which allows to interrupt and terminate running threads in an asynchronous way. Asynchronous exceptions are generated by calling `Thread.interrupt()` or invoking fire on a `AsynchronouslyInterruptedException` object bound to some interruptible thread or event handler. Only methods which explicitly list `AsynchronouslyInterruptedException` in their *throws* clause can be interrupted asynchronously. Furthermore, asynchronous exceptions are deferred while a thread executes a synchronized statement block.

Memory Management While garbage collection is an important feature of Java, it also infers with the needs of real-time computing, as it is hard to estimate the timing behavior of conventional garbage collectors. The RTSJ specifies several mechanisms for memory management, in addition to traditional garbage collected heap memory. Memory is allocated in the currently active `MemoryArea`, either in garbage collected heap memory, in immortal memory, or in a scoped memory area. Objects allocated in `ImmortalMemory` are never reclaimed during the applications life time. This ensures that code only dealing with immortal memory may preempt the garbage collector at any time. Objects in `ScopedMemory` are alive during a certain region of the program (the associated scope), and are discarded afterwards.

Only one scope can be active at a certain point, but it is possible to nest scopes. The outer scope is called the parent scope.

One problem which arises with the use of scoped memory is that no object must be deallocated as long it is referenced by an object still in use (*referential integrity*). Put in other words, as soon as a scope is left, it must not be possible to access objects allocated in that scope. First, due the the exposed application interface, it is impossible that a local variable accessible from outside the scope holds a reference to an object in an inner scope. But the system has to ensure that no object allocated in an outer scope memory area, in immortal memory or heap memory references objects in an inner scope. The RTSJ specifies that implementation must check whether assignments to non-local variables violate this constraint, either statically or at runtime.

Another aspect which complicates memory management in Java are *Finalizers*. If an object defines a finalizer method, the runtime system has to execute it before the corresponding memory is freed. While finalizers for objects created in immortal memory need not be executed, finalizers have to be taken into account when predicting the time needed by the garbage collector, and when leaving scoped memory.

Finally, the `RawMemoryAccess` class provides means to access physical memory modeled as a consecutive array of bytes directly, in order to facilitate the development of device drivers, for example.

Synchronization In Java, synchronized access to shared resources is supported by the `synchronized` keyword, which allows a thread to obtain a lock on some object. If some thread holds the lock for an object, another thread trying to obtain a lock on that object blocks until the lock is released. The RTSJ specifies means to deal with the problem of priority inversion (see section 2.1), either using the `PriorityInheritance` or the `PriorityCeilingEmulation` policy.

The priority initially or manually assigned to some thread is called its *base priority*. The `PriorityInheritance` protocol requires that the priority of a thread holding a lock on an object is at least as large as that of any thread which attempts to acquire the lock. The priority of the object holding the lock is adjusted when another thread tries to obtain the lock. For example, if thread T_1 has priority 4 and holds a lock on o, and then thread T_2 with priority 5 tries to obtain a lock on o, the priority of T_1 is raised to 5.

For the optional `PriorityCeilingEmulation` policy, objects acting as locks are assigned a *ceiling priority*. When a thread obtains a lock for an object with a ceiling priority, its active priority is raised to the ceiling priority. Any thread who attempts to synchronize on such an object has to have a base priority which doesn't exceed the ceiling priority.

2.4 High-Integrity Real-Time Java

High-integrity real-time systems have quite different requirements for a programming language, compared to, e. g., ordinary desktop applications. One approach, which allows to reuse existing tools and programmer knowledge, is to define a restricted subset of an existing language, along with guidelines for the programmer (a *profile*). High-integrity profiles usually remove features which are not formalized properly or complicate the prediction of the program's timing behavior.

In general, the reliability of a system can be increased by using static analysis techniques. Without actually running the application, one proves the absence of certain classes of errors, and checks whether the implementation agrees with specified requirements. [BDV04] lists recommendations for the use of static analysis in sequential high-integrity systems build with the Ada programming language.

- *Control flow analysis* identifies loops and recursive invocations and ensures that the code is well structured and no unreachable code is present.

- *Data flow analysis* can be used to find variables which are unused, read without being defined, or written but never read.

- *Information flow analysis* detects dependencies between input and output variables, which can be checked against a specification.

- *Range Checking* is used to show that the values a variable might take are within a permitted range. Important use cases include checking that array indices are with defined bounds or showing that the second argument to integer division is non-zero.

- *Timing analysis* is concerned with finding upper bounds for the execution time of program fragments.

- *Memory usage analysis* determines memory consumption related safety properties, such as allocation rate bounds. *Stack usage analysis* is a special important case, determining the amount of stack memory that is needed to execute a program fragment.

- *Formal Verification* is the most ambitious static analysis technique. One attempts to prove that the code is correct with respect to a formal specification. First, proof obligations (so called verification conditions) are generated using the specification and the code, which are then verified using a theorem prover.

Additionally, static analysis is complemented by testing, to increase confidence that the implementation matches the specification.

Subsequently, the Ada Ravenscar profile [BDV03] has been defined to allow developers to use the Ada programming language in concurrent hard real-time systems with preemptive scheduling. There are many concurrency related problems, some of which are usually checked at runtime. In a high-integrity system it is desirable to exclude the possibility of such errors using static analysis.

- Detect violations of the priority ceiling protocol.
- Make sure the program is free of race conditions, e. g., deadlocks.
- Ensure that (periodic) tasks do not terminate.
- Detect unsynchronized access to shared variables.
- Perform *WCET analysis* and subsequently *Scheduling Analysis* to ensure that no task will miss its deadline at runtime.

Ravenscar Java In [PW01], the authors define a high-integrity profile for Java, based on the work on Ada Ravenscar. The work was later refined in [KWK02] and given the name *Ravenscar Java*. It is a restricted subset of the RTSJ, avoiding features which make it difficult to build predictable real-time systems. Though Ravenscar Java applications are in principal valid RTSJ applications, the profile introduces a few new classes, which need to be available as a library.

Common restrictions of high-integrity profiles concern the use of hard to analyze constructs. In both high-integrity profiles, `sleep`, `wait`, `notify` and `notifyAll` must not be used and dynamic class loading is prohibited.

In the Ravenscar Java profile, the system is divided into a initialization and mission phase. In the initialization phase, which is assumed not to be time critical, threads are created and configured, event handlers are set up and memory is allocated. The code for the initialization phase resides in the `main()` method of the Java program.

In the time critical mission phase, it is prohibited to create garbage collected objects on the heap. As a consequence, all periodic threads have to be instances of `NoHeapRealTimeThread`. Ravenscar Java restricts periodic threads further, only permitting instances of `PeriodicThread`, which take the actual thread logic as an argument, and call `waitForNextPeriod()` as soon as the supplied `Runnable` completes its work.

Event handlers have to be bound to dedicated real-time threads. All event handlers have to be instances of `SporadicEventHandler`, which itself is a subclass of `AsyncBoundEventHandler`. Similar to `PeriodicThread`, the actual handler logic

can either be implemented by subclassing `AsyncBoundEventHandler`, or by supplying an object of type `Runnable`.

During the mission phase, objects can only be allocated in scoped memory. It is also possible to directly access physical memory.

In Ravenscar Java, no deadline or cost overrun handlers are present, as one has to show statically that no deadline misses will occur. The mechanisms for asynchronous transfer of control as defined in the RTSJ have been excluded, as they make timing analysis difficult. Furthermore it is mandatory to use priority ceiling emulation. Listing 2.1 shows an example of a Ravenscar Java program.

The JOP Real-Time Java Profile JOP also defines a real-time Java profile [Sch04a], tailored for resource-constrained embedded systems. It is not fully compatible with the RTSJ, but shares similar concepts. It is possible to implement Ravenscar Java on top of the JOP profile, demonstrating that the JOP profile is sufficiently expressive, while having a smaller runtime and memory overhead than the Ravenscar Java profile.

Real-time threads are derived from `RtThread`, are started by the scheduler when `missionStart()` is invoked and complete whenever `waitForNextPeriod()` is called, as in the RTSJ. A real-time thread is configured by specifying a priority, a period and a start offset, while the deadline is implicitly assumed to be equal to the period.

There are two kind of event handlers, `HwEvent` for hardware generated interrupts, and `SwEvent` for software generated events. The latter are released when `SwEvent.fire()` is invoked. For event handlers, the minimum inter-arrival time is specified.

The scheduler is a preemptive, priority-based scheduler, but uses unique priority levels, i. e., each schedulable object has to have its own priority level. Scoped memory is supported, but garbage collection is not. Consequently, unless a scoped memory area is associated with the current execution context, objects are allocated in immortal memory, which has no representation on its own.

Another outstanding feature of the JOP profile is the fact that the scheduling algorithm can be implemented at the application level. Listing 2.2 illustrates how to implement the example given in Listing 2.1 using JOP's real-time Java profile.

Tool Support A high-integrity profile needs strong tool support to ensure that the rules defined are obeyed. [PW01] provides a list of requirements for analysis tools.

- The tool has to check that no features prohibited by the profile, such

```
public class ExampleApp extends Initializer {
  class RtWorker implements Runnable {
    SporadicEvent event;
    public Worker(SporadicEvent ev) { this.event = ev; }
    public void run() { /* real-time work */
      ...
      event.fire();     /* fire event */
      ...
    }
  }
  class RtEventHandler implements Runnable {
    public void run() {
      ... /* real-time work */
    }
  }
  public void run() { /* Non time-critical initializer */
     SporadicEventHandler eventHandler =
       new SporadicEventHandler(
         new PriorityParameters(13), /* Priority 13 */
         new SporadicParameters(
            new RelativeTime(1,0),   /* 1 ms MIT */
            5),                      /* event queue size */
         new RtEventHandler());
    final SporadicEvent event = new SporadicEvent(eventHandler);
    PeriodicThread  worker1 =
      new PeriodicThread(
        new PriorityParameters(12),
        new PeriodicParameters(
            new AbsoluteTime(0,0),   /* start time */
            new RelativeTime(5,0)),  /* 5 ms period */
        new RtWorker(event));
    PeriodicThread  worker2 =
      new PeriodicThread(
        new PriorityParameters(11),
        new PeriodicParameters(
            new AbsoluteTime(10,0),   /* start after 10 ms */
            new RelativeTime(20,0)),  /* 20 ms period */
            new Runnable() { public void run() { ... } });
    worker1.start();
    worker2.start();
 }
 public static void main(String[] args) {
   new ExampleApp().start();
 }
}
```

Listing 2.1: Ravenscar Java example

as dynamic class loading or invocations of `sleep` are used during the application, or during the mission phase. Threads and event handlers must not be created during the mission phase.

- It needs to check the correct use of scoped memory (referential integrity) and make sure that real time threads do not create garbage collected objects.

- Other sequential static analysis techniques, as described before,

```
public class ExampleApp {
  /* Real time thread */
  class WorkerThread extends RtThread {
     public WorkerThread (int period,
                          int deadline,
                          SwEvent eventCallback) {
        super(period,deadline);
        init(); /* non time critical initialization */
     }
     public void run() {
        for(;;) {
          work();
          /* completion */
          waitForNextPeriod();
        }
     }
     private work() {
       ...
       eventCallback.fire();
       ...
     }
  }
  class SwEventHandler extends SwEvent {
     public SwEventHandler(int priority, int mit) {
       super(priority,mit);
     }
     public handle() {
       work();
     }
  }
  public static void main() {
    /* initialization phase */
    /* highest priority thread, 1 ms MIT */
    SwEventHandler eventHandler =
      new HwEventHandler(13,1000);
    /* higher priority thread, 5 ms period */
    new WorkerThread(12,5000,eventHandler);
    /* lower priority thread, 20 ms period */
    new RtThread(11,20000,eventHandler) {
      public void run() {
        for(;;) {
           ... /* work */
           waitForNextPeriod();
        }
      }
    }
    /* start mission phase */
    RtThread.startMission();
    /* non time-critical work, below RT priorities */
    for(;;) {
      System.print("Alive\n");
      Thread.sleep(500);
    }
  }
}
```

Listing 2.2: JOP real-time application

should be employed. For example, one should check that no `NullPointerException`s are thrown at runtime.

- The tool has to perform scheduling analysis and timing analysis (see chapter 3).
- Worst-case memory consumption and worst case stack memory usage need to be computed.
- The tool should perform integrity checks, to make sure that neither source nor object code are manipulated after the analysis has been performed.

Discussion The high-integrity profiles for Java try to facilitate the use of Java for safety-critical real-time systems. Inspired by the high-integrity profiles for the Ada programming language, they exclude problematic features and list requirements for static analysis.

Both high-integrity profiles require a preemptive scheduler. While preemptive scheduling relieves the programmer from the burden of splitting tasks into smaller subtasks manually, timing analysis for preemptive systems is much harder. Especially in the presence of instruction and data caches, the time penalty of a context switch is hard to estimate. As a workaround, one can use synchronized blocks, locking the task to an object with highest-priority ceiling, to avoid preemption at time critical points. Still, this question is not discussed in detail in the presented profiles.

Another aspect which is not handled is fault-tolerance. While we want to ensure the absence of most errors at design time, it is important for high-integrity systems to deal with "the impossible", for example caused by hardware failures, erroneous specifications or a bug in the verifier. To some extend the Ravenscar specification is inconsistent here, stating that handlers for missed deadlines are unnecessary as scheduling is proven statically, but having `waitForNextPeriod` return `false` if the deadline is missed.

The static analysis of scoped memory also deserves more attention. I think that a runtime violation of scoping rules is not acceptable in high-integrity systems, so it should be mandatory to prove that no such violation will occur at runtime.

Chapter 3

WCET Analysis of Java Tasks

As explained in the last chapter, when dealing with hard real-time systems, we need to predict statically whether timing constraints will be met at runtime. For this purpose, we need to estimate how long it takes to execute a piece of code, e. g., a task or some part of it.

One possibility to obtain timings is to *measure* the execution time. The execution time of a task, however, depends both on its input and the execution environment. Influencing factors comprise shared resources, task parameters, and input from the external environment. Additionally, the internal state of the processor or virtual machine, especially its cache state, will influence the execution time as well. All these factors have to be taken into account when using a measurement-based approach. As a consequence, for most real world use cases it is intractable to obtain a provably safe WCET bound using measurements only.

The approach which will be pursued here, called *static* WCET analysis, is to calculate a *safe* upper bound for the execution time of program fragments at design time. This bound need not be equal to the actual worst case execution time, but should be as close as possible.

In general, calculating a WCET bound is an undecidable problem. Therefore, certain assumptions have to be made regarding the structure of the program to be analyzed [PW01]. For each loop and recursive call in the program logic, an upper bound on the number of executions has to be known. In Java, dynamic loading of classes is prohibited, and in this discussion, exceptions are excluded as well.

WCET analysis comprises low-level analysis, high-level analysis and the actual WCET calculation.

- Low-level analysis is about modeling the timing behavior of the execution environment, including effects of caches and pipelines.
- High-level analysis is concerned with extracting and modeling the set

of possible execution paths the program might take.

- Given appropriate models, we need some algorithm to calculate the actual WCET bound. In this thesis, we will consider formulating the calculation as an integer linear program (ILP), discussed in Section 3.5, and computing the WCET using a model checker for timed automata (Chapter 4). Finally, a tool implementing both approaches will be presented in Chapter 5.

As noted in [PB01], by using Java, one is able to separate high-level and low-level analysis effectively. The high-level analysis can be performed on Java source code and Java bytecode only, while the low-level analysis contributes timing information for the specific execution environment.

WCET analysis is a necessary prerequisite for scheduling analysis, but it is important to note that it is not independent of scheduling. In concurrent real-time systems, preemption and synchronization have to be taken into account, complicating the task considerably, especially if global low-level effects such as cache misses have to be taken into account. Here we will only be concerned about calculating an upper bound for the execution time of sequential program fragments, not considering interaction between tasks.

3.1 Control Flow Analysis

Control flow is concerned with the order in which instructions of a program, method or piece of code are executed. We distinguish *local control flow* within one method, and *global control flow* across method boundaries.

A *Basic Block* is a sequence of instructions, where the first instruction is the only entry, and the last one the only exit point. This implies that the last instruction in a basic block is the only one allowed to alter the control flow, and other jumps or branches may only target the first instruction of a basic block.

A *control flow graph* (CFG) is a representation of a method's control flow in terms of a directed graph. The CFG's nodes are basic blocks, and its edges represent a *potential transfer of control*. A CFG has an unique entry node, without incoming edges, and an unique exit node, without outgoing ones.

If the execution of the last statement of basic block A is possible followed by execution of the first statement of basic block B, there is a directed edge from A to B in the CFG. A path from the entry to the exit node corresponds to one execution sequence. Control flow graphs are conservative approximations to the actual control flow: If a property is true for all paths of the CFG, it also holds for all

```
/* Bytecode corresponds to:
 *   int val = 123;
 *   for (int i=0; i<100; ++i) { val += i; }
 *   return;
 */
 0:  bipush 123
 2:  istore_0
 3:  iconst_0
 4:  istore_1
 5:  iload_1
 6:  bipush 100
 8:  if_icmpge 21
 11: iload_0
 12: iload_1
 13: iadd
 14: istore_0
 15: iinc  1, 1
 18: goto  5
 21: return
```

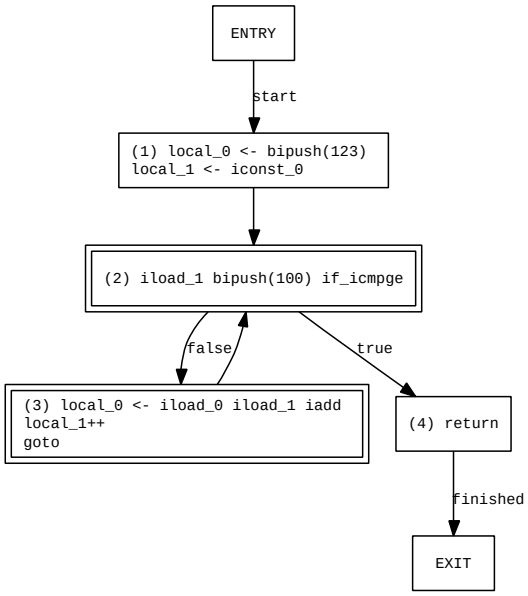

Figure 3.1: A Control Flow Graph for a simple Java method

possible execution sequences. Figure 3.1 shows an example of a CFG representing the given bytecode sequence.

A *loop* is a piece of code, whose execution may repeat without executing surrounding code in between. Loops are strongly connected components of a control flow graph, i. e., there is a path between every pair of basic blocks in the loop. We will only consider *reducible loops* with a single entry node, the *loop header*. The loop header *dominates* all other nodes in the loop, i.e. every path from the methods' entry to some node in the loop passes through the loop header. In Figure 3.1, the basic blocks (2) and (3) form a loop. (2) is the loop header, the edge (1) $\rightarrow$ (2) enters the loop, the *back edge* (3) $\rightarrow$ (2) continues the loop, and the edge (2) $\rightarrow$ (4) exits the loop. For a rigorous definition of loops, and algorithms to compute them, the reader is referred to [Hav97].

While control flow graphs capture the local flow of control, *call graphs* are static descriptions of the possible sequences of method invocations during execution. A call graph is a directed graph, whose nodes and edges represent methods and method invocations, respectively. In order to distinguish different invocations from and to the same two methods, the edges are labelled with *call sites*, the locations of the corresponding invoke instructions. If at call site L in method m, method n is possibly invoked, then there is an edge from m to n labelled with L in the call graph. Cycles in the call graph correspond to recursive method invocations.

Supergraphs [Mye81] are generalizations of control flow graphs, combining the call graph with the set of corresponding control flow graphs. In supergraphs, invoke

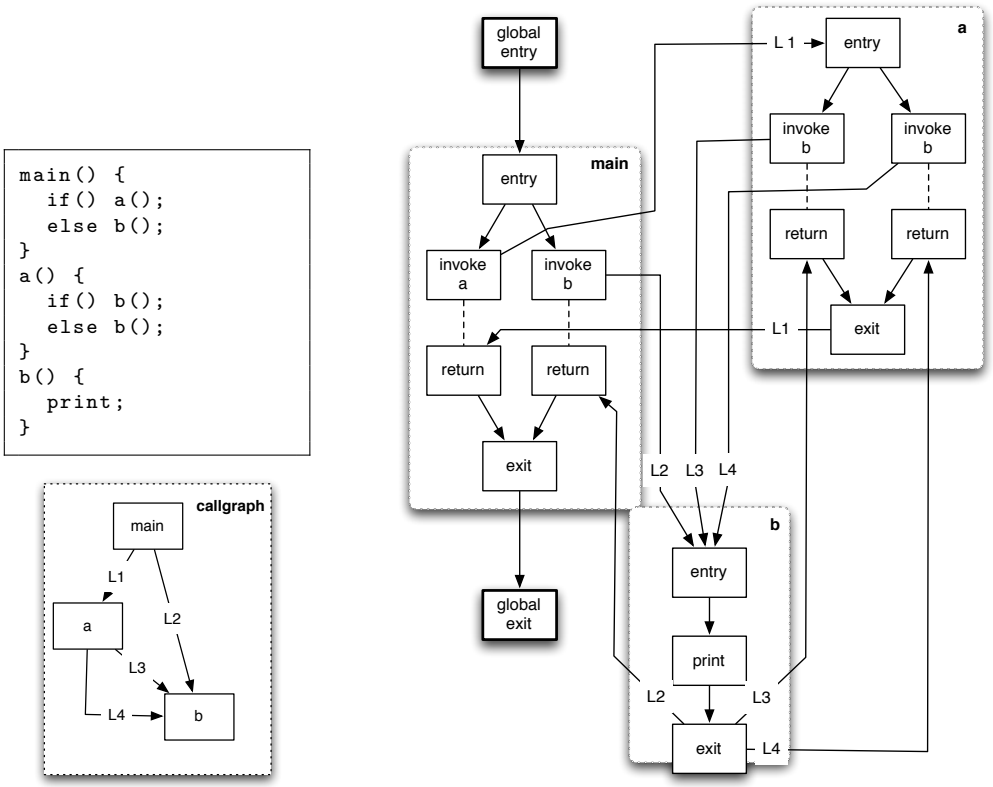

Figure 3.2: A supergraph of three methods `main`, `a` and `b`, and the corresponding call graph

instructions are considered as control flow statements and have to be the only statement in a basic block.

Given the control flow graphs of all methods present in the call graph, the supergraph is constructed as follows. For each invoke instruction, replace the corresponding node n with two nodes n_{call} and n_{return}. Incoming edges (x, n) are replaced by (x, n_{call}), and outgoing edges (n, y) by (n_{return}, y). Next, the individual control flow graphs are connected by adding *call edges*, from n_{call} to the entry node of the invoked method's CFG, and by adding *return edges* from the CFG's exit node to n_{return}. Finally, if the supergraph is intended to model the execution of some dedicated method `main`, we add global entry and exit nodes, one edge from the global entry to the entry of `main` and one edge from the exit of `main` to the global exit node. Figure 3.2 shows an example of a small supergraph, and its corresponding call graph.

The set of potential execution sequences represented by a supergraph do not correspond to the set of all paths from the entry to the exit node, but to the set of *valid paths* only. A path in the supergraph is valid, if the corresponding execution

path always returns to the site of the most recent call.

3.2 High-level Program Analysis

The goal of high-level WCET analysis is to describe the set of possible execution paths of a program. In order to calculate the worst-case execution time, the set of execution paths has to be finite. This implies that there is an upper bound on the number of times an instruction is executed.

High-level analysis can either be performed on the source code or object code level, or both. If the analysis is performed on the source code, it is necessary to map the path information on the machine code later, and take it into account during optimization. In addition to static analysis techniques, path information can be provided by humans using *annotations*, additional information for WCET analysis embedded in the source code.

Flow Facts Without additional information, cycles in the control flow graph (*loops*) or call graph (*recursion*) lead to infinite execution sequences, prohibiting the calculation of an WCET bound. Therefore, we need to limit the number of times cycles are executed. Secondly, the control flow graph might include so-called *infeasible paths*, execution paths which are never taken at runtime. To obtain tight WCET bounds, we also want to exclude as many infeasible paths as possible.

The first question to be addressed is how to describe restrictions to the set of executions paths, so called *flow facts* [KKP+05] . A formalism for describing flow facts is needed regardless wether they are obtained automatically, or are provided by the user. The most important requirement for computing WCET estimates is that the number of times a loop is executed must be bounded by some constant (*loop bound*). Additionally, either the call graph has to be acyclic, or bounds on the recursion depth have to be given.

The most common technique for expressing flow facts is to formulate them as *execution frequency constraints*. They limit the absolute or relative number of times a statement is executed along some execution path. Execution frequency constraints are valid in some *scope*, which means they apply to every execution of the statements belonging to the scope. Here are some examples:

- "During the execution of the method `f()`, the `print` statement is executed at most K times."
  The is an absolute bound on the execution frequency of the `print` statement, whose scope is the method `f()`. From a global perspective, this implies that the print statement is executed at most K times as often as `f()` is invoked.

- "The body of loop L is executed at most K times."
  Here the scope is the surrounding context of the loop. A more precise formulation would be "Every time L is entered, it's loop header is executed at most $K + 1$ times"
- "When executing the loop body, either branch A and D or branch B and C are taken."
- "For one execution of the method `f()`, the inner loop's body is executed as most K^2 times."

Relative capacity constraints can be arbitrary expressions involving execution frequencies, though linear arithmetic expressions are sufficient in many cases.

To increase the expressive power of frequency constraints, constraint may depend on a *context*. A *call-context* sensitive constraint only applies to those executions of a method invoked from certain other methods, while *loop-context* sensitive constraints only apply to some of the loop iterations. The following examples illustrate this concept:

- "If method g has been invoked from method f, the loop bound is K, otherwise it is J."
- "If method g has been invoked from method f, and f in turn has been invoked from h or i, the A branch isn't executed."
- "In the first 5 iterations of the loop, either branch B or branch C is executed, but not both of them."

Modeling possible execution paths using relative execution frequency constraints is the dominating approach for high-level analysis, but it is not the only possible one.

One potential drawback of relative execution frequencies is that the order of execution paths is lost and cannot be taken into account when estimating global low-level effects. As a consequence, it is not possible to accurately simulate the behavior of the cache during WCET calculation, for example. Furthermore, relative flow constraints sometimes fail to express loop flow concisely. Consider the program given in Listing 3.1: The execution frequencies of $L3$ and $L4$ are related, but it is hard to find a closed form in terms of relative execution frequencies.

In [Par93], valid execution paths are specified using regular expressions. As the set of execution paths needs to be finite, this method is complete. Furthermore, the

```
L1:
for(int i = 0; i < K; i++) {
  L2:
  if(rand() < 0.5) {
    for(j=K; j > i; j/2) { L3: /* workload */ }
  } else {
    for(j=K/2; j > i; j--) { L4: /* workload */ }
  }
}
```

Listing 3.1: Example of complex execution path set

order of execution paths is specified explicitly. On the downside, regular expressions describing execution paths can become rather complicated, and calculating the WCET is quite costly as well. As regular expressions correspond to finite state machines, this implies that a finite set of execution paths in general, and flow facts in particular, are representable by finite automata. This insight is a necessary prerequisite for using model checkers to calculate WCET estimates, and could be exploited to relate the above author's work with ours.

Until now, we've only discussed how to describe restrictions to the set of execution paths, but not how to obtain them. There are two complementing possibilities: Using static analysis to extract loop bounds from the program automatically, and extracting annotations explicitly specified by the application designer.

Flow annotations In the context of Java, there have been several proposals for WCET annotations.

In [BBW00], invocations of static methods defined in the class `WCETAn` are used to annotate the control flow. The only purpose of those method invocations is to provide informations about feasible paths, so they can be eliminated before deploying the application. It is important, however, that the compiler does not move, inline or eliminate these instructions.

Markers are objects representing locations in the source code, which are used to identify branches and to formulate frequency constraints. They are represented by `Label` objects in the proposal, and are defined by calling `WCETAn.Identify_Code(label)`.

Call contexts are identified using `Mode` objects. If a call site preceded by a call to `WCETAn.Define_Mode(mode)`, annotations referencing this mode only apply if the call stack includes the corresponding call site.

The bound of some loop is set by calling `WCETAn.Loopcount(iteration-count)` or directly before the loop. For call context sensitive loop bounds, a *Mode* is passed as additional parameter (`WCETAn.Loopcount(iteration-count,mode)`). Finally, infeasible paths are first identified using `WCETAn.Identify_Code(label)` at the

appropriate locations, and then calling `WCETAn.Dead_Path(mode,label)`.

Note that only the static identity of `Label` and `Marker` objects is of interest, which have to be inferred using program analysis.

These ideas could be extended to express arbitrary, context-sensitive relative frequency constraints, but to the best of my knowledge, this approach hasn't been pursued further.

The advantage of [BBW00]'s approach is that the flow facts are still present in the compiled bytecode, and are therefore portable. On the other hand, the code becomes cluttered with WCET aspects, which do not contribute anything to the actual behavior of the program.

Unfortunately, the authors do not pay a lot of attention to extracting path information and eliminating annotation-related code. In fact, creating new `Label` objects within mission-critical code is not possible in general, and a postprocessing step to eliminate all annotation-related code seems to be absolutely necessary. Using the static identity of local variables for labels is problematic too, as it requires a precise knowledge of the compilation strategy.

Because of this drawbacks, [HBW02] propose to annotate flow facts as comments in the source code, and then map them to bytecode, creating *annotation files* complementing the bytecode. The authors propose to use the same vocabulary (`Loopcount` etc.), but instead of actually calling a static method, a comment is placed at the right position in the source code. Either by modifying the compiler, or by relying on the compilation strategy of a standard compiler and using an extraction heuristic, the source code annotations are then related with the bytecode.

A third possibility, sketched by [HK07], is to extend and use Java's annotation mechanism. As of the time of writing, statement annotations are not yet part of the Java language, this approach requires to build a modified Java compiler. The second problem is that it is not obvious how statement annotations should be handled in general, as the compiler is free to duplicate statements, for example.

What all approaches have in common is that using an off-the-shelf compiler requires some knowledge about its inner workings and means to control optimization. Sun's java compiler makes it easy to map annotations to the bytecode level, as it only performs some rudimentary optimizations. Furthermore, bytecodes can be mapped to the line number of the source code file they have been generated from. Still, relying on a particular, but not formally specified compilation strategy might not meet the high standards one would expect from safety-critical systems.

Finally, one should note that flow annotations are not always necessary. Many loop bounds, and to some extend infeasible paths, found in hard real-time systems can be derived automatically using static analysis. For example, in an case study on the use of *abstract execution* for deriving loop bounds and infeasible paths, the

authors report that for the most part, the analysis generated flow facts of at least the same quality as the bounds provided earlier [BEG+08]. In our tool, we also try to detect as many loop bounds as possible, using the dataflow analysis framework presented in [Puf09].

3.3 The Java Optimized Processor

One problem which remains to be addressed for high-integrity Java applications is to find an appropriate execution environment.

The Java Optimized Processor (JOP) [Sch05] is an implementation of the JVM in hardware, i. e., a processor executing Java bytecode. It has been designed with WCET analysis in mind, simplifying low-level analysis considerably.

The processor can be configured for different FPGA devices, and, in a typical configuration, consists of the processor's core, a memory interface, IO devices and an extension component supplying a multiplier and connecting the subsystems.

The complexity of Java bytecodes varies, some are very simple, while others are too complicated to realize them in hardware. For this reason, bytecode instructions are translated to a different, JOP-specific instruction set, the so-called *microcode*. The microcode is designed to fit the execution model of the JVM, also being a stack-oriented language with additional local variables. In contrast to bytecode, each microcode instruction is encoded using exactly 10 bits, carefully designed to make decoding easy. With the exception of `wait`, every microcode instruction takes exactly one cycle to execute.

The most important, simple bytecode instructions correspond to a single microcode instruction, while some more complicated ones are translated to microcode sequences. Those sequences are short assembler-like programs, and may also include (microcode) branches.

All microcode sequences reside in a dedicated ROM area of the processor, and each bytecode is mapped to an address within this area. Two bits of a microcode instruction are used for the `nxt` and `opd` flags. The former ends a microcode sequence and tells the processor to fetch the next bytecode, while the `opd` flag tells the processor that the next byte should be saved in an register, for later use as an operand. Interrupts are handled by inserting a special bytecode in the bytecode stream. Therefore, interrupts may preempt execution at bytecode boundaries only.

Even when using microcode, it might be to expensive to hardwire microcode sequences for all bytecodes. To adopt the processor to the resources available on a target platform, more complicated bytecodes can be implemented in Java itself, as static Java methods. Of course, there is a considerable overhead invoking a method for executing a bytecode, but for rarely used or expensive bytecodes it can

```
/* 1-to-1 correspondence */
iadd:   add nxt

/* bytecode to microcode sequence */
ineg:
      ldi -1        //
      xor           // bitflip
      ldi 1         //
      add nxt       // add 1

/* Bytecode implemented in Java */
static int f_newarray(int count, int type) {
    return GC.newArray(count, type);
}

/* JOP-specific Bytecode */
public class Native {
  ...
  /* translated to bytecode 'jopsys_rd' */
  public static native int rd(int adr);
  ...
}
jopsys_rd:
      stmra         // store memory read address (top of stack)
      wait          //
      wait          // wait for memory instruction to complete
      ldmrd    nxt // load data read from memory
```

Listing 3.2: Bytecode and microcode

be worth the savings in hardware resources.

Finally, some bytecodes are specific to JOP, and used for e. g., accessing the IO subsystem. The JOP linker rewrites invocations of methods in `com.jopdesign.sys.Native` to those bytecodes. Listing 3.2 shows the four different kind of instructions used in a typical JOP configuration.

Processor architecture The actual processor core consists of 4 pipeline stages, *bytecode fetch*, *microcode fetch*, *decode* and *execute*.

The first stage fetches bytecodes from *Bytecode RAM*, and maps the bytecode to an address in the microcode ROM. The Bytecode RAM itself is a method cache, filled on invoke and return. When the `nxt` flag is set for some microcode instruction, the next bytecode is fetched and translated to microcode. Otherwise, if the `opd` flag is set, the bytecode is loaded into the operand register.

The *microcode fetch* stage fetches microcode instructions from the microcode ROM and executes microcode branches. Relative microcode branch offsets are stored in a table, generated during the assembly of the microcode ROM.

The *decode* stage decodes the microcode instructions and generates addresses for the Stack RAM. The following trick is used to safe a pipeline stage: Microcodes

are classified as either *push* (increase stack size) or *pop* (decrease stack size) during decoding, allowing to perform the necessary stack fills or spills independently in the next stage.

Finally, the *execute* stage operates on two dedicated registers, A and B. For ALU operations, those two register hold the operands, the result is stored in register A. One a store instruction, the value of register A is written into memory. In both cases, the third value on the stack is moved from stack RAM into register B (fill). On a load instruction, the value is copied from memory into register A. Here, the old value of register A is written into register B, and the value in register B is moved into stack RAM (spill). In [Sch05], it is shown that this kind of stack cache can be implemented in such a way, that all reads and writes are executed during the execution stage, eliminating possible data dependencies.

Finally, there are several recent extensions to JOP, e.g. for symmetric chip multiprocessing [Pit08] and for adding hardware support for real-time garbage collection [PS08]. They will not be discussed here, but provide starting points for future work.

3.4 Low-Level Timing and Cache Analysis for JOP

Low-level timing analysis builds a timing model for the execution of single instructions or basic blocks, which depends on the particular execution platform used. In the simplest case, the number of cycles needed for executing a statement is independent of the history of the computation. Unfortunately, modern processor features, such as out-of-order execution, branch prediction, and instruction and data caches complicate the timing analysis considerably. Global low-level analysis tries to improve the WCET by incorporating timing dependencies beyond the basic block level.

A predictable timing behavior was a dedicated design goal of JOP. The time needed for executing bytecodes is independent of the execution context, and can be determined cycle-accurately. There is no out-of-order execution or branch prediction, there are no pipeline stalls, and the data cache does not influence the timing behavior.

JOP's only low-level feature with non-local effects is the instruction cache, which is discussed below. JOP's instruction cache is filled on invoke and return instructions only, limiting cache effects to those instructions.

To determine the time JOP needs to execute a bytecode, we have to analyze its translation to microcode. Bytecodes which are implemented in Java need not

```
/* 12 + 2w cycles, where w is the write delay  */

getstatic:
  ldm cp opd ; nop opd  ; ld_opd16u /* 3 cycles */
  add        ; stmra    ; wait      /* 6        */
  wait                             /* 7+w      */
  ldmrd      ; stmra    ;  wait     /* 10+w     */
  wait                             /* 11+2w    */
  ldmrd nxt                        /* 12+2w    */
```

Listing 3.3: Analyzing microcode

be considered, as they are treated in a similar way to ordinary static method invocations.

All microcodes except `wait` take one cycle to execute. When a memory access has been started, two consecutive `wait` instructions stall the processor until the data is available. Ordinary memory access has a fixed delay for read and write. The only data dependent operation is the cache load initialized by `stdbcrd`, where the number of cycles the second `wait` instruction delays depends on the size of the loaded method and whether it is cached or not.

To compute the number of cycles a microcode sequence needs to execute, we explicitly enumerate all execution paths and compute the cycles the instruction takes for each paths.This is feasible, as the number of branches in microcode sequences is typically very small. The WCET of the microcode sequence, given memory and cache characteristics, is then computed by taking the maximum number of cycles one execution path might take. A small examples is given in Listing 3.3.

JOP's Instruction Cache Instead of caching individual instructions, JOP loads the code of single methods at once. This decision was based on the desire to simplify WCET analysis, and the fact that typical Java methods are small.

Due to JOP's architecture, a certain number of cycles needed for loading a method are hidden, depending on the instruction causing the cache access. Therefore it is important to distinguish the kind of instruction which causes a cache miss.

The simplest kind of method cache is a one-block cache, which is filled on each invoke and return instruction. In this case, the time needed by invoke and return instructions can be statically determined, by computing the size of the largest method possibly loaded.

A *N-block LRU cache* holds up to N methods and uses a *least recently used* replacement strategy. In a two-block LRU cache, returns from leaf methods, i. e., methods which do not invoke other methods, are guaranteed to be cache hits. Additionally, if a leaf method is invoked several times in a row, all accesses but the first one will be cache hits. For more than two blocks, static control flow analysis

```
void a() {
        b();
        for(i = 0; i < N; i++) {
            c(true);
            c(false);
            b();
  }
}
void c(boolean invokeB) {
  if(invokeB) {
    b();
  }
}
void b() { }
```

Listing 3.4: FIFO cache timing anomaly

is used to classify cache accesses as either *always hit*, *always miss*, *first hit* or *first miss* [Mue00]. When using a N-block LRU cache, the size of the cache has to be at least N-times as large as the size of the largest method possibly invoked.

Variable Block Cache In its default configuration, JOP uses a so-called variable block cache, introduced in [Sch04b]. A *variable block cache* comprises N blocks, each with a fixed size B. However, methods may, depending on their size, occupy more than one block in the cache. The big advantage of variable block caches is that they can be kept relatively small (size of the largest invoked method), hopefully still providing a good caching behavior.

JOP implements a variable block cache using so called next-block replacement, effectively a first in, first out (FIFO) replacement strategy, with each method occupying a variable number of blocks.

Unfortunately, FIFO caches exhibit *unbounded memory effects* [RGBW07], so simulating the cache with an initially empty cache state is not a safe approximation. We identified that this negative result also applies when the cache sequence corresponds to a path in a non-recursive call tree (see following example). Furthermore, a LRU cache can possibly outperform a FIFO cache, even if it only consists of two blocks.

Example 3.4.1 (Timing Anomaly)
To illustrate timing anomalies that arise when using a FIFO cache, we consider the execution of method `a` of the program given in Listing 3.4, using a two-block FIFO method cache. We compare two cache behaviors, that of the initially empty cache E, and that of a filled cache F, initially containing `b`, followed by `a`. Before

reaching the loop in method `a`, E needs to load both `a` and `b`, while F does not change its state.

Instruction	Empty Cache E	Loads(E)	Filled Cache F	Loads(F)
	$\|?\|?\|$	0	$\|a\|b\|$	0
invoke a	$\|a\|?\|$	1	$\|a\|b\|$	0
invoke b	$\|b\|a\|$	2	$\|a\|b\|$	0
return to a	$\|b\|a\|$	2	$\|a\|b\|$	0

Now, we analyze the loop, and show that the E needs 3 loads per iteration, while the initially filled cache F needs to load 6 methods per iteration. At the end of the loop, the cache states are the same as at the beginning. For N iterations, E needs to load $2 + 3 * N$ methods in total, while F loads $6 * N$ methods. Therefore, for all $N \geq 1$, the initially empty cache needs to load fewer methods than the cache initially filled with `a` and `b`. We also conclude that the difference between the number of loads is not bounded by a constant.

Instruction	Empty Cache E	Loads(E)	Filled Cache F	Loads(F)
	$\|b\|a\|$	0	$\|a\|b\|$	0
invoke c	$\|c\|b\|$	1	$\|c\|a\|$	1
invoke b	$\|c\|b\|$	1	$\|b\|c\|$	2
return to c	$\|c\|b\|$	1	$\|b\|c\|$	2
return to a	$\|a\|c\|$	2	$\|a\|b\|$	3
invoke c	$\|a\|c\|$	2	$\|c\|a\|$	4
return to a	$\|a\|c\|$	2	$\|c\|a\|$	4
invoke b	$\|b\|a\|$	3	$\|b\|c\|$	5
return to a	$\|b\|a\|$	3	$\|a\|b\|$	6

The FIFO replacement strategy does not perform as well as the LRU replacement, but the constraint that a method has to span several contiguous blocks makes the hardware implementation of a LRU strategy difficult for variable block caches.

This is bad news for WCET analysis: abstract interpretation usually needs to assume that the cache is initially empty, but in the case of a FIFO cache we really need to *flush the cache* in order to get a safe approximation.

Even when inserting cache flushes, we cannot simply adopt existing dataflow analysis techniques. This is because in a FIFO cache, the effect a cache access has on the cache depends on whether the access is classified as hit or miss.

One technique to approximate the cache miss cost is based on the following fact: If we can prove that the set of methods accessed within a program fragment fits into the cache, each of them will be missed *at most once* when executing that fragment. The total cost for cache accesses when executing such an *all-fit code region*, can therefore be estimated by stating that if some method within the region is executed, it is loaded exactly once. Except for leaf methods, we do not know, however, whether the method will be loaded on invoke or return.

3.5 Calculating the WCET using IPET

The idea of the implicit path enumeration (IPET) technique is to represent the set of execution paths in terms of a directed graph, together with a set of *linear capacity constraints* [PS97]. The edges of the directed graph correspond to the execution of some piece of code, and are associated with the time needed to execute that code on the given platform. Alternatively, one can associate pieces of code with nodes, and sum the frequencies of incoming or outgoing edges to that node. The former approach has the advantage that timing may depend on the edge taken. For example, one some platforms taking the *branch* edge is more expensive than taking the *next* edge. The solution to an IPET problem consists of the calculated WCET and of an implicit representation of the worst case path. This implicit representation maps edges to the number of times they are executed on the worst case path.

Integer Linear Programming An *Integer Linear Program* (ILP) is an optimization problem of the form

$$\max \sum_{i \in \{1,\ldots,n\}} c_i x_i \quad \text{subject to } \mathcal{C}$$

The goal is to find integral values for a set of variables $\{x_1, \ldots, x_n\}$, s.t. the given *objective function* $\sum_{i \in \{1,\ldots,n\}} c_i x_i$ is maximized, where $c_i \in \mathbb{R}$. The solution is subject to the restriction, that all of the given linear constraints in $\mathcal{C}$ are fulfilled.

A *linear constraint* over a set of integer variables $\{x_1, \ldots, x_n\}$ is an inequality or equation of the form

$$\sum_{i \in \{1\ldots n\}} a_i x_i \leq d$$

The $a_i \in \mathbb{R}$ are called coefficients, and $d \in \mathbb{R}$ is called the inhomogenous term. In vector notation, constraints are written more succinctly as $ax \leq d$. Note that an

inequality $ax \geq d$ corresponds to $-ax \leq -d$, and an equality $ax = c$ corresponds to the conjunction of two inequalities $ax \leq d$ and $ax \geq d$.

A variable assignment fulfilling all of the given linear constraints is called *feasible solution* or simply solution, and the value of the objective function w. r. t. to a solution is called *objective value*. An ILP either has an optimal solution, is *infeasible* (there is no solution), or *unbounded* (there are solutions with arbitrarily large objective values).

Solving ILPs is in general a hard problem (it is NP-complete). Some instances, such as network flow problems, can be solved in polynomial time though. For a nice introduction to linear programming, see [BM07].

Mapping WCET calculation to Integer Linear Programming For each edge $e \in E$ in the supergraph, there is one integer variable $f(e) \geq 0$ in the ILP, corresponding to number of times e is executed. Let $\gamma(e)$ be the fixed cost of executing edge e. Then calculating the WCET bound corresponds to solving

$$\max \sum_{e \in E} \gamma(e) f(e) \quad \text{subject to } \mathcal{C}$$

where $\mathcal{C}$ is a set of linear constraints, to be described below.

The first set of constraints is called *flow conservation constraints*. For all nodes n but the global entry and exit node, the sum of the execution frequencies of the incoming edges has to be equal to that of the outgoing edges. The sum of the flow through outgoing edges of the global entry node, and that through the incoming edges of the global exit node should be exactly one. This is a necessary (but not sufficient) condition for the solution to represent a set of valid paths.

$$\begin{aligned}
\sum_{e_i \in \text{incoming}(n)} f(e_i) &= \sum_{e_o \in \text{outgoing}(n)} f(e_o) \\
\sum_{e_o \in \text{outgoing}(\texttt{entry})} f(e_o) &= 1 \\
\sum_{e_i \in \text{incoming}(\texttt{exit})} f(e_i) &= 1
\end{aligned}$$

If loops are present in the control flow graph, without additional constraints the solution will be unbounded using flow conservation constraints only. We could try to limit the flow through cycles by adding absolute frequency constraints, bounding the number of times the loop is executed globally. But as [PS97] have shown, this is not sufficient, and may lead to a solution with unconnected components.

To ensure that the solution represents a valid execution path set, we add *connectivity constraints* for loops. We need to assert that the loop header h is only executed

if at least one of the edges $(n, h) \in E_{impl}$, which imply that the loop will surely be executed, but which are themselves not part of the loop, is executed as well. We also need to limit the frequency of the loop's header's execution. Assuming that K is maximum number of times the loop header is executed, we add the constraint

$$\sum_{e_h \in \texttt{outgoing}(h)} f(e_h) \leq \sum_{e_i \in E_{\text{impl}}} K f(e_i)$$

for each loop. Note that loops may consist of more than one cycle, but for reducible loops it is sufficient to add one such constraint for every loop header.

If we deal with supergraphs, we also have to consider interprocedural cycles (recursion) and ensure that only valid execution paths are considered. Recursion is handled in the same way as loops are, while for the latter is is sufficient to add a constraint for each pair of invoke and return edges e_{invoke} and e_{return}:

$$f(e_{invoke}) = f(e_{return})$$

Finally, one may want to add additional constraints to exclude infeasible paths or describe more complicated loop bounds. For loop bounds of the form

> For every execution of one of the edges $e_m \in E_{\text{marker}}$ the loop identified by e_{header} is executed as most K times

we add the constraint

$$f(e_{header}) \leq \sum_{e_m \in E_{\text{marker}}} K f(e_m)$$

This constraint is similar to a connectivity constraint, but doesn't replace it - connectivity is only guaranteed when all marker imply that the loop header is executed.

For infeasible path constraints of the form

> During the execution of the scope entered when executing one edge $e_s \in E_{\text{scope}}$, each of edges $\{e_i \mid i \in \{1 \ldots n\}\}$ is executed at most once, but it is impossible that all of them are executed.

we add the constraint

$$\sum_{i \in \{1 \ldots n\}} f(e_i) \leq (n-1) \sum_{e_s \in E_{\text{scope}}} e_s$$

Many other flow facts can be modeled using linear constraints, sometimes by introducing new decision variables as demonstrated in Example 3.5.1. It has been shown that it is possible to model the exact implicit execution frequencies using the IPET approach. The proof of this fact given in [PS97] requires as many constraints as their are execution paths, however, and therefore cannot be used in practice.

On the other hand, flow constraints commonly used, such as ordinary loop bounds, show a high degree of locality, i.e. they only affect small, connected portions of the program, and are easy to handle for modern ILP solvers.

Example 3.5.1
Suppose we're given the following specification

> It is impossible that all of the edges $\{e_i \mid i \in \{1 \dots n\}\}$ are executed, though some of them might be executed more than once.

For this example, we need to introduce additional binary decision variables, integer variables which may only take the values 0 or 1. Additionally, we need a "big constant" M, which is known to be larger than any expected execution frequency. While it can be tricky to choose M right, it is straightforward to check a posteriori if M was chosen large enough. Now, for each edge e_i we introduce a decision variable y_i, which should be 1 if and only if e_i is executed at least once.

$$f(e_i) \leq My_i \leq Mf(e_i)$$

Finally, the flow fact can now be formulated in terms of the binary decision variables

$$\sum_{i \in \{1 \dots n\}} y_i \leq n - 1$$

Context-Sensitive Flow Constraints Context sensitive frequency constraints only apply to the execution count of those edges representing executions in some specific call or loop context. For context sensitive constraints, edges are qualified by the set of call stacks and set of loop iterations they apply to. For loops, we usually distinguish the first and subsequent iterations.

To encode this kind of constraints, parts of the supergraph are duplicated, indexing edges of the duplicated subgraphs by their context. The call graph needs to be analyzed to determine which control flow graphs should be distinguished, and call and return edges have to be adopted accordingly.

The usual flow conservation and connectivity constraints are duplicated as well. For other flow constraints, a careful analysis of the supergraph is needed to relate the frequency of edges in different copies of a subgraph. We will give an example of context sensitive constraints below, but the exact formalisms needed for context sensitive constraints are beyond the scope of this thesis. The interested reader is referred to [The02].

Modeling method cache access In order to incorporate JOP's method cache into the IPET model, we insert *cache access nodes* into the supergraph. Before control flow changes to an invoke or return instruction, it either executes a cache load or cache hit, delaying the execution accordingly.

In principle, the same range of relative execution frequencies described for ordinary basic blocks can be used to restrict the execution frequency of cache load edges. In practice, cache access are classified as *hit* or *miss*, depending on the call and loop context. For example, if exactly one leaf method is executed during some inner loop's body, and a two-block LRU cache is used, the cache access can be classified as *first-miss*. The corresponding execution frequency constraint states that the cache load is executed only in the first iteration of the loop.

The approximation for the variable block cache described in section 3.3 can be implemented as follows. First, assume that all methods possibly invoked fit into the cache, and therefore each method is missed at most once. For every method m, add a decision variable b_m, which is true if and only if m is actually on the worst case path.

$$b_m \leq \sum_{e_{ca} \in \text{cache\_access}(m)} f(e_{ca}) \leq M b_m$$

If b_m is true, one of the edges leading to a cache miss node for m has to be executed, otherwise, none of them.

$$\sum_{e_{cm} \in \text{cache\_miss}(m)} f(e_{cm}) = b_m$$

Now, given the supergraph, find call sites where the invoked method and all methods possibly invoked during its execution fit into the cache, but the invoking method does not. If some method is reachable from one or more of those call sites, the corresponding supergraphs are duplicated. Finally, the constraints given above are added for each supergraph duplicate.

Chapter 4

Calculating the WCET using Timed Automata

Timed Automata [HNSY94] are finite-state automata extended with real-valued clock variables, developed to model and verify real-time systems. A model checker for timed automata decides whether safety and liveness properties, formulated as temporal logic formulas, hold for a given timed automaton, by exploring a symbolic representation of the automata's state space.

For execution time analysis, the model checker is used to explore an abstract model of the actual program, keeping track of the time passed. The abstract model should keep as much information relevant to the WCET analysis as possible, but still stay simple enough to avoid a state space explosion. States of the model may include information about the execution history, which in turn can be used to improve the approximation of global low-level effects, such as cache accesses. Moreover, timed automata have an expressive notion of time, and can be used to model more complex timing aspects, such as synchronization and scheduling of concurrent processes.

In this chapter, we model the behavior of Java tasks as networks of timed automata, and calculate a WCET bound using the model checker UPPAAL. In order to demonstrate the increased expressive power, we show how to simulate JOP's method cache, and discuss other possible applications of timed automata model checking.

4.1 Introduction to Timed Automata

In this section, we formally define timed automata, following the presentation of [BY04].

A timed automaton, or more precisely *timed safety automaton*, is a finite automaton consisting of a finite set of locations and of directed edges connecting locations. Edges are associated with actions, represented by symbols from a finite alphabet Σ.

$\mathcal{C}$ is a finite set of clock variables, ranging over the non-negative reals. A *clock assignment* u is a function $\mathcal{C} \to \mathbb{R}^+$ assigning a non-negative value to each clock. The clock assignment $u + d$ assigns the value $u(c) + d$ to each clock $c \in \mathcal{C}$, while $u[r \mapsto 0]$ assign 0 to the clocks $c_r \in r$, and the same value as u to those not in r.

A *clock constraint* is a *conjunction* of atomic constraints of the form $x \bowtie c$ or $x - y \bowtie c$, with $x, y \in \mathcal{C}$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. We write $B(\mathcal{C})$ to denote the set of all clock constraints over $\mathcal{C}$.

Example 4.1.1
Assume $x, y, z \in \mathcal{C}$ are clock variables. Then $x < 3$ and $z = 5$ as well as $x - y \leq 5$ and $z - x > 3$ are atomic clock constraints. The conjunctions $0 < x \leq 3$ and $1 \leq y \wedge z - x = 4$ are clock constraints as well. In the clock assignment $\{x \mapsto 1, y \mapsto 1, z \mapsto 5\}$ all of these clock constraints evaluate to *true*.

A timed automaton $\mathcal{A}$ is a tuple $\langle N, l_0, I, E \rangle$. N is the set of locations, with l_0 being the unique start location. I is a function $L \to B(\mathcal{C})$, mapping locations to *invariants*.

The transition relation E is a set of labeled edges $l \xrightarrow{a,g,r} l'$, where $l, l' \in N$ are locations, $a \in \Sigma$ is an *action*, $g \in B(\mathcal{C})$ the transition's *guard* and $r \subseteq \mathcal{C}$ is a set of clocks, which are reset when the transition is taken.

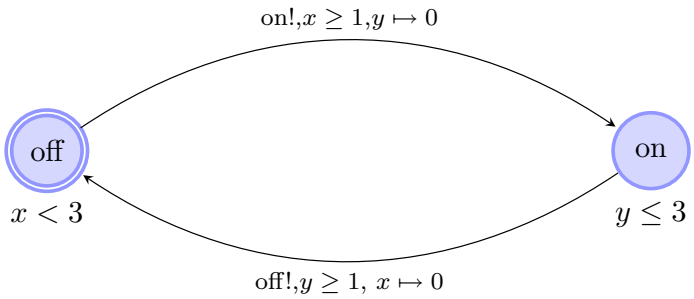

Figure 4.1: Simple timed automaton referencing two clocks $x, y \in \mathcal{C}$

Example 4.1.2
Figure 4.1 illustrates a simple timed automaton

$$\langle \{\texttt{on}, \texttt{off}\}, \texttt{on}, \{\texttt{off} \mapsto x < 3, \texttt{on} \mapsto y \leq 3\}, E \rangle$$

$$\text{with } E = \{\texttt{off} \xrightarrow{\text{on!}, x \geq 1, y \mapsto 0} \texttt{on}, \texttt{on} \xrightarrow{\text{off!}, y \geq 1, x \mapsto 0} \texttt{off}\}$$

which will be used as a running example. The automaton references two clocks $x, y \in \mathcal{C}$. It has two locations, the initial location `off` and another one named `on`. Intuitively, the system delays at least one and at most three time units at `off` or `on`, respectively. Then it switches to the other state, resetting y or x to 0.

The operational semantics of a timed automaton $\langle N, l_0, I, E \rangle$ is defined in terms of a *timed transition system*. The timed transition system's state $\langle l, u \rangle$ consists of a location $l \in N$ and a clock assignment u. The initial state is $\langle l_0, u_0 \rangle$, with u_0 assigning 0 to each clock variable. In each step the transition system takes, either time passes (delay transition $\xrightarrow{d}$) or the location changes and some clocks are reset (action transition $\xrightarrow{a}$).

- Two states are related by a *delay transition*

  $$\langle l, u \rangle \xrightarrow{d} \langle l, u + d \rangle$$

  if the invariant $I(l)$ evaluates to true in $u + d$.

- Two states are related by an *action transition*

  $$\langle l, u \rangle \xrightarrow{a} \langle l', u[r \mapsto 0] \rangle$$

  if all of the following conditions hold:

  - There is a transition $l \xrightarrow{a, g, r} l' \in E$
  - The guard g evaluates to true in u
  - The invariant $I(l')$ is true in $u[r \mapsto 0]$

The behavior of an automaton is characterized by the set of possible *runs* of the corresponding timed transition system. A run is a sequence of consecutive delay and action transitions

$$\langle l_0, u_0\rangle \xrightarrow{d_1}\xrightarrow{a_1} \langle l_1, u_1\rangle \xrightarrow{d_2}\xrightarrow{a_2} \langle l_2, u_2\rangle \ldots$$

The corresponding *timed-trace* is a sequence of time-action pairs

$$\langle (d_1, a_1), (d_1 + d_2, a_2), (d_1 + d_2 + d_3, a_3), \ldots\rangle$$

Finally, the timed language $L(\mathcal{A})$ accepted by the timed automaton $\mathcal{A}$ is the set of all timed-traces for which there exists a run.

Example 4.1.3
The initial state of the transition system for the automaton in Figure 4.1 is $(\texttt{off}, \{x \mapsto 0, y \mapsto 0\})$. In the initial location, the system has to delay for at least 1 time unit before the transition to `on` is enabled, but must take the transition before 3, because of the invariant $x < 3$. The action transition to `on` resets the clock y to 0. In some state $\langle \texttt{on}, u\rangle$ with $u(y) = 0$ the system first delays between 1 and 3 time units and then takes the transition to `off`. One possible run for this transition system starts with

$$\begin{aligned}
\langle \texttt{off}, 0\rangle &\xrightarrow{2}\xrightarrow{\texttt{on?}} \langle \texttt{on}, \{x \mapsto 2, y \mapsto 0\}\rangle \\
&\xrightarrow{3}\xrightarrow{\texttt{off?}} \langle \texttt{off}, \{x \mapsto 0, y \mapsto 3\}\rangle \\
&\xrightarrow{2.99}\xrightarrow{\texttt{on?}} \langle \texttt{on}, \{x \mapsto 2.99, y \mapsto 0\}\rangle \\
&\xrightarrow{1}\xrightarrow{\texttt{off?}} \langle \texttt{off}, \{x \mapsto 0, y \mapsto 1\}\rangle \\
&\ldots
\end{aligned}$$

The corresponding timed trace is

$$\langle (2, \texttt{on?}), (5, \texttt{off?}), (7.99, \texttt{on?}), (8.99, \texttt{off?}), \ldots\rangle$$

Given two automata $\mathcal{A}$ and $\mathcal{B}$, the question whether $L(\mathcal{A}) \subseteq L(\mathcal{B})$ (*language inclusion*) is undecidable in general. The question wether there is a run reaching some state $\langle l, u\rangle$ (*reachability*), however, is decidable. Reachability is central for verifying safety properties. By checking that a state $\langle l_f, u_f\rangle$ is *unreachable*, we can verify that a some kind of failure, represented by $\langle l_f, u_f\rangle$, will never occur.

Zone graphs The state space of a timed automaton has a finite representation in terms of normalized zone graphs. A zone graph consists of symbolic states, pairs of locations and zones, related by a symbolic transition relation.

A zone is a representation of the solutions to a clock constraint, i. e., it represents the set of all clock assignments which satisfy the constraint. The following operations on zones are needed to formalize zone graphs, and are used in the reachability algorithm presented later in this chapter. It is crucial for model checkers to provide efficient implementations of these operations. Given zones D and D':

- $D \subseteq D'$ is true if every clock assignment which is in D is also in D'.
- $D \wedge D'$ denotes the intersection of two zones, i. e., , the zone obtained by taking the conjunction of the corresponding clock constraints.
- $D^{\uparrow} = \{u + d \mid u \in D, d \in \mathbb{R}_{+}\}$ is the zone consisting of all clock assignments which can be reached from D by an arbitrary delay.
- $\text{reset}(D, r \leftarrow 0) = \{u[r \mapsto 0] \mid u \in D\}$ is the zone D with all clocks $c \in r$ reset to 0.

Symbolic states are pairs $\langle l, D\rangle$ of a location l and zone D, related by the *symbolic transition relation* $\rightsquigarrow$.

For every transition $l \xrightarrow{a,g,r} l'$ in the timed automaton there is a symbolic transition

$$\langle l, D\rangle \rightsquigarrow \langle l', \text{reset}(D \wedge g, r \leftarrow 0) \wedge I(l')\rangle$$

corresponding to the action transition in timed transition systems. The zone $\text{reset}(D \wedge g, r \leftarrow 0) \wedge I(l')$ is computed by first removing those clock assignments which do not satisfy the guard, then resetting the clocks in r to 0 and finally excluding those assignments, which do not satisfy the invariant $I(l')$.

In addition to the symbolic action transitions, there is a symbolic delay transition for every location $l \in N$:

$$\langle l, D\rangle \rightsquigarrow \langle l', D^{\uparrow} \wedge I(l)\rangle$$

To compute $D^{\uparrow} \wedge I(l)$, all clock assignments reachable via some delay from D are included first, then restricting the set to those, which satisfy the location's invariant.

Zone graphs closely correspond to timed transition systems.

- *Soundness*: Whenever there is a sequence of symbolic transitions leading from $\langle l, \{u_0\}\rangle$ to $\langle l', D\rangle$, and D includes the the clock assignment u, then there is also a run of the timed transition system from $\langle l, u_0\rangle$ to $\langle l', u\rangle$.

- *Completeness*: Conversely, if there is run from $\langle l, u_0 \rangle$ to $\langle l', u \rangle$, then there is also a sequence of symbolic transitions from $\langle l, \{u_0\} \rangle$ to $\langle l, D \rangle$, with $u \in D$.

Example 4.1.4
For the automaton in Figure 4.1, the initial symbolic state is $\langle \texttt{off}, x = 0 \wedge y = 0 \rangle$. Figure 4.2 illustrates the corresponding zone graph, comprising 6 states. States with empty zones and transitions to those states have been left out.

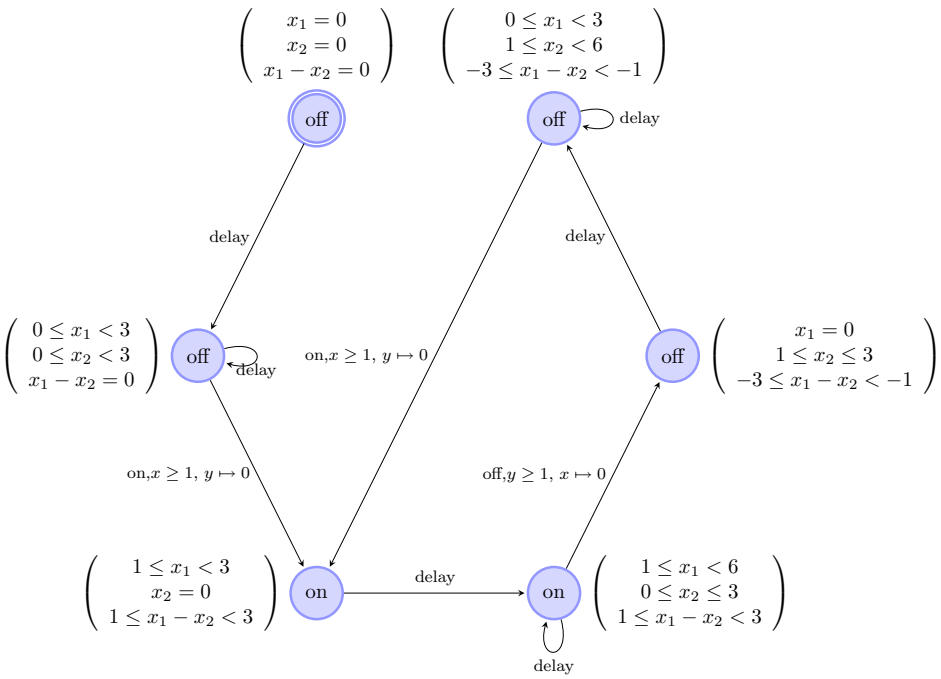

Figure 4.2: Zone Graph for the timed automaton of Figure 4.1

Unfortunately, the zone graph of an automaton may be infinite, possibly leading to non-termination when checking reachability. For this reason, a *zone normalization* operator relaxing zone constraints is applied, following the intuition that as soon as a clock is assigned a value larger than some constant, it doesn't matter how large exactly that value is.

For every clock x, $k(x)$ denotes the clock ceiling of x, the largest constant used in a clock constraint referencing x. If all clock constraints are of the form $x \bowtie c$, it

is sufficient to remove constraints $x < c$ and $x \leq c$ when $c > k(x)$, and replace constraints $x > c$ and $x \geq c$ by $x > k(x)$ if $c > k(x)$. In the presence of difference constraints of the form $x - y \bowtie c$, a different, more expensive normalization algorithm is needed [BY03].

In summary, reachability is decidable for timed automata, because for every timed transition system there is a finite zone graph, which is a sound and complete w.r.t. the timed transition system as defined above.

Reachability analysis *Symbolic reachability analysis* for timed automata computes whether a state satisfying some predicate ϕ is reachable. The algorithm presented in [LY97] stores the set of symbolic states that have already been considered in the set `PASSED`. Additionally, a worklist `WAIT` keeps the set of generated states, which have not been processed yet. The search proceeds as follows:

1. At the beginning of the search, add the initial symbolic state $\langle l_0, D_0 \rangle$ to the worklist `WAIT`.
2. Get and remove one symbolic state $\langle l, D \rangle$ from `WAIT`
3. If $\langle l, D \rangle$ satisfies ϕ, return `REACHABLE`.
4. Otherwise, check if $\langle l, D \rangle$ has already been considered. This is the case if there is some state $\langle l, D' \rangle \in$ `PASSED`, s.t. $D \subseteq D'$.
5. If the state hasn't been considered yet, add $\langle l, D \rangle$ to `PASSED`, generate all successor states $\langle l', D' \rangle$ with $\langle l, D \rangle \rightsquigarrow \langle l', D' \rangle$ and add them to `WAIT`.
6. If `WAIT` is not empty, goto 2. Otherwise, return `NOT REACHABLE`.

Note that in order to verify that some state cannot be reached, all states generated have to be kept in memory, limiting the size of search space that can be explored. Therefore, optimization techniques to minimize the amount of memory needed to store a zone are crucial for an efficient model checking implementation.

Networks of timed automata A *network of timed automata* is a parallel composition of a fixed number of n timed automatons, called processes, which share the same clocks and actions. Edges are now labeled with input actions $a?$, output actions $a!$ and one special symbol for internal actions, τ. Pairs of input and output actions $(a?, a!)$ form so-called channels, used for synchronized communication.

The transition system for the network of automata keeps track of all locations, i. e., the combined system's state consists of a vector $(l_1, \ldots, l_n)$ and a clock assignment u. Delay transitions are performed on all automata in parallel, and are suspect to

all of the invariants in the automata. Transitions labeled with the action symbol τ have the same semantic as transitions in an ordinary timed automaton, and are carried out separately. A synchronizing transition affects two processes with transition relations $l_1 \xrightarrow{a?,g_1,r_1} l_1'$ and $l_2 \xrightarrow{a!,g_2,r_2} l_2'$. If both transitions are enabled, they can be executed in parallel, reseting all clocks in $r_1 \cup r_2$ to 0.

4.2 The Model Checker UPPAAL

UPPAAL is a model checking tool for networks of timed automata [BDL04]. It features an expressive modeling language and useful extensions to the theory of timed automata, along with an efficient implementation of the verifier.

UPPAAL extends the theory of timed automata with bounded integer variables. Bounded integer variables can take a value in some predefined range, and are initialized to some fixed value. Boolean expressions referencing integer variables can be used as guards for transitions, and when taking a transition, variables can be updated using a C-like programming language.

An UPPAAL model comprises global declarations of clocks, channels and variables, and a set of *processes*. Each process is instantiated from a parametrized *template*, and has its own set of local clocks and variables.

UPPAAL introduces a couple of other extensions to the theory of timed automata, which simplify modeling of real-time systems:

- If some process is at an *urgent location*, delay transitions are not permitted.
- If there is process, which is at a *committed location*, the next transition taken has to be a transition from *some* committed location.
- *Urgent synchronization channels* are channels which have to be used as soon as possible, i. e., if the corresponding synchronizing transitions are enabled, they have to be used without delay.
- *Broadcast channels* are similar to synchronization channels, but instead of exactly one, zero or more processes synchronize with input actions. As soon as a transition synchronizing on a broadcast channel is enabled, it must be taken.

Verification using UPPAAL The purpose of a model checker, such as UPPAAL, is to verify certain properties of a model. Those properties are specified using

a temporal real-time logic, which asserts predicates on states and paths of the transition system.

Atomic *state formulas* are predicates on states of the transition system. UPPAAL has a rich expression language for state formulas, which may reference locations, clocks and integer variables. Additionally, the expression `deadlock` is true at some state if it has no outgoing transitions.

A maximal path of a transition system is either infinite, or ends in a state without outgoing transitions. Using state formulas as their basic building blocks, paths formulas are predicates on the set of all maximal paths starting from the initial state.

- `A []` ϕ, where ϕ is a state formula, is true if ϕ is true in all reachable states.
- `E <>` ϕ is true if there is a state such that ϕ is true.
- `A <>` ϕ is true if on all paths, ϕ becomes true eventually.
- `E []` ϕ is true if ϕ holds on all states of some path.
- ψ `-->` ϕ, "ψ leads to ϕ", is true if, whenever ψ is true in some state, ϕ will eventually become true.

Path formulas are used to verify reachability, safety and liveness properties.

A *reachability* property `E <>` ϕ states that a certain state is reachable. Verifying that a state is reachable is often used for basic consistency checks.

Safety properties `A []` ϕ assert that some invariant holds in every state of the system. To assert that some state formula ϕ holds at a specific location, we write `A [] (Process.Location &&` ϕ`)`.

Liveness properties are of the form ψ `-->` ϕ, stating that whenever some event takes place, it is followed by another event eventually.

Example 4.2.1
Assume we are modeling a real-time system for WCET analysis. Reachability formulas of the form `E <> Task.Location` are used to check that every location in some task is indeed reachable. The property that a task completes in time is written as `A[] (Task.Exit imply time` $\leq$ `c)`. To express that whenever a task is started, it eventually completes, we write `Task.Entry --> Task.Exit`.

Convex Hull Approximation While reachability is decidable using normalized zones, the number of states generated during the search may still be to large for practical purposes. One possibility to reduce the number of states is to perform approximate reachability analysis [Bal96]. Instead of enumerating the exact zones reachable for a location in the automaton, the convex hull, a convex over-approximation of the zones' union, is stored. If a state is not reachable using approximate reachability analysis, it is not reachable using an exact search either. Therefore, this approach is sound, but not complete for safety properties.

4.3 Calculating the WCET of Java Tasks

As explained in Chapter 3, WCET analysis is restricted to tasks with a finite set of execution paths. This set can be represented by a finite automaton, with nodes representing statements and edges possible flow of control. The key insight for WCET analysis is that by adding the elapsed time to the state space, it is possible to verify that a given execution time is indeed a safe WCET bound. The translation given here is based on the ideas from [OL], and has been subsequently refined based on [BKHO+08].

Overview We start by declaring a global clock t, which represents the total time passed so far, and build timed automata simulating the behavior of the program, one for each method. Additionally, we add a clock representing the local time passed at some instruction, t_{local}. We will not consider the analysis of multi-threaded applications here, but conceptually we need one local clock for each concurrent process.

There is one location I, which is the entry point of the program, and one commited location E, which corresponds to the program's exit. When execution has finished, the system takes the transition from E to the final state EE (Figure 4.3).

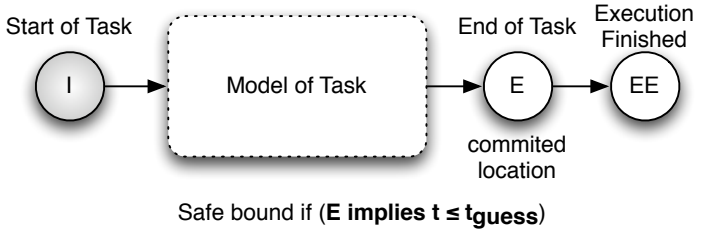

Figure 4.3: Calculating the WCET bound using UPPAAL

If we want to check whether t_{guess} is a safe WCET bound, we ask the model checker to verify

$$\texttt{A[]}\ (E \implies t \leq t_{guess})$$

If this property is satisfied, t_{guess} is a safe WCET bound, otherwise we have to assume it is not. Given a known lower bound and a safe upper bound, we perform a binary search to determine the precise WCET bound. If there are no known lower and upper bounds, we set $t_0 = 1$ and $t_i = 2t_{i-1}$ and search for the first n s.t. t_n is a safe bound.

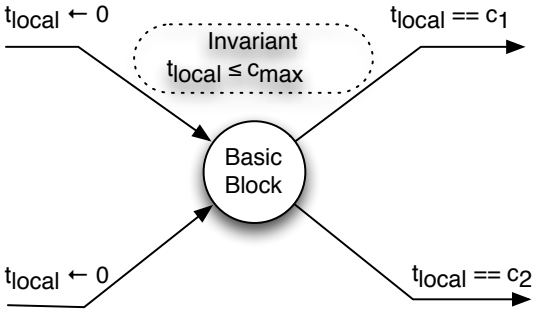

Figure 4.4: Modeling Basic blocks

From Control Flow Graphs to Timed Automata Given the CFG of a Java method m_i, we build an automaton simulating the behavior of that method by adding locations representing the CFG's nodes and transitions representing the flow of control. The initial location $M_i.I$ corresponds to the entry node of the CFG, and the location $M_i.E$ to its exit node.

To model the timing of basic blocks, we add updates, guards and invariants at the respective locations in the automaton. If the execution of a basic block takes at most c_e cycles depending on the outgoing edge e chosen, and at most c_{max} cycles independent of the chosen edge, we proceed as follows (Figure 4.4):

- On each ingoing transition to the location representing the basic block, we reset t_{local}.
- We add the invariant $t_{local} \leq c_{max}$ to the location.
- On each outgoing transition corresponding to the edge e, we add the guard $t_{local} == c_e$.

In the case of JOP, the cost is independent of the edge chosen.

In [BKHO+08], the authors use UPPAAL to perform scheduling analysis using stopwatch expressions, a recent addition to UPPAAL[CL00]. A stopwatch expression

ensures that a local clock is only running if some condition is true. Adding appropriate stopwatch expressions to the locations of basic blocks, local time only passes if the scheduler assigns a time-slot to the corresponding thread.

Modeling loops and infeasible paths It would be possible to eliminate bounded loops by *unrolling* them in a preprocessing step, but it is more efficient to rely on bounded integer variables.

Assume it is known that the body of loop n is executed at least L_n and at most K_n times. We declare a local bounded integer variable i_n representing the loop counter, ranging from 0 to K_n. The loop counter is initialized to 0 when the loop is entered, and is used in guards and for updates within the loop (Figure 4.5):

- If an edge implies that the loop header is executed on more time, add the guard $i_n < K_n$ and the update $i_n \leftarrow i_n + 1$ to the corresponding transition.
- If an edge implies that the loop is left, add the guard $i_n \geq L_n$ and the update $i_n \leftarrow 0$ to the corresponding transition.

Resetting the loop counter to 0 when leaving the loop is not necessary, but helps to reduce the number of states. Furthermore, it might be beneficial to set $L_n = K_n$, but this is only correct in the absence of timing anomalies, and therefore in general unsound in the presence of FIFO caches.

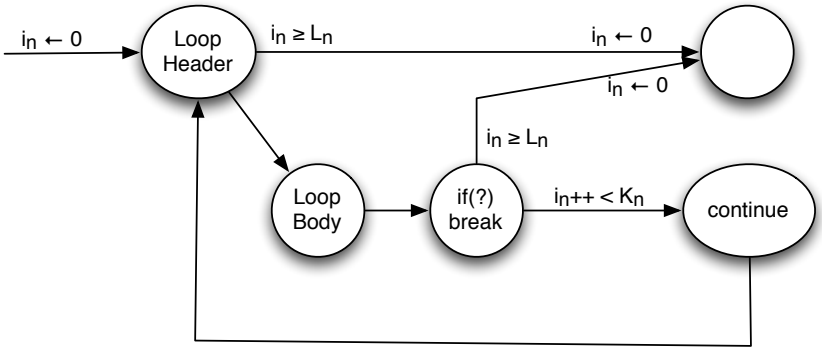

Figure 4.5: Modeling loops using bounded integer variables

In principle, every control flow representable using bounded integer variables can be modeled using UPPAAL, though we only implemented simple loop bounds in the current version of our tool. There is, however, an associated performance penalty – every integer variable contributes to the state space, which might quickly lead to a state space explosion.

For infeasible paths, if the execution of some basic block implies that another one is not executed, a boolean variable can be set when executing the first block and used as a guard before executing the second. When the corresponding scope is entered, the variable is reset. Other flow facts can be represented in a similar way.

Example 4.3.1
The program given in Listing 3.1, which is difficult to represent using flow graphs and relative execution frequencies, has a relatively simple representation using timed automata, shown in Figure 4.6. The number of states in the transition system depends on the size of the loop bound k though, and may grow fast with an increasing loop bound.

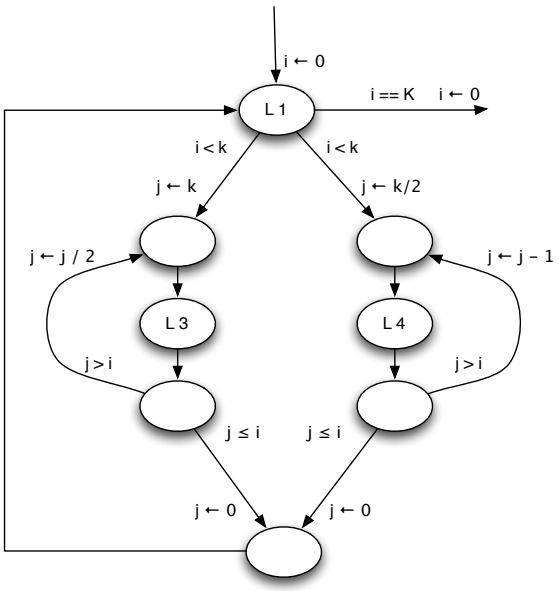

Figure 4.6: Timed automaton for the nested loop from Listing 3.1

Method invocations If there is more than one method, we build one automaton M_i for each reachable method m_i. For all methods m_i but the root method m_0, which is never invoked by another method, we add a transition from $M_i.E$ to $M_i.I$, to allow the method to be invoked several times (Figure 4.7).

To model method invocations we synchronize the method's automata using channels. Our initial design used two dedicated synchronization channels and additional bounded integer variables to select the invoked method and the correct caller when returning. It turned out, however, that a better performance is achieved when using one channel per method.

When a method m_i is invoked, the invoke transition synchronizes with the outgoing transition of $M_i.I$ on the invoked method's channel. When returning from method m_i, the transition from $M_i.E$ to $M_i.I$ synchronizes with the corresponding return transition in the calling method. This translation assumes that there are no recursive calls.

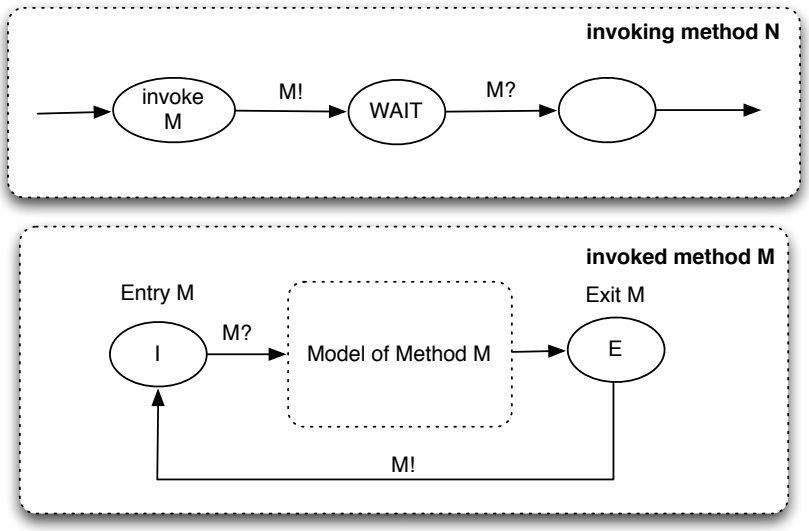

Figure 4.7: Modeling method invocations using processes and synchronization channels

Method Cache Simulation Using timed automata, it is possible to directly include the cache state into the timing model. It is most important, however, to keep the number of different cache states low, to limit space and time needed for the WCET calculation. JOP's method cache is especially well suited for model checking, as the number of blocks and consequently the number of different cache states are small.

To include the method cache in the UPPAAL model, we introduce an array of global, bounded integer variables, representing the blocks of the cache. It is assumed that the cache initially only contains the main method and is otherwise empty. We insert two additional locations right before and after the invoke location, modeling the time spent for loading the invoked and the invoking method, respectively (Figure 4.8).

Before invoking a method m_i or returning to a method m_i, the global state is modified by `access_cache(i)`. This UPPAAL procedure updates the global cache state and sets the variable `lastHit` to `true` if the access was a hit, and to `false` otherwise. Finally, the transition to the locations modeling the cache miss are guarded by `!lastHit`.

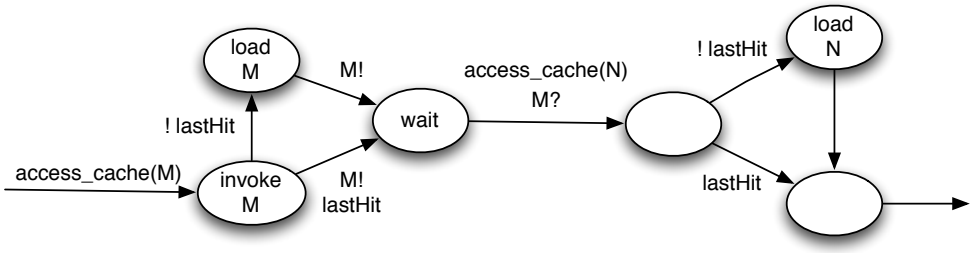

Figure 4.8: Modeling cache access when invoking method M from method N

For a N-block LRU cache using one block per method, the cache state is an array of N bounded integer variables, ranging from 0 to the number of methods. When the cache is accessed, we first test whether the first variable represents the method invoked. In this case, the method is in the cache and the cache state does not change. Otherwise, the invoked method is moved to the beginning of the cache, and all methods up to the invoked one are moved to the next position. If the invoked method is not in the cache, the method at the last position is removed from the cache and `lastHit` is set to `false` (Listing 4.1).

For variable block FIFO caches, we need to assume that the cache is initially empty. As this is not a safe approximation in general, we have to ensure that the first access to some method is actually a cache miss, for example by inserting a cache flush at the beginning of the main method (see section 3.3). The choice of the best position for inserting cache flushes has not been investigated during this thesis, but is certainly an important ingredient for using this technique.

The cache state is again represented by an array of bounded integer variables. For FIFO caches, the cache state does not change if the method is in the cache. Otherwise, the blocks of the invoked method are written into the beginning of the array, and all other blocks are moved back accordingly (Listing 4.2). The actual update procedure could be implemented using a pointer instead of moving all the cache blocks, but this pointer should not be part of the cache state, as this would increase the number of different cache states and therefore states which need to be explored.

```
const int EMPTY_TAG = NUM_METHODS;
int[0,NUM_METHODS] cache[NUM_BLOCKS] = { ROOT_METHOD, EMPTY_TAG, ... };
bool lastHit;
void access_cache(int mid) {
  lastHit = false;
  if(cache[0] == mid) {
    lastHit = true;
  } else {
    int i = 0;
    int last = cache[0];
    for(i = 0;
        i < NUM_BLOCKS - 1 && !lastHit;
        i++) {
      int next = cache[i+1];
      if(next == mid) {
        lastHit = true;
      }
      cache[i+1] = last;
      last = next;
    }
    cache[0] = mid;
  }
}
```

Listing 4.1: LRU cache simulation

```
const int NUM_BLOCKS[NUM_METHODS]    = { /* number of blocks per method */ };
const int EMPTY_TAG = NUM_METHODS;
int[0,NUM_METHODS] cache[NUM_BLOCKS] =
    { EMPTY_TAG, ..., ROOT_METHOD, EMPTY_TAG, ... };
bool lastHit;
void access_cache(int mId) {
  int i = 0;
  int sz = NUM_BLOCKS[mId];
  lastHit = false;
  for(i = 0; i < NUM_BLOCKS; i++) {
      if(cache[i] == mId) {
        lastHit = true;
        return;
      }
  }
  for(i = NUM_BLOCKS - 1; i >= sz; i--) {
     cache[i] = cache[i-sz];
  }
  for(i = 0; i < sz-1; i++) {
     cache[i] = EMPTY_TAG;
  }
  cache[i] = mId;
}
```

Listing 4.2: FIFO variable block cache simulation

Chapter 5

A WCET Analysis Tool for JOP

In the course of this thesis, a tool for the WCET analysis of high-integrity Java applications has been developed. It is integrated into the JOP tool chain, and supports both IPET and timed automata based WCET calculation. Currently, JOP's low-level timing model is the only one supported, though we plan to include other processors in the future. The tool is not only useful directly before deploying the application, but also aids the developer early during the implementation phase, by generating reports about the expected timing behavior.

By combining IPET and timed automata model checking into one tool, different degrees of precision can be used to calculate the WCET, trading verification time for the tightness of the WCET bound. Assuming that flow facts usually have a local scope, it is feasible to calculate the WCET of subtasks separately and simplify the problem by pre-calculating the WCET for local subgraphs of the CFG. This allows us to restrict more expensive analysis methods to *relevant* parts of the program, and to reduce the time needed for WCET analysis.

JOP's method cache is taken into account in both calculation methods. In the case of IPET, we use a simple static classification of the variable block method cache, as explained in Section 3.5. For the model checking approach, we use the cache simulation described in the previous chapter. For FIFO caches, the model checking approximation is only safe if we flush the cache at the beginning of the analyzed task.

5.1 WCET Tool Architecture

The goal of the WCET tool is to analyze Java methods used in the mission phase of high-integrity Java applications. Concurrency aspects such as synchronization or preemption, as well as calculating the worst-case memory consumption are

considered to be future work.

The Java language subset supported corresponds roughly to the one specified by the JOP high-integrity Java profile. JOP's Java implemented bytecodes are supported, as well as virtual method invocations. For the latter, we either analyze the subtype relations of all known classes, or use a dataflow analysis to determine the set of receiver types [Puf09]. To support dynamic dispatch, the application's main entry point has to be known, and all classes possibly used in the application need to be loaded.

The dataflow analysis also provides an automated loop bound analysis. Additionally, we extract flow fact annotations from the source code, similar to the ones used in [SP06]. Either way, it is necessary that for every loop the number of times the body is executed is bounded by a constant.

We do not support recursive method invocations, as their use is discouraged on Java processors due to the high runtime overhead, and they complicate the analysis considerably. Therefore, the static call graph, generated by analyzing the class hierarchy or using the results from the dataflow analysis, has to be acyclic. Using the results of the receiver type dataflow analysis, it is nevertheless possible to support certain uses of the popular delegator pattern (Listing 5.1).

In Java, there are exceptions which subclass `java.lang.Exception` and need to be declared explicitly, and *runtime errors*, which do not need to be declared. *Declared exceptions* are in principal not an issue from an analysis point of view, though tool support is still missing. However, the use of exception handlers which catch runtime errors is not supported, and neither are applications throwing these exceptions explicitly. It seems to contradict the idea of a predictable system to catch runtime errors during a task, as they are usually caused by programming errors, such as division by zero. Moreover, runtime errors are problematic from an analysis point of view, because arbitrary bytecodes can cause them to be thrown, complicating both microcode and program analysis.

Tool architecture The WCET tool is integrated into the JOP tool chain, and uses several facilities it provides. This also increases the confidence that the application analyzed is the one actually executed on the processor.

First, the Java application is compiled using an off-the-shelf Java compiler, producing the class files. In the WCET tool, we also load the corresponding source files, perform a basic check that they haven't been modified, and extract the WCET annotations.

All packages in the tool chain use the *BCEL* library [Dah01] to analyze, transform and attach analysis results to Java bytecode. The *JOP class loader* loads all classes possibly used by the application, and performs some basic transformations,

```
interface Reader {
      char getChar();
}
static class DelegatingReader implements Reader {
  private Reader reader;
  public DelegationReader(Reader impl) {
    reader = impl;
  }
  /* DFA analysis can prove that there is no recursion here,
     as the reader field's type is always ConstReader */
  public char getChar() {
      return reader.getChar();
  }
}
static class ConstReader implements Reader {
  public char getChar() {
    return 'x';
  }
}
main() {
  Reader r = new DelegationReader(new ConstReader());
  action(r);
}
void action(Reader r) {
  /* DFA analysis determines loop bound N=100 */
  for(int i = 0; i < 100; i++) {
      r.getChar();
  }
}
```

Listing 5.1: Delegator patterns and Dataflow Analysis

inserting JOP specific bytecodes. The JOP linker (*JOPizer*) collects all classes, and generates a binary file, suitable for execution on hardware or the simulator.

After loading all necessary classes and source code files, we optionally perform a dataflow analysis to find tight receiver type sets and additional loop bounds.

The WCET tool, whose high-level architecture is depicted in Figure 5.1, gets the classes, source files and dataflow analysis results as input.

The *frontend* package extracts the type hierarchy, the callgraph, the control flow graphs and the supergraph. For every control flow graph, we detect loop headers and map the result of the loop bound analysis to the bytecode. Additionally, flow annotations are extracted from the source files and mapped to the bytecode using a heuristic based on the line numbers associated with bytecodes.

The *jop* package provides timing information for JOP, and is used for calculating the execution time of single instructions and basic blocks, depending on the hardware configuration.

The actual WCET calculation is controlled by the *analysis* package, and is carried out by the *ipet* and *uppaal* modules. As motivated above, depending on the time we are willing to spend on the analysis, different methods for computing the WCET

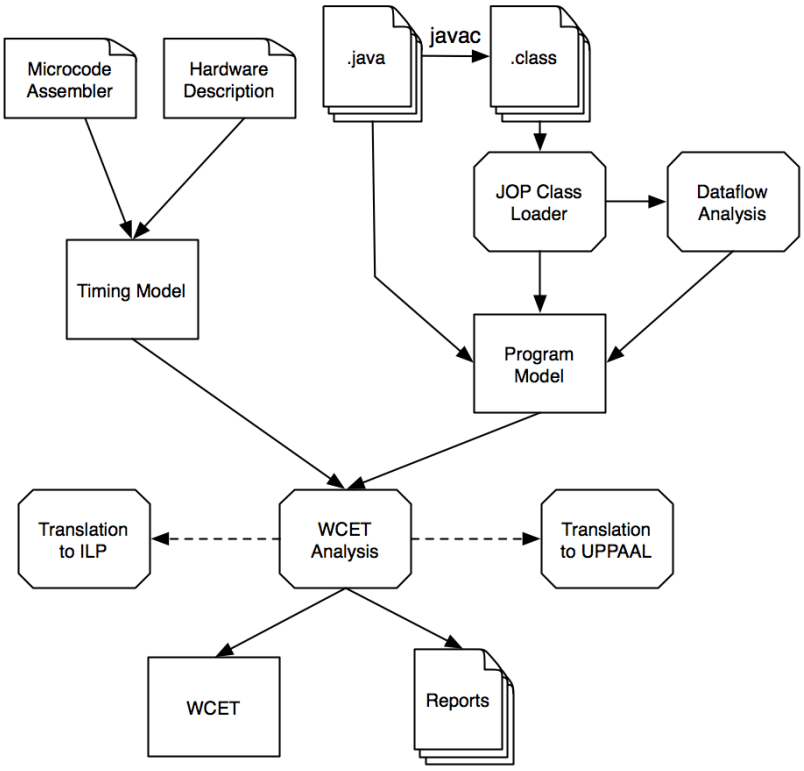

Figure 5.1: WCET tool architecure

bound are appropriate. For each method we want to analyze, we offer the following possibilities:

Local Analysis To calculate the WCET bound of a method, we first compute the bound of each method directly invoked during the execution. This kind of analysis is the fastest, but only supports intraprocedural flow facts. It is useful during development, for obtaining an upper bound for the UPPAAL search and could be used in interactive settings or for WCET-directed optimization.

Interprocedural Analysis using IPET To calculate the WCET bound of some method, we build the ILP from the corresponding supergraph. This method supports interprocedural flow facts, and therefore provides tighter WCET bounds. In the current implementation, interprocedural IPET is used to improve the cache analysis for variable block caches.

Model Checking To calculate the WCET bound of some method, we build the UPPAAL model from the corresponding supergraph. The

> WCET bound is then calculated using binary search, starting with the conservative approximations found by local analysis as upper bound.

The analysis module can be configured to use different calculation strategies. Local IPET is used to calculate bounds without considering the method cache, global IPET for static cache approximations and UPPAAL for dynamic cache simulations.

For the static cache approximation, it is possible to calculate points where inserting cache flushes benefits the analysis, and then assume that all cache loads will occur at invoke instructions only.

Optionally, the tool tries to speed up the UPPAAL analysis. Most important, the analysis time can be reduced considerably by first simplifying the supergraph. Currently, we pre-compute the WCET of loops without method invocations and of leaf methods. It is also possible to turn on UPPAAL's convex hull approximation, though in this case, no trace for the worst case path will be available.

```
305          |         public static void loop() {
306          |
307          |                 int cmd;
308          |
309  [9757]  |                 Msg.loop();                                    // for exact msg
310  [6035]  |                 Triac.loop();                                // should be also exact
311          | // TODO measure jitter with osci
312          |
313  [18]    |                 if (Msg.available) {
314  [179]   |                         cmd = Msg.readCmd();
315  [3067]  |                         if (!handleMsg(cmd)) {
316  [2925]  |                                 handleRest(cmd);
317          |                                 if (state == BBSys.MS_SERVICE) { // never called in
318  [18855] |                                     doService();                 // never return!
319          |                                 }
320          |                         }
321  [21]    |                         lastMsgCnt = 0;
322          |                 } else {
323          |                         chkMsgTimeout();
324          |                 }
325          |
326  [20]    |                 if (blinkCnt==100) {
327  [458]   |                         Timer.wd();
328  [17]    |                         blinkCnt = 0;
329          |                 }
330  [32]    |                 ++blinkCnt;
331          |
332  [554]   |                 int used = Timer.usedTime();
333  [17]    |                 if (maxTime<used) maxTime = used;
334  [2538]  |                 JopSys.benchLoop();
335  [21]    |         }
336          | }
```

Figure 5.2: WCET report for the *Kippfahrleitung* application

The *report* module finally generates HTML reports (see Figure 5.2 for an example). The IPET backend records execution frequencies, which are later mapped backed to the source code for reporting. Back-annotation for UPPAAL is not yet implemented, but possible in principle. The source code is annotated with timing informations, displaying the time spend in each statement. Additionally, the control flow graphs labeled with the worst case execution frequencies are visualized.

5.2 Evaluation

In this section, we will present some experimental results obtained with our new tool. The first problem set consists of small benchmarks to evaluate the model checker's performance for intraprocedural analysis. The second set comprises embedded applications with more than one method, used for evaluating analysis times and the quality of the cache approximations.

First, we list the benchmarks along with some metrics, followed by a performance evaluation of the UPPAAL model checker. Finally, we conduct several experiments with JOP's method cache, using the model checker for exact simulation. For all experiments, the timing model for the *dspio board* [1] was used.

Problem Set The first problem set consists of a couple of small single-method benchmarks, with the exception of *ShortCrc* provided by the Mälardalen Real-Time Research center [2]. We vary the input data's size for those algorithms to investigate the relation between the number of loop iterations, and the time needed for model checking the method.

The second set consists of larger benchmarks, mainly taken from real world applications. The `Kfl` benchmark has a comparatively high complexity, so we also consider the two subtasks, `Kfl.Triac` and `Kfl.Msg`.

Table 5.1 lists the size of the tasks under consideration (source and compiled), and the number of methods. Additionally, we list the size of the largest method in bytes, which determines a lower bound for the size of the method cache. Typically WCET analysis targets single tasks, so the size of the real world applications seems to be reasonable. On the other hand, we would definitely benefit from a larger and varying set of benchmarks.

Table 5.2 compares the measured execution times and the computed WCET estimates. For this experiment we used the default JOP configuration for the *dspio* board, using a variable block method cache with 16 cache blocks, each spanning 64 words. The WCET was computed using the IPET method, once assuming a single block method cache (*WCET SB*), and once using the static method cache approximation (*WCET VAR*). Pessimistic estimations partly result from conservative flow facts (*BubbleSort*) or data dependent flow (*CRC*), but are also due to the fact that the measurements do not cover all possible execution paths (*Kfl*).

Performance Analysis We have evaluated the time needed to estimate the WCET using the UPPAAL model checker for the benchmarks listed above. We

[1] `http://ti.tuwien.ac.at/rts/teaching/soc/dspio`
[2] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

Problem	Description	LOC	Bytecode Size	Number of Methods (size of largest)
ShortCrc	Compact CRC (4 bytes)	8	54	1
DiscreteCosineTransform	FDCT ($8x8$ array)	223	953	1
Fibonacci(N)	Fibonacci number ($O(N)$)	37	39	1
BubbleSort(N)	Bubble sort ($O(N^2)$)	63	78	1
CRC	Multi-method CRC (40 bytes)	154	394	5 (252)
LineFollower	A simple line-following robot	89	226	9 (74)
Lift	Lift controller	635	1206	13 (215)
Kfl	*Kippfahrleitung* application	1366	2533	46 (252)
Kfl.Triac	*Kfl* sensor and control task		977	14 (191)
Kfl.Msg	*Kfl* communication task		573	8 (250)

Table 5.1: Problem sets for evaluation

Problem	Measured	WCET VAR	WCET SB	Pessimism Ratio
ShortCrc	1449	1513		1.04
DiscreteCosineTransform	19124	19131		1.00
Fibonacci(30)	1127	1138		1.01
BubbleSort(100)	1262717	1852573		1.47
CRC	191825	383026	469726	2.00
LineFollower	2348	2369	2610	1.01
Lift	5446	8344	8897	1.53
Kfl	10411	39555	49744	3.80
Kfl.Triac	2209	5252	6139	2.38
Kfl.Msg	4099	8216	11976	2.00

Table 5.2: Measured Execution Time and WCET calculated using IPET

measure the total time spent in the binary search, and the maximum time spent in one run of the UPPAAL verifier. All experiments were carried out using an Intel Core Duo 1.83Ghz with 2GB RAM on `darwin-9.5`. For model checking, we used UPPAAL 4.0, the linear programs were solved by `lp_solve 5.5`.

For the first problem set, the analysis times for the ILP approach are too small to be measured reliably, so we only present the times for model checking, and also evaluate UPPAAL's convex hull approximation.

Summarized in Table 5.3, we observe that UPPAAL handles the analysis of single methods well, as long as the loop bounds don't get too large. The model checker was able to analyze 300000 inner loop iterations within 5-10 seconds, for both the *Fibonacci* and *BubbleSort* benchmark. For the *Fibonacci* and *BubbleSort* benchmark, convex hull approximation (denoted by *CH* in the table) speeds up analysis by 14 respectively 21 percent on average, which is a small, albeit useful improvement.

For the second problem set, we measure the time needed for local IPET, inter-

Problem	N	WCET	Verify	Verify CH	Search	Search CH
ShortCrc		1,513	0.05	0.05	0.58	0.57
FDCT		19,143	0.06	0.06	0.76	0.73
Fibonacci	3000	102,177	0.11	0.11	1.79	1.66
Fibonacci	30000	1,020,143	0.60	0.47	11.94	9.79
Fibonacci	300000	11,700,143	5.88	4.98	135.28	114.13
BubbleSort	100	1,852,573	0.35	0.26	7.22	5.66
BubbleSort	200	7,445,371	1.20	0.94	27.90	21.96
BubbleSort	400	29,770,571	4.87	3.85	122.00	96.75
BubbleSort	600	66,975,771	11.12	8.62	288.96	227.43

Table 5.3: Analysis times (UPPAAL) for problem set 1

Problem	IPET solvertime		UPPAAL Verifier Time			UPPAAL Search Time		
	Local	Global	N	CH	SIMP	N	CH	SIMP
CRC	0.00	0.00	0.87	1.30	0.19	16.46	14.26	3.49
LineFollower	0.00	0.00	0.07	0.07	0.07	0.86	0.89	0.83
Lift	0.01	0.03	0.18	0.17	0.09	2.43	2.30	1.11
Kfl.Msg	0.01	0.01	0.28	0.25	0.14	3.95	3.41	1.98
Kfl.Triac	0.01	0.04	0.46	0.39	0.33	5.62	5.03	4.32
Kfl	0.04	0.18	111.00	98.62	16.22	1796.55	1555.29	250.54

Table 5.4: Analysis times for problem set 2

procedural IPET, UPPAAL model checking and for model checking a locally simplified problem (Table 5.4). The simplified problem is obtained by first analyzing invocation-free inner loops and leaf methods separately, and then replacing them by pseudo locations annotated with the WCET of the corresponding program fragment. The columns *N* and *CH* list the time needed for UPPAAL computation without and with convex hull approximation, respectively, while *SIMP* is the time needed for the simplified problem using convex hull approximation.

For the non-simplified problems, UPPAAL needed a considerable amount of time for the *CRC* (16 seconds) and *Kfl* (1797 seconds) problem. For the simplified problems on the other hand, UPPAAL solved *CRC* almost instantly, and needed "only" 250 seconds for the *Kfl* problem. This suggests that local simplifications should be further investigated, and is the reason we used the simplification when experimenting with method cache simulations below.

Unsurprisingly, the ILP solver managed to handle all problems easily, as no interprocedural flow facts were present. The cumulative time spent in the solver in the per-method approach is below 0.1 seconds for all problems, so it seems this approach can be used for interactive evaluation and similar tasks requiring instant response.

Evaluation of the Method Cache Approximations Now we present the results of computing the WCET of some of the programs in the second problem set using different method cache configurations. For each problem, we vary the size of the cache depending on the size of the program. The UPPAAL models are simplified in a preprocessing step as described in the previous section.

The following cache approximations were evaluated:

- *IPET VAR*: Variable block FIFO method cache, using IPET and the *all-fit code region* approximation.
- UPPAAL *VAR*: Variable block FIFO method cache, using UPPAAL and the corresponding cache simulation.
- UPPAAL *LRU*: On-block-per-method LRU cache, using UPPAAL and the corresponding cache simulation.
- UPPAAL *FIFO*: On-block-per-method FIFO cache, using UPPAAL and the corresponding cache simulation.

For UPPAAL *FIFO* and UPPAAL *VAR*, an initially empty cache is assumed. That is, the WCET bound is only safe if a cache flush is inserted before executing the method. The results of the experiments are given in Table 5.5, 5.6, 5.7, 5.8 and 5.9.

We observe that in in general, the variable block method cache outperforms the fixed blocked caches if the same cache size is assumed. This is because a block in a fixed block cache has to be at least as large as the largest method possibly invoked.

Comparing the UPPAAL and IPET estimations for the variable block cache, the latter does not work well if the cache is small, and delivers slightly worse results even when all methods fit into the cache, due to the fact that cache loads have to be assumed to happen on return. On the other hand, the IPET approximation is still good enough for most purposes.

In the *Kfl.Triac* benchmark, the time spend in the model checker varies significantly depending on the cache implementation, where the time needed for fixed block LRU cache is significantly lower than that needed for FIFO caches.

Cache Size	Approximation	Blocks x Block Size	WCET	Verify (s)	Search (s)	Ratio WCET
64						
	IPET VAR	8 x 8	469726		0.01	1.00
	UPPAAL VAR	8 x 8	469652	0.36	6.78	1.00
128						
	UPPAAL LRU	1 x 128	469726	0.31	5.54	*1.00*
	IPET VAR	16 x 8	383026		0.01	**0.82**
	UPPAAL VAR	16 x 8	382965	0.32	5.99	**0.82**
256						
	IPET VAR	16 x 16	383026		0.01	**0.82**
	UPPAAL VAR	16 x 16	382965	0.32	5.82	**0.82**
	UPPAAL LRU	2 x 128	383142	0.29	5.18	**0.82**
	UPPAAL FIFO	2 x 128	394179	0.31	5.81	0.84
512						
	IPET VAR	16 x 32	383026		0.01	0.82
	UPPAAL VAR	16 x 32	382965	0.31	5.71	0.82
	UPPAAL LRU	4 x 128	383045	0.31	5.84	0.82
	UPPAAL FIFO	4 x 128	383142	0.30	5.27	0.82

Table 5.5: Cache analysis: *Cyclic redundancy check*

Cache Size	Approximation	Blocks x Block Size	WCET	Verify (s)	Search (s)	Ratio WCET
64						
	UPPAAL LRU	1 x 64	2610	0.10	1.04	*1.00*
	IPET VAR	8 x 8	2582		0.01	0.99
	UPPAAL VAR	8 x 8	2443	0.09	1.07	**0.94**
128						
	IPET VAR	16 x 8	2411		0.01	0.92
	UPPAAL VAR	16 x 8	2368	0.09	1.05	**0.91**
	UPPAAL LRU	2 x 64	2453	0.11	1.00	0.94
	UPPAAL FIFO	2 x 64	2471	0.09	1.04	0.95
256						
	IPET VAR	16 x 16	2411		0.01	0.92
	UPPAAL VAR	16 x 16	2368	0.10	1.09	**0.91**
	UPPAAL LRU	4 x 64	2453	0.09	0.97	0.94
	UPPAAL FIFO	4 x 64	2453	0.10	1.00	0.94

Table 5.6: Cache analysis: Line Following Robot

Cache Size	Approximation	Blocks x Block Size	WCET	Verify (s)	Search (s)	Ratio WCET
128						
	UPPAAL LRU	1 x 128	8897	0.13	1.63	*1.00*
	IPET VAR	16 x 8	8595		0.02	0.97
	UPPAAL VAR	16 x 8	8400	0.54	2.26	**0.94**
256						
	IPET VAR	16 x 16	8595		0.01	0.97
	UPPAAL VAR	16 x 16	8355	0.21	1.94	**0.94**
	UPPAAL LRU	2 x 128	8422	0.14	1.50	0.95
	UPPAAL FIFO	2 x 128	8422	0.13	1.49	0.95
512						
	IPET VAR	16 x 32	8466		0.04	0.95
	UPPAAL VAR	16 x 32	8343	0.14	1.91	**0.94**
	UPPAAL LRU	4 x 128	8355	0.11	1.47	**0.94**
	UPPAAL FIFO	4 x 128	8411	0.12	1.47	0.95
1024						
	IPET VAR	16 x 64	8466		0.04	0.95
	UPPAAL VAR	16 x 64	8343	0.15	1.62	**0.94**
	UPPAAL LRU	8 x 128	8344	0.14	1.58	**0.94**
	UPPAAL FIFO	8 x 128	8355	0.12	1.52	**0.94**

Table 5.7: Cache analysis: *Lift Controller*

Cache Size	Approximation	Blocks x Block Size	WCET	Verify (s)	Search (s)	Ratio WCET
128						
	UPPAAL LRU	1 x 128	11976	0.23	2.08	*1.00*
	IPET VAR	16 x 8	11022		0.01	0.92
	UPPAAL VAR	16 x 8	8234	0.89	11.33	**0.69**
256						
	IPET VAR	16 x 16	8216		0.01	0.69
	UPPAAL VAR	16 x 16	8152	0.80	11.36	**0.68**
	UPPAAL LRU	2 x 128	9492	0.36	4.97	0.79
	UPPAAL FIFO	2 x 128	10444	1.12	7.64	0.87
512						
	IPET VAR	16 x 32	8216		0.01	0.69
	UPPAAL VAR	16 x 32	8152	0.80	11.29	**0.68**
	UPPAAL LRU	4 x 128	8153	0.74	10.59	**0.68**
	UPPAAL FIFO	4 x 128	8153	0.79	11.08	**0.68**

Table 5.8: Cache analysis: *Kippfahrleitung communication task*

Cache Size	Approximation	Blocks x Block Size	WCET	Verify (s)	Search (s)	Ratio WCET
128						
	UPPAAL LRU	1 x 128	6139	0.80	9.93	**1.00**
	IPET VAR	16 x 8	5564		0.03	0.91
	UPPAAL VAR	16 x 8	5105	7.55	93.86	**0.83**
256						
	IPET VAR	16 x 16	5537		0.06	0.90
	UPPAAL VAR	16 x 16	5039	33.52	416.09	**0.82**
	UPPAAL LRU	2 x 128	5806	0.99	12.77	0.95
	UPPAAL FIFO	2 x 128	5806	1.17	15.47	0.95
512						
	IPET VAR	16 x 32	5252		0.04	0.86
	UPPAAL VAR	16 x 32	5025	32.43	428.71	**0.82**
	UPPAAL LRU	4 x 128	5266	1.36	17.96	0.86
	UPPAAL FIFO	4 x 128	5422	3.70	49.30	0.88
1024						
	IPET VAR	16 x 64	5252		0.04	0.86
	UPPAAL VAR	16 x 64	5025	31.78	426.86	**0.82**
	UPPAAL LRU	8 x 128	5026	6.82	92.48	**0.82**
	UPPAAL FIFO	8 x 128	5041	31.35	414.94	**0.82**

Table 5.9: Cache analysis: *Kippfahrleitung sensor and control task*

Chapter 6

Conclusion

In this thesis, techniques for the WCET analysis of high-integrity Java applications were investigated, presenting and evaluating an analysis tool for the Java processor JOP.

WCET calculation takes a high-level, abstract model of the program, focusing on control flow, and a low-level model of the execution platform as input. After the model extraction, the main challenges are to include flow facts, static context dependencies, and dynamic dependencies on the execution history in the calculation, reducing the gap between the actual and the statically computed WCET.

IPET is an established and fast technique for calculating the WCET. Relative execution frequency constraints and a static call-context dependencies can be encoded efficiently, while on the other hand dependencies on the execution history, or facts which depend on the order of execution, are difficult to incorporate into this static timing model.

Timed automata are an alternative, powerful modeling method. Beside their support for an expressive notion of time, the allow to model the execution path set using finite-state automata, and the simulation of global low-level timing dependencies, e. g., those caused by instruction caches. The analysis time is a significant issue when using timed automata, and so it seems to be most important to simplify the model as far as possible.

We combined IPET and model checking, allowing us to get rid of large inner loops in a preprocessing step. While this approach was quite effective for our purposes, it seems that a scalable model checking approach ultimately depends on the ability to solve the problem in a compositional way, abstracting away details which are not of immediate interest. For scheduling analysis, for example, the model should only reflect synchronization and concurrency aspects, but abstract away as many sequential details as possible.

There seem to be a lot of optimization opportunities for model checking based WCET calculation. For the analysis of sequential code fragments, it is not even necessary to use the strong timed automata model, and using a model checker with a simpler notion of time would be sufficient, potentially improving performance. We note that:

- All states which only differ with respect to the elapsed time can be merged, keeping only the state with the largest time stamp.
- The WCET model checking might benefit from a carefully chosen search order, which can be easily derived by analyzing the control flow graphs.
- For sequential analysis, it should be possible to summarize the effect of executing some automaton in some initial state, as only the set of final states with the corresponding elapsed time matters.

Regarding the tool implementation, it turned out that JOP is indeed a good target platform for WCET analysis. The only global low-level effect, method caches, can be approximated rather effectively. If the variable block method cache is sufficiently large, the static approximation worked very well, while the UPPAAL simulation showed even better results for smaller variable block caches. Small per-thread cache areas could be beneficial for the analysis of preemptive systems, were the context switch cost has to be taken into account. The all-fit region approximation would benefit from a better analysis proving there is no cache overflow, instead of the currently used, simple heuristic, summing the size of all methods possibly invoked.

The analysis times for IPET where below one second for all problems, although we used decision variables for the static cache approximation. However, no interprocedural flow facts or context dependencies beside the ones for cache approximation were present, so we cannot safely conclude that analysis time is not an issue when using IPET in general. The timed automata calculation was fast enough for program fragments spanning a few methods, but took too long to calculate cache approximation for the largest of the test applications. The size of the cache, and the replacement strategy used, also has a significant impact on the analysis time, and even using simplifications we only managed tasks with up to 14 methods when using a realistic cache model.

In the future, we want the tool to support other Java processors, such as *jamuth* [UW07], and hardware extensions to JOP, and improve the quality of the WCET bound for the multiprocessing extension. Finally, the high-level analysis should be improved, reducing the needs for loop bound annotations further and taking infeasible paths and context dependencies into account.

Appendix A

Obtaining and Using the WCET Tool

The new WCET tool is part of the GPL-licensed JOP project.

First, obtain the source via *CVS*,[1] as explained in the download area of JOP's webpage [2].

The tool chain's source is located in `java/tools/src` and is build with

```
make tools
```

After a successful build, the JOP tool library should reside at `java/tools/dist/lib/jop-tools.jar`. The necessary third-party Java libraries [3] [4] [5] [6] [7] [8] are included in the CVS tree, and need to be on the *classpath*, along with the JOP tools archive:

```
CLASSPATH = java/lib/bcel-5.1.jar:java/lib/jakarta-regexp-1.3.jar:\
            java/lib/lpsolve55j.jar:java/lib/log4j-1.2.15.jar:\
            java/lib/jgrapht-jdk1.5.jar:java/lib/velocity-dep-1.5.jar:\
            java/tools/dist/lib/jop-tools.jar
```

Additionally, the WCET tool needs the free `lp_solve` ILP solver library [9], and

[1] http://www.nongnu.org/cvs/
[2] http://www.jopdesign.com/
[3] BCEL Bytecode Engineering Library: http://jakarta.apache.org/bcel/
[4] Jakarta Regexp: http://jakarta.apache.org/regexp/
[5] lp_solve Java bindings: http://lpsolve.sourceforge.net/5.5/Java/README.html/
[6] Apache log4j: http://logging.apache.org/log4j/
[7] JGraphT free Java graph library: http://jgrapht.sourceforge.net/
[8] The apache Velocity project: http://velocity.apache.org/
[9] http://sourceforge.net/projects/lpsolve

Option	Description
`app-class`	the name of the class containing the main entry point of the application
`target-method`	the name (optional: class,signature) of the method to be analyzed
`dataflow-analysis`	whether dataflow analysis should be performed
`outdir`	parent directory for generating output
`cp`	the classpath for the analyzed application
`sp`	the sourcepath for the analyzed application
`debug`	whether to print verbose debugging messages

Table A.1: General options

Option	Description
`cache-impl`	method cache implementation (no, LRU fixed block or FIFO variable block)
`cache-size-words`	size of the cache in words
`cache-blocks`	number of cache blocks
`cache-approx`	static cache approximation (no or all-fit approximation)

Table A.2: Options for analyzing the method cache

Option	Description
`uppaal`	whether UPPAAL should be used for WCET calculation
`dyn-cache-approx`	dynamic cache approximation (no or UPPAAL simulation)
`uppaal-verifier`	path to UPPAAL's verifier binary (`verifyta`)
`uppaal-collapse-leaves`	pre-calculate inner loops and leaf methods to speed up simulation
`uppaal-convex-hull`	use UPPAAL's convex hull approximation

Table A.3: Options for the UPPAAL model checker

the UPPAAL model checker [10].

The main entry point of the WCET analyzer is `com.jopdesign.wcet.WCETAnalyis`. Supplying `-help` as a command line argument provides an overview of the available options. Options can bet set by specifying a property file, or as command line arguments.

The WCET tool takes the main entry point of a real-time Java application, and the name of the method to be analyzed. It then performs a fast, local IPET analysis, followed by a more precise one, either using intraprocedural IPET, or the UPPAAL model checker. The options accepted by the tool are listed in Table A.1, A.2, A.3 and A.4.

The tool outputs the requested WCET bounds, and generates HTML reports in the configured report directory.

[10] http://www.uppaal.com/

Option	Description
`reportdir`	the directory to write reports into
`templatedir`	directory with custom templates for report generation
`error-log`	the error log file
`info-log`	the info log file
`program-dot`	if graphs should be generated, the path to the `dot` binary
`dump-ilp`	whether the LP problems should be dumped to files

Table A.4: Options for generating reports

References

[Bal96] F. Balarin. Approximate reachability analysis of timed automata. In *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, page 52, Washington, DC, USA, 1996. IEEE Computer Society. 47

[BBW00] Guillem Bernat, Alan Burns, and Andy Wellings. Portable worst-case execution time analysis using java byte code. In *Proc. 12th EUROMICRO Conference on Real-time Systems*, Jun 2000. 25, 26

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004. 45

[BDV03] Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide to the use of the ada ravenscar profile in high integrity systems. Technical Report Technical Report YCS-2003-348, University of York (UK), 2003. 5, 14

[BDV04] Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems. *Ada Lett.*, XXIV(2):1–74, 2004. 13

[BEG+08] Dani Barkah, Andreas Ermedahl, Jan Gustafsson, Björn Lisper, and Christer Sandberg. Evaluation of automatic flow analysis for wcet calculation on industrial real-time system code. In *ECRTS '08: Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, pages 331–340, Washington, DC, USA, 2008. IEEE Computer Society. 27

[BGB+] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. The real-time specification for java 1.0.2. Available at: http://www.rtsj.org/specjavadoc/book_index.html. 9

[BKHO+08] Thomas Bogholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim G. Larsen. Model-based schedulability analysis of safety critical hard real-time java programs. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)*, pages 106–114, New York, NY, USA, 2008. ACM. 47, 48

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, Berlin, 1 edition, September 2007. 34

[BS88] T. P. Baker and A. Shaw. The cyclic executive model and ADA. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 120–129, 1988. 4

[BY03] Johan Bengtsson and Wang Yi. On clock difference constraints and termination in reachability analysis of timed automata, 2003. 44

[BY04] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. 2004. 38

[CL00] Franck Cassez and Kim Larsen. The impressive power of stopwatches. In *In Proc. of CONCUR 2000: Concurrency Theory*, pages 138–152. Springer, 2000. 48

[Dah01] Markus Dahm. Byte code engineering with the BCEL API. Technical report, Freie Universitat Berlin, April 2001. 55

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley Professional, Boston, Mass., 2005. 5

[Hav97] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997. 21

[HBW02] Erik Yu-Shing Hu, Guillem Bernat, and Andy Wellings. A static timing analysis environment using Java architecture for safety critical real-time systems. In *WORDS '02: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, page 77, Washington, DC, USA, 2002. IEEE Computer Society. 26

[HK07] Trevor Harmon and Raymond 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. 26

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:394–406, 1994. 38

[KKP+05] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Ingomar Wenzel. Wcet analysis: The annotation language challenge. In *Proc. 7th International Workshop on Worst-Case Execution Time Analysis*, Pisa, Italy, July 2005. 23

[KWK02] Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-Java: A high integrity profile for real-time Java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 131–140. ACM Press, 2002. 5, 14

[LY97] Kim G. Larsen and Wang Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *In Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, 1997. 44

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999. 6, 8

[Mue00] Frank Mueller. Timing analysis for instruction caches. *Real-Time Syst.*, 18(2-3):217–247, 2000. 31

[Mye81] Eugene M. Myers. A precise inter-procedural data flow algorithm. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 219–230, New York, NY, USA, 1981. ACM. 21

[OL] Martin Ouimet and Kristina Lundqvist. Verifying execution time using the TASM toolset and UPPAAL. Technical Report Embedded Systems Laboratory Technical Report ESL-TIK-00212, Embedded Systems Laboratory Massachusetts Institute of Technology. 47

[Par93] Chang Yun Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Syst.*, 5(1):31–62, 1993. 24

[PB01] Peter Puschner and Guillem Bernat. WCET analysis of reusable portable code. In *ECRTS '01: Proceedings of the 13th Euromicro Conference on Real-Time Systems*, page 45, Washington, DC, USA, 2001. IEEE Computer Society. 20

[Pit08] Christof Pitter. Time-predictable memory arbitration for a Java chip-multiprocessor. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)*, pages 115–122, New York, NY, USA, 2008. ACM. 29

[PS97] Peter Puschner and Anton Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13(1):67–91, Jul. 1997. 33, 34, 36

[PS08] Wolfgang Puffitsch and Martin Schoeberl. Non-blocking root scanning for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, September 2008. 29

[Puf09] Wolfgang Puffitsch. Supporting wcet analysis with data flow analysis of java bytecode. Research Report 16/2009, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2009. 27, 55

[PW01] Peter Puschner and Andy Wellings. A profile for high integrity real-time Java programs. In *4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001. 14, 15, 19

[RGBW07] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Journal of Real-Time Systems*, 37(2):99–122, Nov. 2007. 31

[Sch04a] Martin Schoeberl. Restrictions of Java for embedded real-time systems. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 93–100, Vienna, Austria, May 2004. IEEE. 8, 15

[Sch04b] Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer. 31

[Sch05] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005. 27, 29

[SP06] Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press. 55

[SR94] K.G. Shin and P. Ramanathan. Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6–24, Jan 1994. 3

[SSTR07] Martin Schoeberl, Hans Sondergaard, Bent Thomsen, and Anders P. Ravn. A profile for safety critical Java. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, pages 94–101, Santorini Island, Greece, May 2007. IEEE Computer Society. 9

[The02] Henrik Theiling. Ilp-based interprocedural path analysis. In *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software*, pages 349–363, London, UK, 2002. Springer-Verlag. 37

[UW07] Sascha Uhrig and Jörg Wiese. jamuth: an IP processor core for embedded Java real-time systems. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 230–237, New York, NY, USA, 2007. ACM Press. 67

List of Figures

List of Tables

Listings