Predicting Time and Memory Demands of Object-Oriented Programs

Patrik Persson

.....

Department of Computer Science
Lund Institute of Technology
Lund University

Department of Computer Science
Lund Institute of Technology
Lund University
Box 118
SE-221 00 Lund
Sweden

E-mail: Patrik.Persson@cs.lth.se
WWW: http://www.cs.lth.se/~patrik/

ISSN 1404-1219
Dissertation 12, 2000

Abstract

Embedded computer systems are subject to a multitude of requirements. These include real-time requirements, that is, such computers must respond to external events within limited time. Many systems, such as satellites and telephone switches, must also operate unattended for long periods of time. They must not fail due to defective software.

Modern object-oriented programming languages, particularly Java, offer type safety, automatic memory management (garbage collection), dynamic loading of code, and object-oriented abstraction mechanisms. All these features, designed to increase software quality and flexibility, are highly desirable in embedded systems. Yet object-oriented languages are often avoided in such applications. One reason for this is that previous techniques for worst-case execution time (WCET) predictions are unsuitable for object-oriented languages. WCET predictions are necessary to guarantee fulfilment of real-time requirements.

We present techniques for predicting the WCET of programs in object-oriented languages. We also show how to predict the amount of memory required by an object-oriented program; such information is required for safe scheduling of real-time garbage collection. The techniques are mainly automatic (assisted by manual annotations) and benefit from integration with a compiler. They are being implemented in an interactive development environment for a subset of the Java programming language.

The presented techniques make object-oriented programming languages with garbage collection more predictable and thus more appropriate for hard real-time systems. The declarative implementation technique (reference attributed grammars) facilitates a clear and concise implementation suitable for our interactive environment. This interactivity allows timing problems, requiring revision of design or requirements, to be detected early.

Preface

This is a thesis for the degree of Licentiate in Engineering, a Swedish degree between MSc and PhD. The thesis consists of an introduction and three papers:

I. P. Persson and G. Hedin. Interactive Execution Time Predictions using Reference Attributed Grammars. *Proceedings of WAGA'99: Second Workshop on Attribute Grammars and their Applications*, Amsterdam, The Netherlands, March 1999.

II. P. Persson. Live Memory Analysis for Garbage Collection in Embedded Systems. *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*, Atlanta, Georgia, May 1999. © 1999 ACM.

III. P. Persson and G. Hedin. An Interactive Environment for Real-Time Software Development. *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages (TOOLS Europe 2000)*, St. Malo, France, June 2000. © 2000 IEEE.

Acknowledgments

The work presented in this thesis has been carried out within the Software Development Environments group at the Department of Computer Science, Lund University. I am particularly grateful to three persons who have guided me in this work. My advisor Görel Hedin has provided good ideas and comprehensive, yet precise, feedback. Klas Nilsson, my assistant advisor, has been a great source for exciting discussions. Boris Magnusson, the leader of our group, has frequently helped me to put things into perspective.

The remaining members of our research group have contributed with ideas, discussion, and good company. Your support is most appreciated. Thank you Anders Ive, Roger Henriksson, Ulf Asklund, Anders Nilsson, Eva Magnusson, Daniel Einarsson, Jonas Skeppstedt, Mathias Haage,

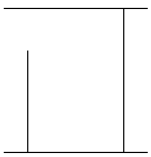
Sven Robertz, and Jonas Persson. I am also grateful to Elizabeth Bjarnason, a previous member of our group, for her great work on APPLAB.

The value of enjoying ones work cannot be overemphasized. I would like to thank everyone at the Department of Computer Science for making it such a pleasant place to work in. Erik Persson enlightened me on some philosophical references, and Jesper Jansson helped me with some nice LaTeX tricks. Fredrik Dahlstrand at the Department of Information Technology read and commented on early drafts of this thesis; together with Bengt Öhman he also regularly ensured that I retained some practical engineering skills and essential nutrition during my work.

This work is part of a cooperation with the Department of Automatic Control, Lund University. Anton Cervin and Johan Eker have given me valuable insight into actual real-time control applications. Karl-Erik Årzén has revised my view on deadlines on more than one occasion.

David Whalley and Chris Healy at Florida State University showed great hospitability during my visit there, and I learned much from them. The Swedish WCET network has been a forum for many stimulating discussions.

This work has been financially supported by ARTES (A network for Real-Time research and graduate Education in Sweden) and SSF (the Swedish Foundation for Strategic Research).



Contents

Introduction **The Hare, the Tortoise, and Plato** **1**
1 Real-Time Systems 2
2 Scheduling of Real-Time Tasks 3
3 Predicting Execution Times 4
4 Approach and Contributions 14
References 17

Paper I **Interactive Execution Time Predictions Using Reference Attributed Grammars** **21**
1 Introduction 22
2 Timing Properties of Real-Time Systems 23
3 An Interactive WCET Prediction Tool 26
4 Conclusions 32
References 34

Paper II **Live Memory Analysis for Garbage Collection in Embedded Systems** **37**
1 Introduction 38
2 Terminology and General Approach 39
3 Live Memory Analysis 43
4 The Environment 49
5 Discussion 51
6 Conclusions 53
References 54

Paper III **An Interactive Environment for Real-Time Software Development** **59**
1 Introduction 60
2 Aspects of Real-Time Software Development . . 61
3 Predictable Real-Time Java 62
4 The Skånerost Environment 66
5 Conclusions 71
References 72

Introduction

— The Hare, the Tortoise, and Plato

A hare one day ridiculed the short feet and slow pace of the Tortoise, who replied, laughing: "Though you be swift as the wind, I will beat you in a race." The Hare, believing her assertion to be simply impossible, assented to the proposal; and they agreed that the Fox should choose the course and fix the goal. On the day appointed for the race the two started together. The Tortoise never for a moment stopped, but went on with a slow but steady pace straight to the end of the course. The Hare, lying down by the wayside, fell fast asleep. At last waking up, and moving as fast as he could, he saw the Tortoise had reached the goal, and was comfortably dozing after her fatigue.

Slow but steady wins the race.

— Aesop, *The Hare and the Tortoise*

This thesis treats the development of software for embedded real-time computer systems. The term *embedded* means that the computer is an integral component of some system, rather than being a system of its own (such as a PC). The term *real-time* means that the system operates under special timing requirements, which we will soon elaborate on.

Embedded real-time systems are increasingly important to our society. They exist both in consumer products and industrial applications, ranging from mobile phones to industrial robots. Some of these systems are safety-critical in that a failure can jeopardize people's safety, such as an aircraft engine control system.

Programming of real-time systems is, in some respects, different from other types of programming. The timing requirements, typically expressed as limitations on program execution time, must be verified. As we will discuss in Section 3, it is not as simple as running the program and measuring how long time it takes. It is often necessary to *predict* how long the execution time of a program can possibly be.

However, programming of real-time systems also has much in common with programming in general. In particular, it is desirable to use modern object-oriented programming languages for the programming of these systems. Object-oriented languages are often avoided in embedded real-time systems, partly because they complicate the execution time predictions just mentioned.

The specific topic of this thesis is techniques and tools for the prediction of execution times, with focus on object-oriented languages. In Section 1, the area of real-time systems is presented, followed by an introduction to real-time task scheduling in Section 2. Various approaches to, and aspects of, execution time prediction are presented in Section 3 along with an overview of previous work in the area. Finally, in Section 4, the approach and contributions of the papers in this thesis are outlined, including some directions for future research.

1 Real-Time Systems

A real-time system is often defined as a system whose correctness depends not only on the logical results, but also on the time at which the results are produced [33]. Such timing requirements are often expressed as *deadlines* for computations. With this terminology, a real-time system is one that must compute results within deadlines. Missing a deadline is considered to be a failure.

Note that 'real-time' is not the same as 'fast'. Rather, we want a real-time system to be *predictable*, that is, we want to be able to guarantee (to some degree) that the system will fulfill its timing requirements. A cache memory, for example, improves average-case performance but is challenging to predict. (The classic fable of the hare and the tortoise also illustrates this point.)

This is not to say that performance is not important for this class of systems; performance and predictability are just distinct properties.

1.1 Hard and Soft Real-Time Systems

It is common to classify real-time systems as *hard* or *soft*. In a hard real-time system, missing a deadline is a complete failure — a late result is virtually useless. Worse yet, in a safety-critical system, such a failure may put humans in physical danger. Embedded real-time systems are often classified as hard.

Soft real-time systems, on the other hand, can be allowed to miss a deadline as long as it does not happen too often. Consider a telephone switch. Specifications state that upon lifting the receiver from the hook,

a dialing tone should be given within a certain amount of time (say, 0.4 seconds). However, a small additional delay (say, another 0.2 seconds) is not really a failure. The caller may become slightly puzzled, but can still make the call.

In practice, many real-time systems are neither completely hard nor completely soft. Rather, these two models represent the extremes of a continuum. There is a tradeoff between temporal correctness (i.e., keeping deadlines) and utilization of computing power.

2 Scheduling of Real-Time Tasks

The software in a real-time system is typically decomposed into a set of concurrent tasks according to some design strategy (e.g., [7, 30]). The tasks are often assumed to be periodic. (Even if they are not, they presumably have maximal frequencies at which they must run — hence, the worst-case behavior can be expressed as periodic.) To ensure that the system fulfills its timing requirements, the execution of these tasks must be properly scheduled.

Over the last decades, a variety of real-time scheduling techniques have been developed. The most straight-forward approach is *static scheduling*, that is, to assign execution time slots for all tasks off-line, once and for all. This approach is easy to implement, behaves predictably, and has low run-time overhead. However, in many systems tasks are created and destroyed at run-time (in response to external events). Such dynamic behavior is difficult to handle using static scheduling.

In *dynamic scheduling* techniques the real-time kernel decides which task to execute at any given instant. Since these decisions are not made in advance, changes to the task set at run-time can be accommodated relatively easily. A number of dynamic scheduling techniques have been developed, and many of them have come to industrial use. They generally have at least the following elements:

A scheduling algorithm for use by the real-time kernel.

A schedulability test, that is, a method to determine whether a given task set can be safely scheduled using the algorithm. This test can be used both by a system designer (for the scheduling of the initial task set of the system) and by the real-time kernel (to decide whether a new task can be accepted at run-time).

It is also of interest to be able to compute timing properties such as maximal response times for tasks. Such computations are based on a number of parameters (in addition to the scheduling technique at hand),

such as properties of the real-time kernel and its synchronization primitives.

2.1 An Example: Rate Monotonic Scheduling

To exemplify dynamic scheduling, we briefly sketch the *rate monotonic scheduling* (RMS) technique [21]. It is a *fixed priority* scheduling technique, that is, once a task has entered the system its priority will not change over time. (The underlying assumption is that the timing properties of tasks do not change over time.) Of all tasks that are enabled for execution, the one with the highest priority is executed by the real-time kernel. The scheduling algorithm is thus established; the question of how to select task priorities remains.

The rate monotonic task scheduling assumes tasks to be periodic (as is often the case). It is based on two parameters for each task i: the (fixed) period T_i and the execution time C_i. The deadline is assumed to equal T_i, that is, each task execution is required to finish before the next execution (of the same task) is scheduled for execution.

Priorities are assigned to tasks according to their rate of execution; the shorter period a task has, the higher priority it gets. A rather simple condition then states whether a system with n tasks is schedulable (that is, whether all task executions will complete within their deadlines):

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

If the inequality holds, then the system is schedulable.

3 Predicting Execution Times

Any real-time scheduling technique, such as the rate monotonic scheduling outlined in Section 2.1, assumes the task execution times to be known. However, the execution time of a task typically varies from one execution to another. For example, the number of iterations in a loop may vary, and a memory access may or may not hit the cache memory.

To ensure that the scheduling can accommodate *any* possible execution of each task, we need to base the scheduling on the *worst-case execution time* (WCET) of tasks. Given that we typically do not know the actual WCET in advance, we must somehow predict it.

Prediction of WCETs is an area of extensive research. A prediction of the WCET will, in general, differ from the actual WCET. In a hard real-time system, the error must be on the conservative side, that is, the WCET prediction must be equal to or longer than the actual WCET. In

some softer real-time systems, on the other hand, it may be preferable to base the scheduling on a shorter execution time than the WCET. In systems where the average- and worst-case execution times differ significantly, such an approach can increase utilization of the CPU at the cost of a few missed deadlines. Real-time applications where this trade-off is explicit, and even negotiated at run-time, include *Quality of Service* (QoS) and feedback scheduling systems [8].

To predict the WCET of a task, two general approaches are possible: to measure and analyze actual execution times for a number of executions, or to statically analyze the code and automatically predict the WCET. We will now review these two approaches in more detail.

3.1 Dynamic Measurements

We need to know how long time a task will take to execute. An obvious approach, then, is to execute it and measure its execution time. To allow for variations in execution time, we could perform these measurements on a number of executions and make sure to cover as many different executions as possible. We could then add some safety margin to the measured execution time, and use the result as a WCET prediction.

The primary drawback of this approach is that it is inherently optimistic. There is, in general, no way to guarantee that the longest time we observe is indeed the longest execution time that can ever occur. The execution typically depends on some input data, and in an embedded system interacting with its environment (e.g., a control application), it may be impossible to come up with the worst-case input data. Even if a safety margin is added to the time, it is impossible to tell whether that margin is sufficient to cover all future executions (unless a very conservative margin is chosen).

With this drawback in mind, dynamic measurements can indeed be useful, at least for softer real-time systems. In addition to WCET predictions (albeit optimistic ones), dynamic measurements can provide information about the *distribution* of execution times [31]. They can also show whether the worst-case execution times are substantially longer than the average-case ones, and in which contexts the worst-case execution times occur [28]. Such information can be useful when a task scheduling strategy is chosen.

Another dynamic approach is *evolutionary testing* [24], where the input to the system is automatically and iteratively adjusted to seek the worst-case behavior. However, there is still no guarantee that the longest observed execution time is indeed the longest possible.

3.2 Static Worst-Case Execution Time Analysis

A complementary approach to dynamic measurements is static analysis. The purpose of a *WCET analysis* is to automatically compute a WCET prediction for a given program. The computed prediction should be conservative, that is, the predicted WCET should be at least as long as the actual WCET. It should, however, be as close to the actual WCET as possible to ensure proper utilization of the processor.

The most primitive approach is to manually examine the machine code of interest and compute the WCET as the sum of the individual instructions' execution times. This is, however, only feasible for very small and simple (i.e., trivial control flow) assembler programs running on deterministic processors. High-level languages, complex control flow, and high-performance hardware require special treatment.

As suggested by Shaw in [32], there are two fundamental aspects of WCET analysis:

Source level analysis (or *structure level analysis*) concerns determining the longest possible execution path in a program, and the execution time for that path.

Hardware level analysis concerns determining the WCET of segments of object code with respect to the processor and the memory hierarchy (data and instruction cache memories).

Although these two aspects are largely separate, they do interact in some important ways. The time to execute the longest possible path depends on the processor; conversely, the timing properties of cache memories depend on the past history, such as the path of execution. Hence, both aspects should be considered.

3.3 Source Level Analysis

From the perspective of the source code, determining the worst-case execution time is a matter of control flow. The execution path that takes the longest time to execute, with respect to loops and conditional statements, must be identified.

The problem of bounding the number of loop iterations is thus important in WCET analysis. The general problem of bounding loops, however, constitutes a special case of the halting problem (which is known to be undecidable).

On the other hand, infinite loops are rarely of interest in real-time systems. Although the cyclic nature of real-time tasks is typically expressed

```
/**
 * Compute integer approximation to square root of x
 * where x is known to be in the range [0..1000].
 */
int sqrt(int x) {
    int a = 0;
    if (x < 0 || x > 1000) abort("sqrt: x out of range");
    while ((a + 1) * (a + 1) <= x)  /*$ loop-bound 32 */
        a++;
    return a;
}
```

Figure 1 A loop (in C/Java syntax) whose loop bound is difficult for a tool to deduce automatically, but straightforward for a programmer to provide. The loop will terminate after at most $\lceil \sqrt{1000} \rceil = 32$ iterations. The syntax of the annotation permits the timing information to be utilized by a WCET analysis tool, yet ignored by a traditional compiler. (The example is taken, with cosmetic changes, from [32].)

as an infinite loop, it is not the execution time of that loop we are interested in. Rather, it is the *body* of the loop that operates under real-time requirements.

Programmers write loops that they expect to terminate within some number of iterations, the *loop bound*. That bound may be more or less obvious from the code, but the programmer should always be aware of it. If no loop bound exists, the WCET is unbounded and the system cannot possibly be scheduled safely.

Given that a loop bound exists, we could perform some sort of automatic analysis to determine it. Such an analysis would certainly fail for some complicated loops. (This is inevitable, given the relation to the halting problem.) However, it may provide useful information in many common situations, such as *for* loops with simple integer induction variables.

In other cases, where the loop bound is known to the programmer but less obvious in the code, *annotations* (or *assertions*) can be used to provide these bounds to a WCET analysis tool. Such situations include, for example, loops traversing data structures. An example of an annotated loop is given in Figure 1. As the example indicates, these annotations are typically expressed as source code comments adhering to some defined syntax. They can thus be identified and parsed by a WCET analysis tool, but are ignored by a traditional compiler.

Timing Schema

An early conceptual framework for WCET analysis, the *timing schema*, was established by Shaw in [32]. Park and Shaw also later developed a tool based on this concept [26]. The straightforward idea of the timing schema is to describe timing properties of programs in terms of source code constructs. For example, the worst-case execution time of the *while* statement in Figure 1 is modeled as

$$T_{\texttt{while}} = (N_{bound} + 1) \times T_{cond} + N_{bound} \times (T_{\texttt{a++}} + T_{loop}) + T_{exit}$$

where N_{bound} denotes the loop bound (32), T_{cond} denotes the WCET of the loop condition evaluation ($(a + 1) * (a + 1) <= x$), $T_{\texttt{a++}}$ denotes the WCET of the statement inside the loop, T_{loop} denotes the overhead of the loop (e.g., branching to the loop header), and T_{exit} denotes the time to leave the loop. Of course, T_{cond} can in turn be expressed in terms of the times for multiplication, addition, and comparison.

The timing schema approach thus models execution times in a divide-and-conquer, syntax-oriented fashion. These times ultimately depend on times for some *atomic blocks*, such as individual multiplications or the loop overhead ascribed T_{loop} above. The exact boundary of an atomic block (that is, the interface to the hardware level analysis) is chosen by the tool designer.

The Balance between Annotations and Analysis

There is a balance to keep between programmer annotations and automatic analysis. On one hand, some properties of the developed program cannot be deduced by a tool. The halting problem says, simply put, that it is impossible (in the general case) for an automatic analysis to determine whether a given program will terminate. This implies that we cannot expect a WCET analysis tool to handle arbitrarily complex loops.

When restricted approaches to analyzing loops (such as identifying and analyzing induction variables) are not enough, manual annotations are needed. Such annotations represent the programmer's interpretation of the timing requirements, and it is thus quite reasonable for them to be explicit in the code. In addition, measurements on existing embedded code [9] indicate that rather simple control flow dominates, requiring few and straight-forward annotations.

On the other hand, programmers can certainly make mistakes, and automatic analysis can sometimes be used to replace or verify annotations. Several researchers have devised techniques for computing loop bounds and other related properties. Such automatic analyses can, in

```
if (x < 5) a = 3;
if (x > 7) a = 9;
```

Figure 2 A program with a false path. The conditions in the *if* statements cannot possibly both be true in a single execution (assuming x cannot be modified by another thread of execution). Thus, the execution path including both assignments to a is a false one.

some cases, provide loop bounds that are hard to describe using programmer annotations. Such a case is nested loops where the number of iterations in the inner loop depends on the value of the index in the outer loop. (The bubblesort algorithm, for example, is often implemented this way.)

The loop bound prediction technique devised by Healy et al. [12] is based on classic control-flow analysis. Using information about block dominance and loop frontiers, their analysis can give quite tight predictions of the number of iterations in loops with integer indices. In particular, their technique gives tight results for nested loops where the number of iterations in an inner loop depends on the index of an outer one.

Gustafsson [11] used abstract interpretation to automatically determine loop bounds and *false paths*, that is, paths of execution that cannot possibly be executed due to semantic dependencies. A simple example of such a false path can be seen in Figure 2. Without consideration to false paths, the WCET of this code excerpt would include both assignments and thus be overly conservative.

Park [27] and Bernat [4] used explicit annotations to identify false paths. Lundqvist [22] used instruction-level symbolic execution for path analysis. Liu and Gomez [18] used similar techniques (on the source code level) to determine loop bounds, recursion depths, and false paths. Altenbernd [2] used a related approach to exclude false paths.

Several of the WCET analysis techniques just mentioned can be categorized as symbolic execution (or abstract interpretation) of the program. In this context, such symbolic execution differs from ordinary execution in that variables may assume the value *unknown* in addition to regular values. (Input to the program, for example, can be assumed to have that initial value.) Computations involving *unknown* values give the result *unknown*.

For example, the WCET of an *if* statement

```
if (x) s1; else s2;
```

can then be modeled (in a timing schema fashion) as

$$T_{\texttt{if}} = T_{\texttt{x}} + T_{branch} + \begin{cases} T_{\texttt{s1}} & x = \textit{true} \\ T_{\texttt{s2}} & x = \textit{false} \\ \max\left(T_{\texttt{s1}}, T_{\texttt{s2}}\right) & x = \textit{unknown} \end{cases}$$

where T_{branch} denotes the overhead of any necessary branching in the *if* statement.

3.4 Hardware Level Analysis

The rapid development in computer architecture provides a plethora of research opportunities in WCET analysis. A fundamental principle in modern computer design is to "make the common case fast" [16]. This principle focuses on *average-case* performance, rather than the *worst-case* performance considered in real-time systems. Pipelines and cache memories, both common in modern high-end processors, are average-case performance enhancements with insidious worst-case performance.

It should be noted that such hardware facilities are quite rare in embedded systems, which are typically based on quite simple 8-bit or 16-bit dedicated microcontrollers [9]. However, as caches and pipelines possibly become more common in the future, a number of analysis techniques have been developed to predict their behavior. These analyses typically operate on an object code representation of the program. This approach results in some restriction on the code that can be analyzed; in particular, many approaches below cannot analyze recursive calls and jumps to addresses given by register contents. As we will discuss in Section 3.5, such indirect jumps are common in code generated for object-oriented programs.

Pipeline Prediction

Pipelined processors exhibit *instruction-level parallelism*: several instructions are processed simultaneously in different stages of execution, analogous to an assembly line. The pipeline *stalls* (that is, the processor is halted for a clock cycle) if two or more instructions in the pipeline interfere in some way (for example, if one instruction depends on a result currently being computed in another stage of the pipeline). During this stall cycle, the instruction producing the result continues execution, while the one anticipating the result is suspended. (Stalls may also occur for other similar reasons.)

To determine the worst-case behavior of a pipelined processor, it is necessary to predict for which instructions a stall can possibly occur. This depends not only on the instruction itself, but also on other preceding instructions.

Healy et al. [13] predict the worst-case effects of pipelining by associating pipeline information with each instruction. That pipeline information is a model of the instruction's worst-case usage of the pipeline stages. The pipeline effect of a sequence of instructions can then be computed by concatenating the pipeline information for individual instructions.

The *extended timing schema* devised by Lim et al. [20] associates a *worst-case timing abstraction* (worst-case pipeline information) with each program construct (rather than an execution time, as in the original timing schema). Timing abstractions are recursively combined, in a fashion analogous to the original timing schema, to produce timing abstractions for larger parts of programs.

Lundqvist's approach [22] also takes pipeline effects into account, as does the integer-linear programming (ILP) approach taken by Li, Malik, and Wolfe [19].

Cache Memory Prediction

In modern high-end computers, there is a discrepancy between the cycle time of the processor and the latency of the main memory. The processor needs instructions and data at much higher rate (often one or two orders of magnitude, or more) than the memory can provide. To support a high processor clock frequency, a small but fast *cache memory* is used to hold instructions and data expected to be needed in the near future. In most cases a required value can be found in the cache; hit rates of 98% are common. If the value is not found, a *cache miss* occurs and the value must be fetched from main memory. Such misses are usually infrequent, but costly in terms of execution time.

The cache memory is typically divided into an instruction cache and a data cache, since these kinds of memory accesses follow different patterns. Whereas instruction fetching follows a fairly regular pattern, data memory accesses are more difficult to model. A given load instruction, for example, always appears at the same position in memory, but the memory cell it accesses may vary from one execution to another (depending on register values).

Predicting the worst-case timing behavior of cache memories is thus a question of predicting when cache misses can possibly occur. As one might suspect from the preceding discussion, data caches are more complex to predict well than instruction caches are. The analysis techniques we review in this Section typically treat these two cache types differently. Since less information is available (at analysis time) about data memory references than instruction fetching, the prediction of data caches is usually less accurate than that of instruction caches.

Cache prediction requires a rather detailed model of the actual cache

design at hand. Such a model includes information about whether the cache is direct-mapped or set-associative [16]. For set-associative caches, the policy for block replacement must also be known.

The extended timing schema of Lim et al. [20] handles both instruction and data caches. They also discuss possible hardware support to increase the accuracy of data cache prediction. Alt et al. [1] use abstract interpretation to predict the behavior of instruction and data caches. The ILP approach of Li, Malik, and Wolfe [19] includes prediction of a direct-mapped instruction cache. Lundqvist's symbolic execution [22] treats set-associative instruction and data caches.

Healy et al. [13] categorized instruction cache accesses as *always miss*, *always hit*, *first miss*, and *first hit* (conflict) using data-flow analysis. White et al. [35] extended this approach to handle set-associative data caches, and Mueller [23] extended it to multi-level caches.

Other Issues

In addition to the pipeline and cache issues previously mentioned, a number of other considerations exist in hardware level WCET analysis. Context switches and interrupts, for example, invalidate any assumptions about cache contents in the cache analyses above and thus require special treatment. Also, direct-memory access (DMA) interfaced hardware affects the WCET of executing programs [17, 26].

3.5 Considerations in Object-Oriented Languages

As we will motivate further in Section 4, we are interested in using object-oriented languages for the development of real-time embedded software. However, the source language WCET issues discussed in Section 3.3 are generally aimed at procedural languages such as C. Although most object-oriented languages share much of their imperative semantics with procedural languages, there are some additional issues that need to be addressed.

One such issue is calls to virtual methods (dynamic binding). The method implementation actually called is determined at run-time (based on type information) and is thus, in general, not statically known. To predict the WCET of such a call, a conservative estimation of the set of possibly called implementations can be made. Related analyses have been made to facilitate optimizations in compilers for object-oriented languages [5, 34].

Note that the problem is not solved by resorting to simpler, non-object-oriented languages such as C. A virtual method in Java essentially corresponds to a function pointer in C. Such function pointers are considerably

more difficult to handle for a WCET analysis. Thus, an implementation of a given design may well be more difficult to analyze if it is expressed in C than in Java.

The Java language supports dynamic loading, which means that such a conservative estimation may be impossible. Any assumptions about possibly called methods need to be reevaluated when new code is loaded. As we will discuss further in Paper III, annotations (similar to those presented in Section 3.3) can be used in this situation.

Analysis of virtual methods generally requires source code information. Most approaches based solely on object code analysis fail when a jump instruction whose target address is given by a register is encountered. It is common to implement virtual method calls with such instructions.

Another important issue is the garbage collection employed by most object-oriented languages. If the garbage collector is not designed for real-time use, unpredictable delays will make a WCET analysis impossible. A number of real-time garbage collection schemes have been developed using approaches such as scheduling [15, 29] and hardware support [25]. However, these schemes require information about the amount of memory the garbage collector should manage. We will treat this issue in Paper II.

There is little previous work on WCET analysis for object-oriented languages. Bernat [4] devised an approach to predict the WCET of Java bytecode, but did not address the issues of virtual methods and garbage collection. Gustafsson [11] used type inference to make WCET predictions in a special real-time dialect of Smalltalk. However, it is unclear how the garbage collector in that system deals with real-time requirements.

3.6 Analysis Time and Precision

Ideally, a WCET analysis tool should provide exact predictions, be able to analyze large and small programs with equal ease, and compute its results in diminutive time. However, these requirements are contradictory and trade-offs have to be made in a practical tool design.

In WCET analysis approaches based on abstract interpretation, symbolic execution, and related techniques, an analysis is itself an execution of the analyzed program (only slower). Although such an analysis can provide much information, the analysis time may be substantial, making the techniques less suitable for interactive environments.

The halting problem implies that we cannot, in general, even guarantee such an analysis to terminate. In particular, for some loops and recursive calls depending on *unknown* values, the analysis will not terminate, even if the analyzed program will. Gustafsson [11] evaded this problem by allocating *time budgets* to tasks.

Another issue is the inherently exponential growth of the number of possible execution paths in a program. Consider, for example, a program section with N consecutive *if* statements. The number of possible execution paths in the program is then 2^N. For an analysis that operates on the object code, such alternative paths occur whenever a conditional branch appears in the code. For large programs it is infeasible to analyze all these paths independently; hence, a trade-off must be made for the analysis to be scalable.

Various approaches are possible to reduce this exponential growth. Al-Yaqoubi [3] partitioned control flow; while analysis within such a (preferably small) partition still has exponential complexity, the results for partitions can be combined in linear time. In Lundqvist's approach [22], information about two execution paths is merged whenever they meet.

These techniques reduce analysis time and complexity at the expense of precision. Other approaches, such as the timing schema and its variants, avoid explicit path enumeration and thus make similar trade-offs implicitly.

4 Approach and Contributions

Object-oriented programming dominates the software development of today. Interestingly, although object-oriented languages have been used in various contexts since the 1960's, their fundamentals have not changed significantly in that time. The semantics of Java [10] are very similar to those of Simula 67 [6]. Perhaps object-orientation simply suits the way we think — it does have some striking resemblances to Plato's theory of Forms[1], an ancient model of the world.

Object-oriented languages are becoming more and more used in real-time software development, but simpler languages such as C or assembly code are quite common. One reason for this is that the timing properties of object-oriented programs are relatively complex to predict, as the discussion in Section 3.5 suggests. Nevertheless, as increasingly complex embedded systems become ubiquitous in our society, it is desirable to use modern languages to develop their software. Object-orientation is indeed already popular for analysis and design of embedded real-time systems [7]; now we need the language support for their implementation.

4.1 Java in Embedded Real-Time Systems

We base our approach on (a subset of) the Java language [10], which has received much attention in the community of embedded real-time sys-

[1] See, for example, the *Republic*.

tems. Java is compiled to *bytecode*, a stack-based code model for a Java virtual machine (JVM). The JVM is almost always a piece of software which can be implemented on more or less any hardware platform.

Java's bytecode approach was originally taken to support code mobility over the Internet (applets), but has some interesting applications for embedded systems. Java is designed to support dynamic loading of code, that is, to load code while the system is running. Such dynamic loading is desirable in some embedded systems, such as industrial robots, telephone switches, and satellites, which often cannot be taken off-line without significant economic penalties.

The Java language rules enforce safety. The language is designed to use compile-time checks in as many cases as possible, and to use run-time checks in the remaining ones. Such checks prevent, for example, stray pointers from destroying data structures and code in other parts of the system (thus leading to strange crashes). Violation of Java's semantics results in a defined behavior (a run-time exception with an error message at the point of error), rather than the unpredictable behavior of unsafe languages (a bus error, segmentation fault, or erroneous behavior at some later, unknown time). Also, the use of garbage collection relieves the programmer of the notoriously error-prone labor of manual memory management, reducing the risk for memory leaks and fragmentation.

This safety is especially important in embedded systems, which must operate autonomously for long periods of time. Unlike some personal computers, they cannot simply be restarted due to crashes a few times a day. The static typing and run-time checks allow faults to be identified early during development.

We use Java bytecode not only as the interface between the compiler and the virtual machine, but also as the the interface between the source level and the hardware level of WCET analysis. This interface thus represents the hardware level analysis aspect of our tradeoff between analysis complexity and precision, as discussed in Section 3.6. The path (source level) analysis approach is that of the timing schema.

Our work is aimed towards relatively predictable processors, since these are by far the most common ones in embedded systems. However, the bytecode approach makes a more aggressive hardware level analysis within individual bytecode instructions possible.

4.2 Enclosed Papers

In this thesis, we present the foundations of an environment for development of object-oriented real-time software. The rest of the thesis is organized as follows:

Paper I shows how to use the timing schema as an actual *implementation* of the WCET analysis, rather than a *specification* as originally intended. This is done in a declarative manner using an extended attribute grammar formalism (reference attributed grammars, or RAGs [14]) which facilitates a clear, concise, and modular implementation.

The use of RAGs also facilitates an interactive analysis, which is further considered in Paper III.

Paper II addresses real-time garbage collection. Existing schemes for real-time garbage collection require certain information about the program, in particular, an upper bound on the amount of *live memory* used by the program. That information can be used to schedule real-time garbage collection safely and is thus analogous to a WCET prediction. In the paper we show how a *live memory analysis* can be used to compute a live memory bound in an object-oriented program. Our approach uses straight-forward annotations regarding size and shape of data structures. The analysis takes object-oriented language features, such as inheritance and virtual methods, into account.

Paper III treats our experimental tool *Skånerost*. The tool integrates syntax-oriented editing with Java bytecode compilation, WCET analysis, and live memory analysis. WCET prediction of virtual method calls in the presence of dynamic loading of code is discussed. Special consideration is given to *interactivity*, that is, allowing the programmer to obtain timing and memory predictions at any time throughout development.

Whereas the first two papers present our analysis techniques, the third paper shows how these techniques fit into the development tools and the associated work process.

4.3 Key Contributions

In this thesis, we show techniques for predicting execution time and memory bounds of real-time software. Special attention is paid to techniques for object-oriented languages with garbage collection. Our work is a step towards making object-oriented languages predictable enough for hard real-time systems.

The declarative implementation techniques used facilitate not only a clear and concise implementation, but also interactivity in the resulting tools. This interactivity is desirable in real-time software development, where design changes due to timing problems become costly if these problems are not detected early.

4.4 Directions for the Future

Several important issues in our work remain. As mentioned in Section 3.5, virtual methods and garbage collection require special treatment in real-time systems. Further research is needed to understand how analysis tools, the run-time system, and the programmer should interact.

Practical experiments to validate our approach will be made. Planned such experiments target a small embedded microprocessor running the Infinitesimal Virtual Machine (IVM), a very compact real-time Java virtual machine currently in development at Lund University. However, much of our work is independent of the bytecode approach and can also be used in conjunction with compilation of Java to native (machine) code.

While static WCET predictions are useful for hard real-time systems, dynamic measurements (Section 3.1) can be useful for scheduling of softer real-time systems. Applications include, e.g., feedback scheduling [8]. Feedback from such measurements in the development environment is an interesting extension to the present work.

An object-oriented design can convey important timing information. For example, it can be used to classify tasks as periodic or non-periodic, and express which parts of the code operate under real-time requirements. The relation between object-oriented frameworks, annotations, and analysis needs to be further investigated.

The Skånerost environment is currently under implementation; that work continues.

References

[1] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache Behavior Prediction by Abstract Interpretation. *Proceedings of the International Static Analysis Symposium (SAS'96)*, LNCS 1145, Springer, 1996.

[2] P. Altenbernd. On the False Path Problem in Hard Real-Time Programs. *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, 1996.

[3] N. M. Al-Yaqoubi. *Reducing Timing Analysis Complexity by Partitioning Control Flow.* Master's Thesis, Department of Computer Science, Florida State University, 1997.

[4] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. To appear in *Proceedings of the 12th EuroMicro Conference on Real-Time Systems*, Stockholm, June 2000.

[5] C. Chambers and D. Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. *Proceedings of the ACM SIGPLAN Conference on Programming Language and Design (PLDI'90)*, White Plains, New York, June 1990.

[6] O.-J. Dahl, B. Myhrhaug and K. Nygaard. SIMULA-67 Common Base Language. Technical Report S-22, Norwegian Computer Centre, Oslo, Norway, 1970.

[7] B. P. Douglass. *Doing Hard Time: Developing Real-Time Systems With UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999.

[8] J. Eker, P. Hagander, and K.-E. Årzén. A Feedback Scheduler for Real-time Controller Tasks. Accepted for publication in *Control Engineering Practice*, 2000.

[9] J. Engblom. Why SpecInt95 Should Not Be Used to Benchmark Embedded Systems Tools. *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*, Atlanta, Georgia, May 1999.

[10] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[11] J. Gustafsson. *Analyzing Execution Time of Object-Oriented Programs with Abstract Interpretation.* PhD Thesis, Department of Computer Systems, Information Technology, Uppsala University, Sweden, May 2000.

[12] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding Loop Iterations for Timing Analysis. *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, Denver, Colorado, June 1998.

[13] C. A. Healy, R. D. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, Vol. 48, No. 1, January 1999.

[14] G. Hedin. Reference Attributed Grammars. *Proceedings of WAGA '99, Second Workshop on Attribute Grammars and their Applications*, Amsterdam, The Netherlands, March 1999.

[15] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems.* PhD Thesis, Department of Computer Science, Lund University, September 1998.

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture — A Quantitative Approach (Second Edition)*. Morgan Kaufmann Publishers, 1996.

[17] T.-Y. Huang and J. W.-S. Liu. Predicting the Worst-Case Execution Time of the Concurrent Execution of Instructions and Cycle-Stealing DMA I/O Operations. *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS'95)*, La Jolla, California, June 1995.

[18] Y. A. Liu and G. Gomez. Automatic Accurate Time-Bound Analysis for High-Level Languages. *Proceedings of the ACM SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, Montreal, Canada, June 1998.

[19] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95)*, Pisa, Italy, December 1995.

[20] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C. S. Kim. An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, Vol. 21, No. 7, 1995.

[21] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, Vol. 20, No. 1, 1973.

[22] T. Lundqvist. *A Static Timing Analysis Method for Programs on High-Performance Processors*. Licentiate Thesis, Department of Computer Engineering, Chalmers University of Technology, Sweden, 1999.

[23] F. Mueller. Timing Predictions for Multi-Level Caches. *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS'97)*, Las Vegas, Nevada, June 1997.

[24] F. Mueller and J. Wegener. A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints. *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, Denver, Colorado, June 1998.

[25] K. Nilsen. Reliable Real-Time Garbage Collection of C++. *Computing Systems*, 1994.

[26] C. Y. Park and A. C. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *IEEE Computer*, Vol. 24, No. 5, 1991.

[27] C. Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, Vol. 5, 1993.

[28] P. Persson, A. Cervin, and J. Eker. Execution Time Properties of a Hybrid Controller. Technical Report, Department of Automatic Control, Lund Institute of Technology, 2000.

[29] T. Ritzau. *Real-Time Reference Counting in RT-Java.* Licentiate Thesis, Department of Computer and Information Science, Linköping University, Sweden, 1999.

[30] B. Sandén. Entity-Life Modeling and Structured Analysis in Real-Time Software Design — A Comparison. *Communications of the ACM*, Vol. 32, No. 12, 1989.

[31] V. Sarkar. Determining Average Program Execution Times and their Variance. *Proceedings of the ACM SIGPLAN Conference on Programming Language and Design (PLDI'89)*, Portland, Oregon, June 1989.

[32] A. C. Shaw. Reasoning about Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, Vol. 15, No. 7, 1989.

[33] J. A. Stankovic et al. Strategic Directions in Real-Time and Embedded Systems. *ACM Computing Surveys*, Vol. 28, No. 4, 1996.

[34] V. Sundaresan, C. Razafimahefa, R. Vallée-Rai, and L. Hendren. Practical Virtual Method Call Resolution for Java. Sable Technical Report No. 1998-7, McGill University, Canada, 1998.

[35] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing Analysis for Data Caches and Set-Associative Caches. *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, Montreal, Canada, June 1997.

Paper I

Interactive Execution Time Predictions Using Reference Attributed Grammars

Patrik Persson and Görel Hedin

Abstract

A central problem for real-time scheduling is to acquire tight but conservative upper bounds on task execution times. We present a prototype for an environment where such bounds are interactively presented in terms of source code constructs to the programmer during development. The prototype is based on the language development tool APPLAB and uses an extended attribute grammar formalism, reference attributed grammars (RAGs), which overcomes some drawbacks of conventional attribute grammars in this context (e.g., description of non-local dependencies). In this paper we show how timing schemata can be implemented as RAGs. Our experience is that the RAG approach allows timing schemata to be implemented in a clear, concise, and modular manner.

1 Introduction

Real-time scheduling algorithms, such as [9], generally assume the worst-case execution time (WCET) of tasks to be known. However, few tools to predict the WCET are available. Without prediction tools, the only way to estimate the WCET is to perform actual measurements of execution times with what is believed to be the worst-case input data. Such measurements are hard to make and generally not reliable — there is typically no way of telling whether the longest observed time is in fact the longest time possible.

Our research is concerned with techniques and tools for predicting the WCET of real-time programs. We base our WCET analysis on a conceptual framework known as timing schema [12]. Since the timing schema is a set of equations for the execution times of various language constructs, it would appear suitable to implement using attribute grammars (AGs). As we will demonstrate, however, conventional AGs have a few drawbacks in this context, and we instead use an extended variant known as reference attributed grammars (RAGs). RAGs allow non-local dependencies (for example, between procedure definition and procedure call sites) to be expressed concisely [5].

Our WCET analysis techniques are designed for integration with a compiler, allowing the analysis to be based on both source level information and generated code. We base our prototype tool on APPLAB [1], an environment for interactive development of domain-specific languages. The environment integrates structure-oriented editing with semantic analysis and code generation. Our tool provides WCET predictions of the program (both in parts and as a whole) continuously as the program is developed.

APPLAB uses a specification language based on the RAG formalism. The timing schema is described as a module of its own, textually separate from the remaining semantic analysis. Other modules include name analysis, type analysis, and code generation.

In this paper, we show how the timing schema formulation can be used for the actual implementation of a WCET analysis tool, rather than as a reasoning methodology as was originally intended. We show the drawbacks of using traditional AGs for this purpose, and how these drawbacks are overcome using RAGs. We also report from other experiences from implementing our WCET prediction tool in APPLAB.

1.1 Paper Outline

In Section 2, we motivate our requirements for a WCET prediction tool. We also present some related work, including the original timing schema

concept and its relation to attribute grammars. Section 3 is an overview of our prototype implementation using RAGs. Section 4 concludes the paper.

2 Timing Properties of Real-Time Systems

Stankovic [13] characterizes real-time systems as follows:

> *A real-time system is one in which the correctness of the system depends not only on the logical results, but also on the time at which the results are produced.*

In other words, a real-time system performs its computations within some timing constraints. This manifests itself in the scheduling of such a system, where the system's tasks are scheduled in time to meet the system's timing constraints (task deadlines). Depending on the nature of the system, a missed deadline may be more or less critical: in a multimedia application it may result in a transient distortion in a video stream, whereas in an engine control system it may result in an engine explosion.

2.1 Predicting the Worst-Case Execution Time

An important area of research is the prediction of the WCET of real-time tasks. A tool for this purpose should provide a prediction that is a tight upper bound of the actual WCET. That is, WCET estimations should be close to, but no lower than, the actual WCET.

The timing schema approach (presented in more detail in Section 2.2) was originally developed by Shaw [12] and was the basis for an experimental timing tool targeted at a subset of the C language [11]. However, in that work the timing schema concept was used as a "reasoning methodology for deterministic timing" rather than as an actual implementation.

As suggested in [11], the problem of bounding the WCET is twofold:

Hardware level analysis: determining the WCET (in cycles or nanoseconds) of a piece of object code on a particular hardware platform. Modern processors employ a variety of techniques to enhance performance, such as caching, pipelining, and speculative execution [6]. Although these techniques enhance average-case performance, they also make it considerably more difficult to predict the worst-case performance. Existing hardware level analysis techniques include low-level data-flow analysis [14] and the extended timing schema [8].

Structure level analysis: determining the execution time of the longest possible path in a program (or part of it) based on the execution

times of individual pieces of the program. This analysis may be performed at the object code level [10] or the source code level [3].

The research we describe in this paper is focused on structure level analysis. We do, however, intend to integrate some kind of hardware level analysis in the future.

A related and interesting problem is to provide the user with feedback about the program at hand. For this information to be useful and understandable, we believe it should be expressed in terms of the source code. Consequently, we work with source code level concepts; more specifically, abstract syntax trees (ASTs).

2.2 Timing Schemata and Attribute Grammars

In the timing schema approach, a WCET prediction with each "atomic block", where an atomic block is essentially any piece of sequential source code. (The original timing schema approach is concerned with source code rather than object code.)

The granularity of atomic blocks is a design parameter of the timing schema. In our present prototype, we have chosen to associate a constant WCET prediction with each terminal symbol. We concentrate on combining those predictions into composite predictions for non-terminals such as loops, procedures, and tasks.

Once timing predictions have been produced for the atomic blocks in a program, predictions can be calculated for composite constructions using their constituents. For example, the time of the assignment statement

```
a = b * c;
```

may be described as $T(b) + T(c) + T(*) + T(a) + T(=)$, where $T(X)$ denotes the worst-case execution (or evaluation) time of the node X in the abstract syntax tree. Similarly, the time of the *if* statement

```
if (exp) stmt1; else stmt2;
```

may be described as $T(\textit{exp}) + T(\textit{if}) + \max(T(\textit{stmt1}), T(\textit{stmt2}))$, where the function $\max(a, b)$ is defined to return the larger of the numbers a and b.

The timing schema formalism is clear and intuitive, and it seems an attractive option is to implement a schema using an attribute grammar. The execution time bounds of the AST nodes can be represented by synthesized attributes, and the timing schema by semantic rules. However, some important drawbacks arising in practice will be addressed in Section 3.1.

2.3 The Role of Timing Assertions

The timing constraints mentioned in the beginning of Section 2 are typically known at the time of system design. Software engineers developing a real-time system must be aware of these requirements throughout the development, including during the implementation phase. A *while* loop, for example, must never loop more than some fixed number of times. If no such bound exists, the WCET of the algorithm is unknown and the system cannot be safely scheduled. Although this information could in some special cases be extracted by an analysis tool, it is in fact an important part of the design.

We allow timing information to be explicitly expressed in the code using what we call *timing assertions*. These represent system design parameters and facilitate a more accurate analysis.

Assertions can also play a slightly different role. In some applications (in particular, multi-media applications), it is desirable to schedule some tasks within tighter bounds than the WCET (giving higher performance/throughput at the cost of sporadic missed deadlines). Timing assertions complemented with time-out exceptions allow real-time tasks to fall back on simpler algorithms when deadlines are missed.

A third use of timing assertions relates to object-oriented languages. The actual code to execute at a virtual method call is determined at runtime based on object type information. Except in some special cases, it is not possible to statically determine which code is actually called. In cases where a global analysis of all possible implementations of a virtual method is not possible, a timing assertion can be associated with the top-level declaration of a method. This way each implementation of the virtual method can be checked to ensure that its execution time does not exceed the execution time specified by the assertion.

2.4 Desired Tool Support

To conclude this section, we propose a WCET prediction tool with the following properties:

- *Interactive, source code oriented user feedback.* Tools and methods for non-real-time systems do not generally address the execution time; it is considered a low-level property of the finished implementation. For real-time systems, however, the timing properties (in particular, the execution time) are an inherent part of the system's behaviour. Execution time requirements are specified at design time. A tool should allow the user to monitor the execution time throughout development, at different granularity levels such as loops, procedures, and entire tasks.

- *Possible integration with low-level hardware specific WCET prediction techniques.* In Section 2.1, the WCET prediction was separated into two subproblems. While we focus on the structure level aspects of WCET analysis, it is desirable to provide for integration with hardware level prediction.
- *Support for assertions in the WCET analysis.* A system should allow the user to specify timing constraints (maximum bounds for the execution time/number of loop iterations) for loops, procedures, and tasks.

3 An Interactive WCET Prediction Tool Based on Reference Attributed Grammars

We have implemented a prototype for a WCET prediction tool. The prototype tool operates on a simple experiment language supporting some advanced object-oriented language concepts such as classes, inheritance, and qualified access. The language bears some similarities to Java [4], which we intend to support in the near future.

In the rest of this section, we present some of the considerations that have to be made when implementing timing schemata using attribute grammars. We also show how timing schemata may be implemented using RAGs.

3.1 Implementing Timing Schemata using Conventional Attribute Grammars

In Section 2.2, attribute grammars were suggested as an attractive technique for implementing timing schemata. Although viable for small examples in simple languages, this approach has a number of drawbacks when applied to a typical procedure- or object-oriented programming language. To see why, consider a function call:

```
f(a, b)
```

A reasonable approach to describing the execution time of the call might be $T(a)+T(b)+T(f_{call})+T(f_{body})$, where $T(a)$ and $T(b)$ refer to the time to evaluate the arguments, $T(f_{call})$ refers to the time for the function call itself, and $T(f_{body})$ refers to the execution time of the function body. The problem is the term $T(f_{body})$. While the other terms are directly available as attributes of either the call itself or its immediate children, the body of the called function is, in general, not automatically available at the call site. Instead it may be located arbitrarily far away in the AST.

One possible approach to making the timing information available is to integrate the timing analysis with the name analysis. The usual approach to name analysis is to make use of an inherited attribute *env* containing a mapping from visible identifier names to declaration information (such as variable types and function signatures). The declaration information in this mapping could be extended with an execution time value, i.e. a relatively modest modification of the specification. However, modularity is lost since WCET analysis is integrated with the conceptually quite unrelated name analysis.

To keep WCET analysis separate from name analysis, a table could be associated with each scope in the program. Such a table would hold the execution time of the body of every procedure declared in the scope. In a conventional procedural language, the execution times of function bodies would be available at the call sites by making the table an inherited attribute in the son nodes of the scope. In other words, this approach involves another name analysis.

Name analysis for object-oriented languages with inheritance and qualified access is more complex, since the dependencies between declarations and uses of identifiers do not follow the block structure of programs. Regardless of which of the two approaches above is chosen, this complexity is reflected in the WCET analysis. Either modularity is lost or a substantial amount of administration is required to describe what is essentially a simple relationship between a function call and the corresponding function body.

3.2 The RAG Implementation in the APPLAB System

We have implemented WCET analysis for a simple object-oriented language in our interactive language tool APPLAB (APPlication language LABoratory), an environment originally designed for interactive development of domain-specific languages [1, 2]. APPLAB is based on reference attributed grammars (RAGs) which is an extension to attribute grammars that allows attributes to be references to syntax nodes, supporting specification of non-local dependencies. For example, a use of an identifier may have a reference attribute denoting the corresponding declaration node.

The APPLAB system allows language specifications (RAGs) and programs to be edited simultaneously. A program is edited using a language based editor controlled by the language specification, allowing attributes of the syntax tree for the program to be displayed. We used this feature to interactively monitor the WCET for the edited program. Figure 1 shows a screendump from the system.

In our prototype, WCET analysis is implemented as a separate grammar module. That module is concerned solely with the WCET analysis

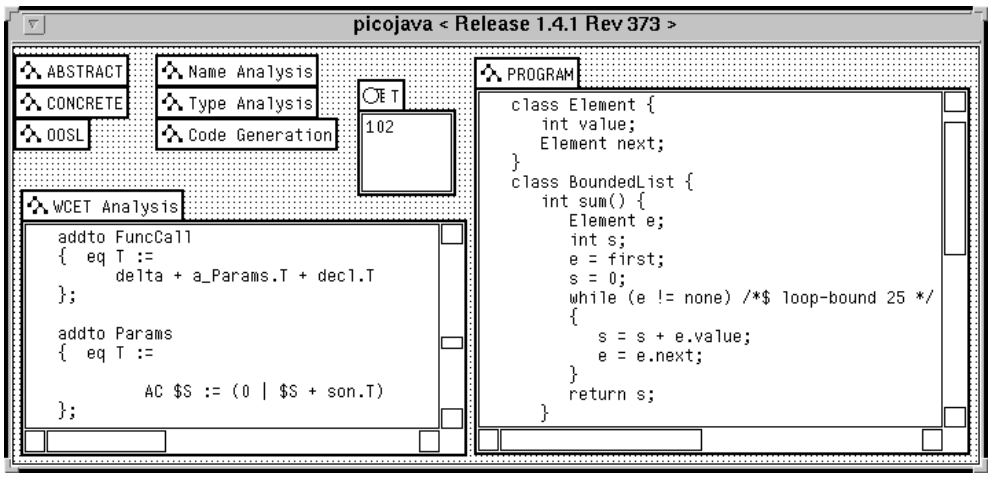


Figure 1 Example WCET analysis in APPLAB. The window labelled *T* shows the WCET prediction (the attribute *T*) of a portion of the program in the *PROGRAM* window.

(thus corresponding directly to a timing schema) but uses results (attributes) computed by the name analysis module, as will be shown below.

We focus on the structure level of WCET prediction as described in Section 2.1, that is, computation of WCET predictions for programs, loops, and methods based on given WCET predictions of individual atomic blocks. (In our case, these atomic blocks correspond to terminal symbols.) In the present prototype, we have assigned constant WCET predictions to all basic operations, such as assignment, arithmetic operations, and so on.

The remainder of this section is based on on the small object-oriented language whose context-free syntax is given in Table 1. The language supports variables, classes, non-virtual functions, qualified access, and statements such as assignments and *while* statements. The grammar notation in APPLAB makes use of an object-oriented extension of RAGs where the nonterminals may be organized in a class hierarchy, and where the productions are leaves in this hierarchy. In Table 1 a superclass is shown to the left of its subclasses (e. g. *Use* is a subclass of *Exp*). Attributes and semantic rules are inherited (in the object-oriented sense) along this hierarchy. For example, an attribute declared in the *Exp* nonterminal will be present in any node which is a subclass of *Exp*, e.g. *QualUse*, *SimpleUse*, or *FuncCall*.

The name analysis attribution module connects uses of identifiers to their declarations. The details of this attribution module are described in [5] and results in a synthesized attribute *decl* of the *Use* nonterminal denoting the corresponding *Decl* node. Table 2 shows this brief interface

Nonterminals			Productions	
Program				→ `program {` *Block* `}`
Block				→ *Decls Stmts*
Decls				→ *Decl**
Stmts				→ *Stmt**
DeclType			*IntDeclType:*	→ `int`
			BoolDeclType:	→ `boolean`
			RefDeclType:	→ *SimpleUse*
Decl			*ClassDecl:*	→ `class` *ID SuperOpt* `{` *Block* `}`
			VarDecl:	→ *DeclType ID*
			FuncDecl:	→ *DeclType ID* `(` *FormalParams* `)` `{` *Block* `}`
SuperOpt			*Super:*	→ `extends` *Use*
			NoSuper:	→
FormalParams				→ *VarDecl**
Stmt			*AssignStmt:*	→ *Use* `=` *Exp*
			WhileStmt:	→ `while (` *Exp* `)` *LoopBound Stmt*
			CompoundStmt:	→ `{` *Block* `}`
Exp	*Use*		*QualUse:*	→ *Use* `.` *UnQualUse*
		UnQualUse	*SimpleUse:*	→ *ID*
			FuncCall:	→ *ID* `(` *Params* `)`
LoopBound				→ `/*$ loop-bound` *INT* `*/`
Params				→ *Exp**

Table 1 Context-free grammar. (Some minor details, mainly syntactic sugar, have been omitted.)

to the name analysis module.

The WCET analysis module is shown in Tables 3, 4, and 5. In Table 3, a synthesized attribute T (modelling the WCET) is declared for the nonterminals *Stmt* and *Exp*, and semantic rules in *AssignStmt*, *SimpleUse*, and *QualUse* define values for this attribute. (We have denoted the WCET of the actual assignment by α, the WCET of a variable access by β, and the WCET of dereferencing a pointer by γ.) These equations could be expressed in most attribute grammar formalisms, since the *Use* and *Exp* instances associated with an *AssignStmt* instance are immediately available.

Function calls are described as a special use of identifiers, that is, we

Nonterminals	Attributes
Use	$\uparrow$ *decl* : **ref** *Decl*

Table 2 Interface to name analysis module.

Nonterminals	Attributes and Semantic Rules
Exp	$\uparrow T$: **integer**
Stmt	$\uparrow T$: **integer**
AssignStmt	$T := \alpha + Use.T + Exp.T$
SimpleUse	$T := \beta$
QualUse	$T := \gamma + Use.T + UnQualUse.T$

Table 3 Excerpt from the timing schema module.

Nonterminals	Attributes and Semantic Rules
Decl	$\uparrow T$: **integer**
FuncDecl	$T := Block.T$
FuncCall	$T := \delta + Params.T + decl.T$
Params	$\uparrow T$: **integer** $T := \sum_{e \in Exp^*} e.T$
Block	$\uparrow T$: **integer** $T := Stmts.T$
Stmts	$\uparrow T$: **integer** $T := \sum_{s \in Stmt^*} s.T$

Table 4 Timing schema for function declarations and calls.

Nonterminals	Semantic Rules
WhileStmt	$T := \epsilon + Exp.T +$ $LoopBound.INT.val * (\zeta + Exp.T + Stmt.T)$

Table 5 Timing schema for *while* statements with timing assertions. The expression *INT.val* denotes the numeric value of the integer constant *INT*.

have a production *FuncCall* which is a subclass of *Use* (refer to Table 3). Table 4 shows how function calls are described in our timing schema. The WCET of a function call is represented as a sum of the time for the actual call (represented by the constant δ), the time to evaluate the parameters, and the time to execute the function.

The WCET analysis uses non-local references to analyze function calls. The *decl* attribute defined in the name analysis module is a reference to the declaration of the called function, a non-local reference. The reference attribute allows the *T* attribute of the declaration (a *FuncDecl*) to be accessed directly in the *FuncCall* production, without the need for introducing auxiliary attributes on the path from the *FuncDecl* to the *FuncCall* in the syntax tree. The WCET analysis module is thus completely independent on the scope rules of the language: the rule for *FuncCall* is the same regardless of if the language is a procedural language with nested functions, or an object-oriented language (like here) where functions are inherited from superclasses to subclasses and can be accessed via a qualifying reference, or if any other scope rules are applied. However, to support also virtual functions (where the function body is not decided until runtime) the specification has to be extended.

A use of timing assertions is given in the *while* statement in Table 5. The statement has an associated loop bound (an integer constant) that states the maximal number of loop iterations. (*INT* is a predefined nonterminal modelling an integer constant.) The WCET of the *while* statement also includes the constants ϵ and ζ. The former accounts for any administration associated with the start and end of the *while* statement, and the latter for the overhead associated with each iteration.

A *while* statement in the language may, for example, look as follows (the compound statement is executed at most 10 times):

```
while (flag) /*$ loop-bound 10 */ {
    ...
}
```

The assertion represents the programmer's knowledge about the maximal number of iterations. (As discussed in Section 2.3, such bounds must exist and be known in hard real-time systems.) Compatibility with traditional compilers is ensured by expressing assertions in terms of special 'tagged' comments, similar in concept to the ones used by Javadoc [4]. This technique allows other compilers to parse the code without problems, while our tool can still predict the WCET.

3.3 Discussion

Implementation of timing schemata using RAGs in APPLAB suggests at least three major advantages:

- *RAGs provide reference attributes.* These permit timing schemata to be implemented in a clear and concise manner. While the traditional AG formalism resembles the timing schema concept, the non-local dependencies that frequently occur in WCET analysis are difficult to describe in traditional AGs. RAGs allow these non-local dependencies to be described in a concise way.
- *RAG modules provide modularity.* The timing schema of a language is described as a module of its own, separate from the name analysis module. The WCET module is concerned solely with the timing properties of the language at hand, and strongly resembles the timing schema. Although the timing schema is textually separate from other modules, it can exploit results from those modules. The timing schema module uses, for instance, the *decl* reference from a procedure call site to the corresponding procedure declaration. That reference is defined in the name analysis module. This makes the timing schema module independent of the scope rules of the language.
- *APPLAB provides interactivity.* For reasons given in Section 2.4, we want WCET predictions to be readily available throughout the development of real-time software. APPLAB allows an attribute, such as the WCET prediction of an arbitrary subtree of the AST, to be viewed at any time without noticeable delay.

4 Conclusions

We have presented an approach to predicting the worst-case execution time of real-time tasks. In our tool these execution time predictions are associated with source level constructs and continuously updated as the program is developed. We suggest this interactivity to be especially useful in the development of real-time software, where the timing properties must be kept in mind throughout the development process.

We base our approach on the timing schema concept [12], an intuitive way of expressing timing properties of programming languages. Although attribute grammars seem superficially suitable for implementing timing schemata, the frequently occurring non-local references are not easily described. The reference attributed grammar (RAG) formalism [5] overcomes this problem and allows timing schemata to be described as equations in a straightforward manner.

Rather than just using the timing schema as a specification, we use it for the actual implementation. The RAG based specification language of APPLAB allows attributes and equations for a particular purpose to be encapsulated into a separate module. In our prototype, one module is concerned solely with the timing schema and contains only the attributes and equations associated with that analysis. That module bears a striking resemblance to the timing schema formulation and holds all information that is required for the structure level WCET analysis.

Although our work is concerned with structure level analysis, we acknowledge the need to predict the low-level hardware timing behaviour. For example, the extended timing schema approach [8] (developed to predict timing behaviour of RISC processors utilizing caches and pipelining) seems suitable for integration with our approach.

4.1 Future Work

The initial prototype presented in this paper deals with a fictive programming language. We intend to extend our environment to handle the Java programming language [4] in the near future. This would require us to address some WCET prediction problems specific to object-oriented languages:

- *Predicting the WCET of virtual method invocations.* It is not possible (in general) to statically determine which code is executed upon an invocation of a virtual method. We plan to make use of timing assertions (as briefly mentioned in Section 2.3) to handle this problem.

- *Predicting the overhead of automatic memory management.* Java employs garbage collection, which may impose unpredictable interruptions of the execution unless special action is taken. The problem of predicting execution time overhead recently has been addressed by integrating garbage collection work with task scheduling [7]. However, some work remains, such as predicting the maximal allocation rate and the maximal amount of live memory possible in a task. We plan to investigate a technique similar to timing schema for this purpose.

Acknowledgments

We wish to thank Boris Magnusson and Klas Nilsson for numerous ideas and suggestions for our work, and Elizabeth Bjarnason for developing most of APPLAB. The anonymous reviewers and Margaret Newman provided several helpful comments.

The work of Patrik Persson is financed by ARTES (A network for Real-Time research and graduate Education in Sweden). The work of Görel Hedin is financed by NUTEK (the Swedish National Board for Industrial and Technical Development).

References

[1] E. Bjarnason. *Interactive Tool Support for Domain-Specific Languages.* Licentiate Thesis, Dept. of Computer Science, Lund University, December 1997.

[2] E. Bjarnason, G. Hedin, and K. Nilsson. Interactive Language Development for Embedded Systems. *Nordic Journal of Computing*, Vol. 6, pages 36-55, 1999.

[3] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculations of Execution Time. *Proceedings of EuroPar '97*, LNCS 1300, Springer, August 1997.

[4] J. Gosling, B. Joy, and G. Steele. *The Java Language Language Specification*. Addison-Wesley, 1996.

[5] G. Hedin. Reference Attributed Grammars. *Proceedings of WAGA'99: Second Workshop on Attribute Grammars and their Applications*, pages 153-172. Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (Second Edition)*. Morgan Kaufmann, 1996.

[7] R. Henriksson. *Scheduling Garbage Collection for Embedded Systems.* PhD Thesis, Dept. of Computer Science, Lund University, Sweden, September 1998.

[8] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An Accurate Worst Case Timing Analysis Technique for RISC Processors. *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'96)*, December 1996.

[9] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, Vol. 20, No. 1, 1973.

[10] T. Lundqvist and P. Stenström. Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques. Technical Report No. 93-3, Department of Computer Engineering, Chalmers University of Technology, Sweden, February 1998.

[11] C. Y. Park and A. C. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *IEEE Computer*, Vol. 24, No. 5, 1991.

[12] A. C. Shaw. Reasoning about Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, Vol. 15, No. 7, 1989.

[13] J. A. Stankovic et al. Strategic Directions in Real-Time and Embedded Systems. *ACM Computing Surveys*, Vol. 28, No. 4, 1996.

[14] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing Analysis for Data Caches and Set-Associative Caches. *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, June 1997.

Paper II

Live Memory Analysis for Garbage Collection in Embedded Systems

Patrik Persson

Abstract

Real-time garbage collection is essential if object-oriented languages (in particular, Java) are to become predictable enough for real-time embedded systems. Although techniques for hard real-time garbage collection exist, they are based on estimations of the maximum amount of referenced (live) memory. Such estimations may be difficult to derive manually for complex programs.

We present techniques for predicting the maximum amount of live memory in object-oriented languages with inheritance and virtual methods. Annotations are used to bound recursively defined data structures. The annotations may also be used for timing analysis of code traversing annotated structures.

A prototype live memory analysis tool has been developed. The tool interactively provides predictions of the maximum amount of live memory referenced from an arbitrary reference or block in an object-oriented program.

1 Introduction

Object-oriented programming (OOP) and Java are currently attracting substantial research interest in the embedded systems community. Although OOP is well established in other areas of software development, some obstacles remain for OOP to become predictable enough for hard real-time systems.

One such obstacle is the garbage collection (GC) employed in Java and other languages. The worst-case GC overhead in execution time must be bounded and known in order to guarantee that the system will fulfill its timing requirements. Still, the worst case should be as close to the average case as possible to allow good hardware utilization in a system scheduled using fixed-priority scheduling.

As shown by Henriksson [10], GC can be used in a hard real-time system by integrating GC with task scheduling. However, that scheduling requires certain information about the program, in particular, the maximum amount of memory possibly used by it. The worst-case execution time of the garbage collection task depends on the worst-case memory consumption.

We use the term *live memory* to denote the memory actually referenced by a program, i.e. the memory occupied by objects the program can use. In this paper, we present techniques for predicting the maximum amount of live memory in programs written in a type-safe object-oriented language such as Java. We also show how this live memory analysis relates to and interacts with timing analysis and traditional semantic analysis. The developed techniques have been implemented in a prototype environment allowing the user to interactively monitor predictions of memory consumption and execution time of a program during development.

1.1 Related Work

A number of efforts exist to adapt Java for use in real-time embedded systems, both by Sun Microsystems [6] and others [19]. Real-time garbage collection is fundamental to real-time Java. However, Henriksson's scheduling-based technique [10] requires information about the maximum amount of live memory for task scheduling. Nilsen's hardware-assisted technique [17] requires similar metrics, which must somehow be determined during system configuration.

Analysis of pointers and references is done in a variety of contexts. Alias analysis [13, 14] determines whether pointers possibly (or, in some cases, definitely) point to the same location and is used to decide which optimizations can be made during compilation. Shape analysis techniques [11, 23] determine the shape of data structures by analyzing the code.

Such analyses can provide conservative classifications of structures into a few broad classes (such as trees, directed acyclic graphs, or generally cyclic structures), but cannot be used to bound the sizes of these structures.

A related area is that of conflict analysis in parallelizing compilers. In the approach taken by Hendren et. al. [9], data structures are annotated with brief information about their shape, facilitating a relatively accurate conflict analysis.

None of the techniques above attempt to bound the amount of live memory possible in a program. Alias analysis and shape analysis techniques may provide partial information to a live memory analysis, but in general the information is not sufficient to determine the maximum amount of live memory.

As will be shown, our work is related to worst-case execution time (WCET) analysis [12, 15, 20, 24]. The present work is primarily related to source code level analyses [1, 5, 18].

1.2 Paper Outline

In Section 2 we present our approach and introduce some concepts and terms, followed by the live memory analysis algorithm in Section 3. Section 4 presents our prototype environment, including a brief overview of the kind of information the environment provides to the user. We discuss the applicability of the presented techniques in Section 5, where we also show the relation to real-time garbage collection. In Section 6 we present our conclusions and directions for future work, such as integration with other analyses.

2 Terminology and General Approach

In general, it is difficult (in practice impossible) to statically predict the exact behavior of garbage collection due to variations in input data. If such predictions were possible, we would be able to replace the garbage collection with properly placed deallocations altogether.

However, the garbage collector handles more information than we are interested in here; for example, we are not concerned with the specific identities of objects that may be referenced during run-time. We focus our work on *determining an upper bound on the amount of live memory* that can possibly exist in a given program at any time. This is a simpler problem than predicting the actual garbage collection in the general case.

To handle recursive data structures, our approach employs annotations on data structure declarations to aid the analysis. Such annotations

indicate, for instance, the maximum length of a linked list or the maximum depth of a binary tree. In general, these annotations indicate the longest possible sequence of linked objects in the annotated data structure. Note, however, that several such bounded data structures of the same type may exist simultaneously.

Although such annotations are probably overly restrictive in general, they are quite reasonable in real-time software for embedded systems. The amount of live memory affects the garbage collection overhead and thus we need an upper bound. In addition, dedicated embedded systems are typically designed for a specific task and are equipped with a suitable amount of memory for that task. (Virtual memory is generally not used in these systems due to the high costs for the infrequent page misses.) These memory limitations must be known and considered throughout development. Thus, the annotations represent information that should already be known to the programmer.

2.1 Type System Assumptions

Throughout this paper, we assume a Java-like object-oriented type system. Dynamic memory is allocated by creating objects. Every object is an instance of a class. A class contains declarations of data (scalar variables, such as *int*s and *boolean*s), references (referencing objects, or being *null*-valued), and methods. Subclasses may introduce additional data, references, and methods. A class may provide alternative implementations of methods inherited from its superclass.

Every reference is qualified by a class. We call this class the *static qualification* of the reference. The type system guarantees every reference to either have the *null* value, or reference an instance of its static qualification (or a subclass of it).

2.2 Bounding Recursive Data Structures

As long as inheritance and recursive data structures are avoided, the maximum amount of live memory can be conservatively (although rather pessimistically) estimated by assuming all references to refer to unique objects. However, the following three circumstances complicate the problem:

- *Recursive data structures.* Without additional information, it is impossible to determine the number of elements in a general recursive data structure, and thus the amount of memory occupied by it.
- *Unnecessary pessimism due to aliasing.* The assumption that all references reference unique objects is generally not true; references may be used for traversing data structures (similar to loop induction

variables) or to otherwise simplify data structure traversals (such as the back reference in a double linked list). Such references are redundant from a live memory analysis point of view.

- *Inheritance and method dispatching as used in object-oriented languages.* Inheritance implies that a reference with static qualification c also may refer to objects of subclasses of c. Since objects of these subclasses may contain more data and references than those of their superclass, the amount of live memory may be affected.

  Memory is referenced from activation records for procedures and methods. Consequently, we also need to consider method dispatching in our live memory analysis. This will be treated in Section 3.4.

2.3 Classification of References

To cope with recursive data structures, we need class declarations to be annotated with a maximum traversal length. Consider, for instance, a linked list. An annotation may be associated with the list element class and denote the maximum length of a linked list. We call a recursively defined class with such an annotation a *bounded class* and a reference to such a class a *bounded reference*.

Using the simple declarations in Figure 1 as an example, we divide references into four categories:

Entry: a reference to some sort of 'root' in the data structure, such as the first element of a linked list, the root node of a tree, or the first node in a directed acyclic graph (DAG). The *first* reference in Figure 1 is an entry in this sense.

Link: a reference introducing recursion into a class, such as the *next* reference in the example.

Redundant: a reference that always refers to data referenced by other non-redundant references, thus not referencing any additional live memory. Such redundant references are used, for example, as 'current' references while iterating through data structures, or to enable traversals in different directions (as the *pred* reference in Figure 1).

Simple: a reference to a non-recursively defined class.

To minimize the specification work for the programmer, we would like to automatically classify references into one of these categories. We employ a conservative classification scheme which may be complemented by annotations. References to recursively defined classes are either entries

```
class List {
    Element first;
}

class Element {
    int data;
    Element next;
    Element pred;
}
```

Figure 1 A simple recursive data structure: a doubly linked list.

or links: references within the declaration cycle are links, and references outside the cycle are entries. References to non-recursively defined classes are automatically classified as simple. Annotations are used to declare that a reference is redundant.

An interesting extension is to use additional analyses to assist the programmer in annotating the program. Such analyses may be used both to verify whether the annotations are consistent with the remaining program, and to refine the automatic default annotations (e.g. to detect redundant references). We address this topic further in Section 6.1.1.

The *Element* class in Figure 1 is recursively defined and can thus not be analyzed without additional information. The class further contains two recursive references. However, knowing that the class is in fact a doubly linked list, the programmer can immediately deduce that the *pred* reference is redundant.

The doubly linked list is given in its annotated form in Figure 2. This representation includes information about the maximum possible length of a list, as well as the usage of the list (the *pred* reference is redundant). The *path-bound* annotation indicates the longest possible sequence of objects in the recursive data structure, which in this case corresponds to the maximum length of a list.

Note that without the redundancy information, the data structure above would be indistinguishable from a binary tree of depth 50. Such a tree may contain up to $2^{50} - 1 \approx 1.12 \cdot 10^{15}$ elements rather than the 50 in a list. A live memory analysis without the redundancy information would clearly be exceedingly pessimistic.

```
class List {
    Element first;
}

class Element /*$ path-bound 50 */ {
    int data;
    Element next;
    Element pred; /*$ redundant */
}
```

Figure 2 The doubly linked list in annotated form.

3 Live Memory Analysis

The maximum amount of live memory depends on the execution point. Memory can be referenced from not only references within the current activation record, but also from all activation records throughout the call chain, from the current record to the outermost scope of the program. To handle this chain of activation records, the prediction of the maximum amount of live memory is performed in two steps:

1. Predicting the maximum amount of memory referenced by a given reference variable.

2. Predicting the maximum amount of memory referenced from a block, possibly via a set of method activation records.

We start with a basic algorithm in Section 3.1. Although this algorithm is unable to deal with some data structures, it serves as an introduction to the more general algorithm in Section 3.2. In Section 3.3, we present a further generalization to accommodate inheritance, and in Section 3.4, we present techniques for predicting the amount of memory referenced from a block.

The algorithms are given in a form suitable for implementation using attribute grammars in, for instance, a compiler-compiler environment. We outline an environment based on such an implementation in Section 4. However, a number of other implementation schemes are possible.

3.1 Basic Algorithm

The idea of our basic live memory analysis algorithm is to recursively compute the size (in bytes) of the maximum set of objects in the bounded data structure. The function $R(p, c, n)$ recursively computes the maximum

amount of memory referenced from the reference p. The two additional parameters, c and n, are used to convey information between recursive calls regarding the analyzed recursive data structure. The currently traversed bounded class is c. The integer n represents the remaining depth to analyze, i.e., the number of times a link to c yet can be traversed.

The algorithm is initially called with the parameters $(p, \mathrm{NO\_CLASS}, 0)$, meaning that no bounded class is initially traversed.

$$R(p, c, n) = \mathrm{sizeof}(c_p) + \sum_{q \in \mathrm{refs}(c_p)} R'(q, c, n)$$

In this equation we denote the static qualification of a reference p by c_p, the size of an object of class c by $\mathrm{sizeof}(c)$, and the set of references declared in class c by $\mathrm{refs}(c)$. We thus compute the maximum amount of memory referenced from p as the sum of the size of the referenced object[1] and the sum of the memory referenced by references in c_p.

For non-recursive data structures, the function R' is identical to R, intuitively computing the maximum amount referenced memory. To cope with recursive data structures, however, R' is defined as follows:

$$R'(q, c, n) = \begin{cases} 0 & q \text{ redundant} & \textit{(i)} \\ R(q, c_q, B(c_q)) & q \text{ entry} \wedge c = \mathrm{NO\_CLASS} & \textit{(ii)} \\ R(q, c, n-1) & q \text{ link} \wedge c = c_q \wedge n > 0 & \textit{(iii)} \\ 0 & q \text{ link} \wedge c = c_q \wedge n = 0 & \textit{(iv)} \\ R(q, c, n) & q \text{ simple} & \textit{(v)} \\ \text{– Error –} & c \neq c_q \wedge c \neq \mathrm{NO\_CLASS} & \textit{(vi)} \\ & \wedge\ (q \text{ entry} \vee q \text{ link}) & \end{cases}$$

Case *(i)* handles redundant references, which do not contribute to the maximum amount of live memory. Case *(ii)* occurs when an entry is encountered; the reference is traversed as usual, but the c and n parameters are set to the class c_q and its bound $B(c_q)$. Case *(iii)* occurs on traversals of a link in the data structure; again, the reference is traversed, but the n parameter is decreased. Case *(iv)* is the base case, which occurs when a link has been traversed the number of times the class bound specifies. Case *(v)* represents traversals of other (non-recursive) unbounded classes. The final case, *(vi)*, occurs when a reference to a bounded class (entry or link) is encountered while traversing a another bounded class. Since the two parameters of the function can only represent the traversal of one bounded class at a time, the algorithm fails in this case.

This algorithm can be used to analyze several common data structures, such as linked lists and trees. However, it prohibits the use of two or more

[1]We assume a compacting real-time GC algorithm, as in [10]. This implies that no fragmentation overhead between objects needs to be accounted for.

different bounded classes in the same data structure, such as a bounded binary tree with a bounded linked list in each node. If such data structures are to be analyzed, we must be able to perform an analysis even when multiple bounds are interleaved, rather than signaling an error as is done above. The following general algorithm is designed to analyze such interleaved bounds.

3.2 General Algorithm

For the general form of the algorithm, we use a *bound set* S consisting of tuples of the form $\langle c, n\rangle$. Each such tuple indicates that the longest possible remaining sequence of c references is n. We have thus generalized the second and third parameter of the basic algorithm to a set of such pairs. (Note, however, that any particular class c occurs in at most one tuple in S; i.e., S is a mapping from classes to integers.) This generalization allows us to analyze data structures with multiple cooperating bounds.

As a shorthand, we introduce two infix binary operators. The first is called the *insert* operator, is written $S \oplus c$, and adds the tuple $\langle c, B(c)\rangle$ to S. If c is already associated with some integer in S, that association is removed. The operator is defined as

$$S \oplus c = \begin{cases} (S - \{\langle c, a\rangle\}) \cup \{\langle c, B(c)\rangle\} & \exists\, a : \langle c, a\rangle \in S \\ S \cup \{\langle c, B(c)\rangle\} & \text{otherwise} \end{cases}$$

where $-$ is the relative complement operator and $\cup$ is the union operator.

The second operator is called the *decrease* operator, is written $S \oslash c$, and decreases the integer associated with a class c in S by one. It is defined only if S contains the tuple $\langle c, n\rangle$ for some n:

$$S \oslash c = (S - \{\langle c, n\rangle\}) \cup \{\langle c, n-1\rangle\}$$

We then generalize the R function as follows:

$$R(p, S) = \text{sizeof}(c_p) + \sum_{q \in \text{refs}(c_p)} R'(q, S)$$

$$R'(q, S) = \begin{cases} 0 & q \text{ redundant} & \textit{(i)} \\ R(q, S \oplus c_q) & q \text{ entry} & \textit{(ii)} \\ R(q, S \oslash c_q) & q \text{ link} \wedge \exists\, a > 0 : \langle c_q, a\rangle \in S & \textit{(iii)} \\ 0 & q \text{ link} \wedge \langle c_q, 0\rangle \in S & \textit{(iv)} \\ R(q, S) & q \text{ simple} & \textit{(v)} \end{cases}$$

The five cases in this version of the R' function correspond directly to the first five cases of the previous version. However, there is no sixth

error case; interleaved traversals of multiple bounded classes are accommodated by case *(ii)*.

3.3 Inheritance in Object-Oriented Languages

The inheritance mechanism in object-oriented languages is not accounted for in the algorithms just presented. A reference declared to reference objects of class c can point to not only c objects, but also to objects of subclasses of c. Subclasses may add data to the object layout, causing objects of subclasses of c to be larger than c objects. The analysis above is thus optimistic in presence of inheritance.

We now describe the extensions necessary for the general algorithm to give safe predictions of referenced memory when inheritance is used. These extensions are based on the following two observations:

- *Subclasses may introduce additional data or references.* A reference declared to reference objects of class c can reference objects of subclasses of c. The function $R(p, S)$ must thus compute the maximum amount of referenced memory from an object of class c_p *or any of its subclasses*.
- *Superclasses may introduce bounds.* Every instance of a class can be considered to be an instance of any of its superclasses. When traversing a bounded class c, it is not sufficient to test the static qualification of an encountered reference p for exact equality with c; if the static qualification of p is a subclass of c, it must still be considered to point to a part of the bounded data structure. That is, bounds should be inherited. (Bounds must not be redefined by subclasses, however.)

With these observations in mind, the extensions to the general algorithm are straight-forward. Using the notation $c_p \preceq c_q$ ($c_p \succeq c_q$) to indicate that c_p is equal to or a subclass (superclass) of c_q, the algorithm looks as follows:

$$R(p, S) = \max_{c \preceq c_p} \left(\text{sizeof}(c) + \sum_{q \in \text{refs}(c)} R'(q, S) \right)$$

In this equation, we assume $\text{refs}(c)$ to include references declared in c as well as in superclasses of c.

$$R'(q, S) = \begin{cases} 0 & q \text{ redundant} & \textit{(i)} \\ R(q, S \oplus c_q) & q \text{ entry} & \textit{(ii)} \\ R(q, S \oslash c_q) & q \text{ link} \wedge \exists\, c \succeq c_q, a > 0 : \langle c, a \rangle \in S & \textit{(iii)} \\ 0 & q \text{ link} \wedge \exists\, c \succeq c_q : \langle c, 0 \rangle \in S & \textit{(iv)} \\ R(q, S) & q \text{ simple} & \textit{(v)} \end{cases}$$

Again, the five cases in this algorithm correspond directly to those in the previous algorithms.

3.4 References from Entire Blocks and Activation Records

When extending the techniques above to bound the memory referenced from an entire block (compound statement, procedure body etc), the effects of references from activation records throughout the call chain must be taken into account. Bearing this in mind, we estimate the maximum amount of memory referenced from a block as

$$R_{block}(b) = \left(\sum_{d \in b.decls} R(d) \right) + \left(\max_{s \in b.stmts} R_{stmt}(s) \right)$$

where $R_{stmt}(s)$ is the maximum amount of memory referenced during the execution of statement s, $b.decls$ is the set of reference declarations in b, and $b.stmts$ is the set of statements in b. For procedure calls, $R_{stmt}(s)$ is computed from the procedure body using the same equation as above; for other statements, $R_{stmt}(s) = 0$.

Virtual methods require special consideration. Since the executed code is not determined until run-time, $R_{stmt}(s)$ must equal the maximum amount of live memory referenced from *any* implementation of the called method.

These considerations can be summarized as

$$R_{stmt}(s) = \begin{cases} R_{block}(s.decl.block), & s \text{ procedure call} \\ \max\limits_{d \in \mathrm{impl}(s.decl)} R_{block}(d.block), & s \text{ method call} \\ 0, & \text{otherwise} \end{cases}$$

where $\mathrm{impl}(s)$ denotes the set of implementations of the method s and $s.decl.block$ denotes the body of the called procedure or method.

Recursive procedures and methods require special care, since the number of recursive calls somehow must be bounded. This requirement is common in real-time systems; for the worst-case execution time to be bounded, the depth of recursive calls must also be bounded. Such bounds may either be given in explicit annotations from the programmer, or determined by special analyses such as the one presented in the following subsection.

3.5 Relation to Timing Analysis and Semantic Analysis

As mentioned in Section 1, live memory analysis can provide input to the scheduling of garbage collection in real-time systems. In addition,

the following three analyses (amongst others) are applicable for real-time systems:

- *name analysis*, that is, the mapping of occurrences of identifiers to declarations,
- *type analysis*, and
- *worst-case execution time (WCET) analysis*.

Although the area of WCET analysis is specific to real-time systems, the former two analyses are performed during compilation of programs in all high-level languages. Live memory analysis can be performed in isolation for many programs, but much can be gained from integration with these other analyses, both in terms of higher accuracy and lower complexity.

The algorithms presented in this section all use information that can be obtained from name analysis. The algorithms for bounding referenced memory in Sections 3.1, 3.2, and 3.3 all require information about the layout of referenced objects. Such a mapping from reference declarations to classes is typically computed during name analysis. Similarly, the equations in Section 3.4 require information about procedure/method declarations at the call site, which is also determined during name analysis.

In addition to this name analysis information, the algorithm in Section 3.3 requires information about inheritance relationships between classes. These relationships are typically determined during type analysis.

The result of the same algorithm also depends on the depth of recursive calls. Determining an upper bound for that depth is a task for the WCET analysis.

The WCET analysis can benefit from an integration with live memory analysis. An important subproblem in WCET analysis is to determine upper bounds for the number of iterations in loops. Although such bounds are difficult to compute in general, they can in some important special cases be derived from the data structure annotations presented in Section 2.3. A general discussion of bounding loop iterations and recursive calls using data structure annotations is beyond the scope of this paper, but we outline some interesting points of contact with the present work.

One important use of loops is to traverse data structures. Such traversals often follow a generic pattern:

```
int traverseList(List l) {
    Element e = l.first;
    while (e != null) {
        ...
        e = e.next;
    }
    ...
}
```

If the list is null-terminated (which may be indicated by an annotation, say `/*$ null-terminated */`), and the list is not modified during the iteration, we can conclude that the *while* loop will not iterate more times than the bound annotation of the *Element* class specifies.

Similarly, a common use of recursion is to traverse data structures. Consider the following method (assumed to be declared in the *Element* class):

```
Element find(int data) {
    if (data == this.data)
        return this;
    else if (next != null)
        return next.find(data);
    else
        return null;
}
```

The depth of the recursive call `next.find(data)` is bounded by the bound of the *Element* class. Again, we require the data structure to be null-terminated.

4 The Environment

We have implemented prototype analyses of memory allocation and execution time for a simple object-oriented language in APPLAB (APPlication language LABoratory), an environment originally designed for interactive development of domain-specific languages [2, 3]. APPLAB integrates structure-oriented editing with semantic analysis and code generation and is based on reference attributed grammars [8], an extended attribute grammar formalism. The language our tool operates on supports advanced object-oriented language concepts such as classes and inheritance.

Separate attribute grammar modules hold specifications of live memory analysis, WCET analysis, type analysis, name analysis, and code gen-

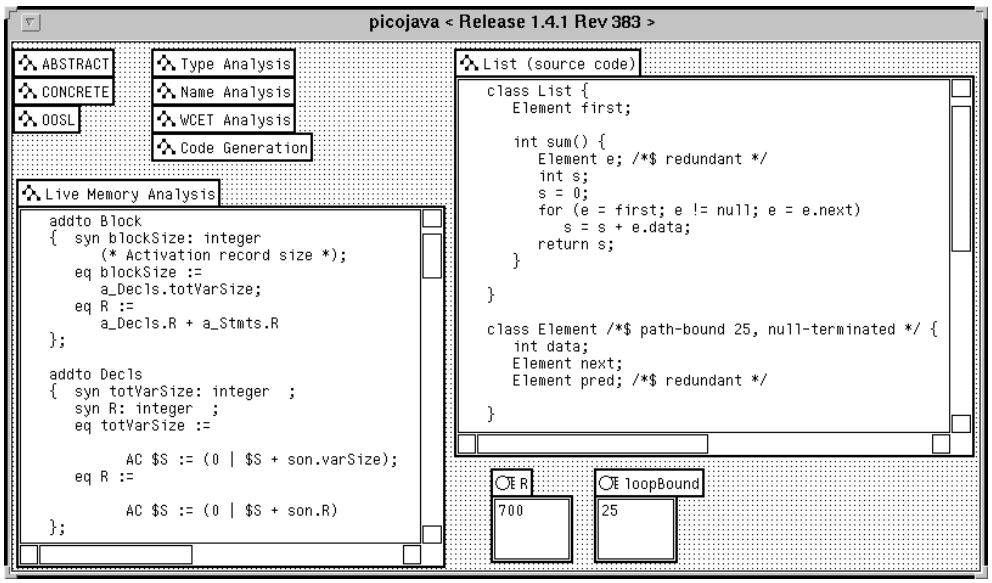


Figure 3 Example live memory analysis. The window labelled *R* shows the maximum amount of memory referenced from the *first* reference. In addition, the *loopBound* window shows the maximum number of iterations in the *for* loop.

eration. In Figure 3 a screendump is given, showing excerpts of an example program as well as the memory analysis module. The screendump also shows sample predictions of memory consumption and loop iteration bounds. Such predictions can be interactively viewed while the program is being edited.

The tool uses data structure annotations as presented in Section 2.3. These annotations are expressed as special 'tagged' comments and thus pose no problem to a traditional compiler, but our tool can parse them and use the information therein.

The live memory analysis currently implemented is based on a subset of the algorithms presented in Section 3. The environment provides predictions of the amount of memory referenced from an arbitrary reference or block. The WCET analysis is based on the timing schema approach [20] and provides timing information in terms of source code constructs to the programmer during development.

The live memory analysis utilizes information from name analysis, type analysis, and data structure annotations. The WCET analysis uses information from name analysis, code annotations, and data structure annotations. The implementations of the different modules are separated, however; for example, the complexity of name analysis for object-oriented

languages is not reflected in either the live memory analysis or the WCET analysis.

5 Discussion

The algorithms presented in Section 3 are applicable for live memory analysis of data structures with increasing degree of complexity. The basic algorithm in Section 3.1 is the least complex to implement and has no analysis-time memory requirements beyond function call administration. It cannot analyze data structures where two or more bounds are combined (such as a DAG with a bounded list of edges in each node, as well as a path bound for nodes themselves). However, it can be used as-is for large classes of common data structures, such as lists and trees. If a DAG is implemented with an array of edges (rather than a bounded list) in each node, the basic algorithm can be used for that data structure as well.

The general algorithm in Section 3.2 handles data structures with interleaved bounds, like the DAG just mentioned. This generality brings a slightly higher analysis-time cost since the set of traversed bounds must be maintained. However, we expect the size of the bound set to rarely become larger than three or four elements.

An extension of the general algorithm to provide safe approximations in the presence of inheritance was presented in Section 3.3. This extension may also be adapted to the basic version of the algorithm without difficulty.

As outlined in Section 3.5, bounds for loop iterations and recursion depth can in some common important cases be derived from data structure annotations. This saves work on the programmer's part, since no additional annotations need to be associated with traversal loops. Since the size of a data structure is a property of the data structure rather than the code using it, we also consider these annotations to be more intuitive than code annotations.

The presented techniques are primarily intended to be used off-line during system development to determine scheduling parameters. However, if dynamic class loading is to be employed (as is possible in Java), a live memory analysis must be performed on-line whenever a class is loaded to determine whether the garbage collection can still be scheduled. Some of the presented algorithms also require global information, such as the subclasses of a particular class or implementations of a particular method. If that information is not available at compile time (for example, due to restrictive separate compilation), the live memory analysis may be performed on-line at class loading time.

We have based our analyses on explicit annotations associated with

data structure declarations, similar to the annotations used by Hendren et. al. for parallelizing compilers [9]. In practice, however, it is desirable to associate bounds with *uses* of (i.e. references to) data structures. Such an extension would allow the existence of data structures of the same type but with different bounds in a program. This extension is outside the scope of this paper; however, it does not affect the presented algorithms, only the form of the annotations.

5.1 Real-Time Garbage Collection — an Application for Live Memory Analysis

As mentioned in Section 1, an important application for live memory analysis is to obtain metrics for scheduling real-time garbage collection. In the remainder of this section we give a brief overview of the approach developed by Henriksson for his thesis [10].

In this approach, the tasks of a real-time system are divided into two groups: high-priority (HP) tasks and low-priority (LP) tasks. (The exact priorities of tasks are not important here, as long as all HP tasks have higher priorities than all LP ones.) An additional garbage collection process, with priority *between* the LP and the HP tasks, is scheduled to guarantee that initialized (i.e. zeroed) memory is always available for the HP tasks when they require it. The LP tasks, on the other hand, perform initialization and garbage collection work along with object allocations within their own contexts.

The garbage collection algorithm used is a variant of Brooks' algorithm [4], which is a compacting algorithm. The *garbage collection work* W that must have been performed at any given time can be expressed as

$$W \geq \frac{W_{max}}{S - E_{max} - M_{HP}} \cdot A$$

where A is the amount of new objects (allocated within the current garbage collection cycle). W_{max} denotes the worst-case amount of work to perform during a garbage collection cycle and depends on the maximum amount of live memory. For the purposes of this overview, W and W_{max} can be considered to be measured in seconds. In addition, S denotes the amount of available memory[2], E_{max} denotes the maximum amount of live memory, and M_{HP} denotes the amount of memory that is pre-initialized to be directly available to HP tasks. (M_{HP} depends on the worst-case allocation rate of HP tasks, which we do not deal with in this paper.)

[2]More precisely, S denotes the size of *tospace* as used in Brooks' algorithm [4].

6 Conclusions

We have presented techniques for determining an upper bound on the amount of live memory possible in a task. Such bounds are necessary for predictable garbage collection in embedded systems. The live memory analysis algorithms compute conservative estimations which may be tightened by annotations representing programmer knowledge.

We base our approach on annotations associated with data structure declarations. These annotations allow us to analyze recursive data structures in a type-safe object-oriented language such as Java. Care is taken to accomodate object-oriented language concepts such as inheritance and virtual methods.

We have outlined how the WCET analysis can benefit from the annotations as well. An important and common use of loops and recursive calls is to traverse data structures, and the presented data structure annotations can in many cases be used to compute bounds for loop iterations and recursion depths. We claim this approach to save the programmer from unnecessary repetitive code annotations.

Although our prototype tool operates on a simple fictive language, it shows how data structures in a type-safe object-oriented language like Java may be conveniently annotated and automatically analyzed.

6.1 Future Work

We are working on extending our tool to handle the Java language and further investigate the interplay between live memory analysis, WCET analysis, and semantic analysis. As mentioned in Section 5, it would be advantageous to associate bounds with *uses* of data structures rather than *declarations*. Such an extension allows data structures of the same type but with different bounds to co-exist. We would also like annotations to reference other annotations. For instance, annotations for complex traversing loops (which cannot be analyzed using the techniques outlined in Section 3.5) may reference annotations for the traversed data structure. Similarly, iterator objects (as used in the Java libraries) may exploit this information.

As mentioned in Section 5.1, Henriksson's real-time garbage collector requires some more information than a live memory analysis can provide, such as the worst-case allocation rate of high-priority processes. We plan to investigate an approach analogous to the timing schema [20] for this purpose.

6.1.1 Combination with Other Analyses

The presented live memory analysis can be complemented by a number of other analyses. In Section 3.3 a reference with static qualification c was assumed to possibly reference an object of class c or any subclass of c. A points-to analysis [21] can provide more detailed information about which objects a given reference actually can refer to, excluding a number of impossible cases. Similar analyses can be used to determine which implementations of a virtual method can be called from a given call site [22], thus improving on the pessimistic assumptions in Section 3.4.

The redundancy annotations presented in Section 2.3 would probably benefit from a complementing alias analysis. Such an analysis can provide redundancy information automatically in some cases. However, since such an analysis must be conservative, we believe explicit programmer annotations to be useful where an alias analysis may be unable to give a sufficiently exact result. The redundancy annotation in the doubly linked list example in Section 2.3 may be difficult to determine by an alias analysis, but significantly influences the live memory analysis. In such cases, an alias analysis may be used to verify explicit annotations by attempting to find contradictions between annotations and uses of references.

Although alias analysis may be used to verify some annotations, complete static verification is not possible in the general case. (As in all verification work, we can ultimately only find faults, not prove their absence.) A reasonable approach would be to use various run-time checks during development, and disable those checks in the final product in a fashion similar to contracts [16].

Acknowledgments

Görel Hedin, Klas Nilsson, Roger Henriksson, David Wise, Anders Ive, Fredrik Dahlstrand, Jonas Persson, and the anonymous reviewers provided helpful comments and inspiring suggestions for this paper.

This work was financed by ARTES (A network for Real-Time research and graduate Education in Sweden).

References

[1] P. Altenbernd. On the False Path Problem in Hard Real-Time Programs. *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, 1996.

[2] E. Bjarnason. *Interactive Tool Support for Domain-Specific Languages*. Licentiate Thesis, Department of Computer Science, Lund University, December 1997.

[3] E. Bjarnason, G. Hedin, and K. Nilsson. Interactive Language Development for Embedded Systems. *Nordic Journal of Computing*, Vol. 6, pages 36-55, 1999.

[4] R. A. Brooks. Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware. *Proceedings of the 1984 Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[5] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculations of Execution Time. *Proceedings of EuroPar '97*, LNCS 1300, Springer, August 1997.

[6] W. Foote. Real-Time Extensions to the Java™ Platform — A Progress Report. *Proceedings of the RTSS '98 Work-In-Progress Session*, Madrid, Spain, December 1998.

[7] J. Gosling, B. Joy, and G. Steele. *The Java Language Language Specification*. Addison-Wesley, 1996.

[8] G. Hedin. Reference Attributed Grammars. *Proceedings of WAGA'99: Second Workshop on Attribute Grammars and their Applications*, pages 153-172. Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.

[9] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*, June 1992.

[10] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD Thesis, Department of Computer Science, Lund University, September 1998.

[11] N. D. Jones and S. S. Muchnick. A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures. *Conference Record of the Ninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'82)*, January 1982.

[12] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An Accurate Worst Case Timing Analysis Technique for RISC Processors. *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'96)*, December 1996.

[13] J. R. Larus and P. N. Hilfinger. Detecting Conflicts Between Structure Accesses. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*, June 1988.

[14] W. Landi and B. G. Ryder. Pointer-induced Aliasing: A Problem Taxonomy. *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL'91)*, 1991.

[15] T. Lundqvist and P. Stenström. Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques. *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, Montreal, Canada, June 1998.

[16] B. Meyer. Applying "Design by Contract". *IEEE Computer*, 1992.

[17] K. Nilsen. Reliable Real-Time Garbage Collection of C++. *Computing Systems*, 1994.

[18] P. Persson and G. Hedin. Interactive Execution Time Predictions using Reference Attributed Grammars. *Proceedings of WAGA'99: Second Workshop on Attribute Grammars and their Applications*, pages 173-184. Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.

[19] Real-Time Java™ Working Group.
http://www.newmonics.com/webroot/rtjwg.html

[20] A. C. Shaw. Reasoning about Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, Vol. 15, No. 7, 1989.

[21] M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 1-14. Paris, France, 1997.

[22] V. Sundaresan, C. Razafimahefa, R. Vallée-Rai, and L. Hendren. Practical Virtual Method Call Resolution for Java. Sable Technical Report No. 1998-7, McGill University, Canada, 1998.

[23] M. Sagiv, T. Reps, and R. Wilhelm. Solving Shape-Analysis Problems in Languages with Destructive Updating. *Proceedings of the Twenty Third Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, January 1996.

[24] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing Analysis for Data Caches and Set-Associative Caches. *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, June 1997.

Paper III

An Interactive Environment for Real-Time Software Development

Patrik Persson and Görel Hedin

Abstract

Object-oriented languages, in particular Java, are beginning to make their way into embedded real-time software development. This is not only for the safety and expressiveness of the source language; the mobility and dynamic loading of Java bytecode make it particularly useful in embedded real-time systems.

However, using such languages in real-time systems makes it more difficult to predict the worst-case execution time of tasks. Such predictions are necessary for predictable task scheduling in the developed system. Garbage collection, common in object-oriented languages, must be considered; to schedule garbage collection safely, we must know how much memory it has to handle. Dynamic binding in conjunction with dynamic loading of code also needs treatment.

We show how techniques for predicting time and memory demands of object-oriented programs are integrated into the *Skånerost* development environment. The environment explicitly targets an iterative development process, which is particularly important in real-time software development since time and memory demands cannot be determined until the code is written. Design changes due to timing problems become more costly as development progresses, and Skånerost allows such problems to be detected early.

1 Introduction

In the past, programming of embedded hard real-time systems was often done by hardware engineers primarily to make the hardware work. The programs were small and typically written in assembly or low-level C code.

Today, more microprocessors are sold for use in embedded systems than for desktop computers. Embedded real-time systems become more commonplace, but they also become more complex. Mobile telephones, photocopiers, and industrial robots are today all based on substantial amounts of software. At the same time, many embedded systems (such as aircraft control systems) are safety critical.

To reduce development effort and increase portability and reliability, a number of efforts [16, 22] are made to adapt high-level languages, in particular Java [5], to hard real-time applications. Besides being a modern object-oriented source language, Java is compiled to portable bytecode for execution by a Java virtual machine. This enables dynamic loading of code, which is of particular interest in many embedded systems, such as industrial robots and satellites.

Java is designed to be a safe language. Compile-time checks are used as far as possible, and run-time checks are used in the remaining cases. This safety of the language facilitates early detection of errors and thus implies safety in the developed system. Such safety is important in the dynamically configured systems mentioned above, where dynamically loaded code must not cause any existing code to fail. In other cases, where predictability requirements rule out dynamic solutions, the need for safety is yet more emphasized.

Real-time software development also places pressure on the software development tools. One of the most fundamental features of a piece of real-time software, its worst-case execution time, is also one of the most difficult ones to predict. Modern object-oriented languages, with mechanisms such as dynamic binding (virtual methods) and garbage collection, complicate the problem further.

Since the correctness of the real-time system as a whole depends on whether timing requirements are met, it is of paramount importance to provide continuous timing feedback to the programmer during development. The *Skånerost*[1] development environment, presented in this paper, is designed to provide such feedback.

[1]The name refers to a rather strong coffee blend originating in the Skåne province in southern Sweden.

1.1 Paper Outline

In Section 2 we present the aspects of real-time software development that influence our development tool design, including some related work. In Section 3 we present our approach to predictable real-time Java and some associated analysis techniques. In Section 4 we present the Skånerost environment. Section 5 concludes the paper.

2 Aspects of Real-Time Software Development

The key property of a real-time system is that it performs its computations within some sort of deadlines. The severity of missing a deadline depends on the application; in an engine control system, it may result in permanent engine damage, whereas in a real-time video application, it may merely result in a transient image distortion.

2.1 Worst-Case Execution Time

Real-time systems are typically modeled as a set of concurrent, periodic tasks. These tasks must be *scheduled* in a way that ensures that the system can fulfill its timing requirements. More precisely, the task schedule in a real-time system is based on the following parameters:

Task period: the interval with which the task is launched for execution.

Task deadline: the point in time when a particular execution of the task must be finished.

Worst-case execution time (WCET): the largest amount of execution time possibly required for any execution of the task.

Real-time scheduling is a well established research area and a number of scheduling techniques are available, e.g., [13]. However, they all require values for the parameters above (and possibly more).

Whereas the first two parameters (periods and deadlines of tasks) are usually design parameters, the third parameter, the worst-case execution time, cannot be deduced from the requirements. Instead it is a property of the executable code and the hardware it runs on. A *WCET analysis* of the code is required to predict its WCET, based on some model of the target environment. (Our target environment is described in Section 3.) A requirement for the WCET can be stated, of course, but that requirement must still somehow be verified.

2.2 Supporting an Iterative Development Model

It is desirable to obtain WCET predictions throughout development, not just for the final code. Such predictions may concern the execution time of individual loops, methods, or entire tasks. Should these predictions indicate that the system's timing requirements cannot be met, either the design must be revised or the timing requirements must be reassessed. In any case, such a timing problem should be detected as early as possible during development.

2.3 Related Work

Most approaches to WCET analysis are targeted towards object code, e.g., [11, 12, 14, 25]. Such analyses have difficulties with object-oriented languages, where invocations of virtual methods are compiled to indirect jumps via pointers. Object code analyses can, in general, not analyze such function pointers.

There are very few publications available on WCET analysis for object-oriented languages besides our own work. Bernat [2] presents an approach to WCET analysis of Java bytecode, but does not discuss dynamic binding or garbage collection. Gustafsson [6] uses abstract interpretation to analyze programs in a special real-time dialect of Smalltalk. He uses type inference to facilitate a WCET analysis of method calls, but does not address the issues of dynamic loading and garbage collection as done in this paper.

The tool we will present allows the programmer to obtain predictions of code portions throughout development. Other approaches typically use less interactive techniques such as integer-linear programming (ILP) [11] or abstract interpretation (symbolic execution) of the object code [6, 14] (which may not terminate). Ko et al. [10] describe an interactive timing analysis environment; however, in that work, the interactive user interface is invoked after a complete analysis of the entire program. They further focus on compiled C code and hardware aspects, rather than the Java bytecode approach in the present work.

3 Predictable Real-Time Java

The target language of our techniques is a subset of Java, supporting classes, inheritance, and dynamic binding. Excluded language features currently include method overloading, interfaces, and exceptions; these will be considered in the future.

We also support the compilation of Java to portable bytecode, a technique originally developed to facilitate code mobility over the Internet in

the form of *applets*. Our motivation is somewhat different. Being able to transport code over a network and dynamically load it into a virtual machine in a safe manner is particularly useful in the context of embedded systems, and has enabled interesting consumer technologies such as Jini [23]. The bytecode approach can also add safety to dynamic loading of embedded controller code as previously done in C/C++ [18].

The bytecode language constitutes a clean cut between the high-level language and the execution environment. This division is useful for WCET analysis. By encapsulating the timing properties of the virtual machine into a special *virtual machine timing model*, porting of the WCET analysis to new platforms is made easier. In our current timing model, we ascribe a constant WCET to each bytecode instruction.

As we will discuss more when we treat the analyses, we use *annotations* in the source code to provide information that should be obvious to the programmer but is difficult for a tool to deduce. These annotations are expressed as source code comments adhering to a special syntax. Such annotations can be identified and used by an analysis tool, yet ignored by a traditional compiler.

One particular case where annotations are sometimes needed is loops. In many important cases, the maximal number of loop iterations can be deduced automatically from the code. For more complicated loops, however, explicit annotations are required to indicate the upper bound. In any case, the halting problem implies that no tool can automatically analyze arbitrary loops.

Although this may at first appear to be a restriction of the language, it is in fact an ubiquitous circumstance of real-time software: if no loop bounds exist, the WCET cannot be bounded either, and the code cannot be scheduled safely. The programmer must thus be aware of these bounds, and it is reasonable to require this knowledge to be stated in the source code.

3.1 WCET Analysis Techniques

A WCET analysis is used to predict the worst-case execution times of tasks. Using object-oriented languages like Java in real-time systems complicates the analysis:

Garbage collection requires special treatment. In a real-time system, a classic “stop-the-world” garbage collector would introduce indeterministic response times and invalidate the assumptions of the task scheduling.

Dynamic binding (virtual methods) must also be considered. Unlike ordinary function or procedure calls, the code to execute upon a virtual

method call is not statically known: it is determined at run-time. Static WCET prediction of such a call requires special techniques.

Our general approach to WCET analysis is based on the timing schema [21] in an attribute grammar context [20]. We will now discuss our approach to the just mentioned issues in WCET analysis for object-oriented languages.

3.2 Real-Time Garbage Collection

As shown by Henriksson [9], a garbage collector can be used in a hard real-time system by proper *scheduling* of the garbage collector. The garbage collection work is scheduled into suitable time slots, and high-priority tasks are guaranteed instant access to free memory. We base this discussion on Henriksson's real-time garbage collection approach.

Predictable garbage collection is not only a matter of execution time. It is also important that the *fragmentation of memory* is predictable (and, of course, preferably small). If it is not, real-time tasks cannot be guaranteed access to memory. To handle fragmentation, a compacting garbage collector is used; that is, the heap is constantly re-organized to keep fragmentation low. (Such a garbage collector may appear to harm the response times of real-time tasks. To the contrary, however, the scheduling approach allows these response times to be *reduced*, since memory management administration is handled entirely in a separate task.)

As the preceding discussion suggests, the amount of garbage collection work that needs to be scheduled depends on the memory consumption of the program at hand. Henriksson's approach requires information about the amount of *live memory* used by the program, that is, the memory occupied by objects the program can use. Other real-time garbage collectors use similar parameters [17].

Analogously to the WCET analysis, a *live memory analysis* can be used to compute an upper bound on the amount of live memory required by a program. For simple, non-object-oriented programs this upper bound can be conservatively (but pessimistically) estimated by assuming all references to refer to unique objects. In general, however, some considerations must be made:

Recursive data structures. Without additional information, it is impossible to determine the number of elements in a general recursive data structure, and thus the amount of memory occupied by it.

Aliasing. The assumption that all references reference unique objects is generally not true; references may be used for traversing data structures (similar to loop induction variables) or to otherwise simplify

data structure traversals (such as the back reference in a double linked list). Such references are redundant from a live memory analysis point of view.

Inheritance. Inheritance implies that a reference with static qualification c also may refer to objects of subclasses of c. Since objects of these subclasses may contain more data and references than those of their superclass, the amount of live memory may be affected.

Dynamic binding. Dynamic binding affects control flow, which in turn affects memory use.

We use annotations from the programmer to get information that is known to the programmer but difficult for a tool to deduce accurately, such as shape and size of data structures. The exact form of our annotations will be presented in Section 4, where we present our tool.

Information about the sizes of data structures is, of course, vital to the live memory analysis. Information about the *shape* of data structures is equally important; without it, a doubly linked list of length 50 would be indistinguishable from a binary tree of depth 50. (Both are typically described by a class C containing two recursive references to C itself.) Such a tree may contain up to $2^{50} - 1 \approx 1.12 \cdot 10^{15}$ elements rather than the 50 in a list. Although techniques for shape analysis exist for other purposes, these are too conservative to facilitate an accurate live memory analysis. Other researchers have recognized the need for annotations to improve the accuracy of shape analysis [8].

Based on these rather straight-forward annotations, our live memory analysis algorithm [19] computes an upper bound on the amount of live memory in a program.

3.3 Dynamic Binding

An automatic WCET analysis of method calls is possible in some cases. If information about all possibly called implementations of a particular method are available for analysis, one safe WCET estimation is the longest WCET of these implementations.

This approach assumes that *any* of the existing implementations may be called from a given call site. Such an approximation may be improved by using existing type analysis techniques (e.g., [24]) developed for optimizing compilers. A similar approach is taken in [6].

However, a global analysis is not always possible. Java's dynamic class loading allows new implementations of a given virtual method to be introduced at run-time. We suggest another approach to handling timing re-

quirements on virtual methods (in the context of dynamic class loading), based on two observations:

- *The WCET of a method call should be known to the programmer implementing that call, even if the implementation is not yet available.* If the execution time of the call is unknown, the programmer has no way of fulfilling the timing requirements.
- *When a new class is loaded, it must not break the timing assumptions of the existing system.* A new implementation of a virtual method should not cause any code *calling* that method to miss its deadline.

These requirements are analogous to those for the type system in any statically typed programming language. For example, assume that the function *f* accepts a single integer argument. This information is used both for semantic analysis of calls to *f* (to ensure that a call passes an integer argument) and of *f* itself (to ensure that it accepts an integer argument).

Timing constraints, such as bounds on WCETs of virtual methods, constitute interface information and should be treated in a manner similar to type information. We thus advocate *expressing timing constraints on a virtual method in the method's signature, along with the types of the parameters and the return value.* This information is expressed as annotations, as discussed in the beginning of Section 3.

To give concrete form to the preceding discussion, Figure 1 shows a small example of an embedded PID controller. The framework allows a method *display* in the operator interface to be called (to display, e.g., the control signal and reference value) upon each iteration of the controller. The controller should be possible to use with a variety of operator interfaces. As an example, an implementation of the operator interface can use *display* to buffer results for use by another task. Regardless of the implementation, we want to bound the WCET of *display* to maintain predictability of the controller.

4 The Skånerost Environment

In Figure 2 an overview of the Skånerost environment is given. It shows the principles of how the tool, the programmer, and the execution environment (the virtual machine) are interconnected.

Three key properties of the environment are shown in this figure:

- *Integration of analysis and compilation.* Both WCET analysis and live memory analysis gain from using information from the compi-

```
class PIDController extends RealTimeThread {
   private double uc, y, u, v;
   private GUI myGUI = null;
   ...
   public synchronized void setGUI(GUI g) { myGUI = g; }
   public synchronized GUI getGUI()       { return myGUI; }
   public void run() {
      long t = currentTime();
      while (true) {
         y  = IO.getY();
         uc = IO.getUc();
         calc_output();
         IO.setU(u);
         update_states();
         GUI g = getGUI();
         if (g != none) g.display(uc, y, u);  // ............ (A)
         t += 100;
         waitUntil(t);
      }
   }
}

class GUI {
   ...
   abstract public void display(double uc,
                                double y,
                                double u)
      /*$ time-bound 25ms */;  // ........................... (B)
}

class BufferedGUI extends GUI {
   ...
   public void display(double uc,
                       double y,
                       double u) {
      // Real-time stuff goes here.  ....................... (C)
   }
}
```

Figure 1 Expressing WCET bounds for the operator interface in a PID controller. The call at (A) has a bounded WCET, since the top-level declaration of the called method at (B) has a WCET bound associated with it. The implementation at (C) must adhere to this bound.

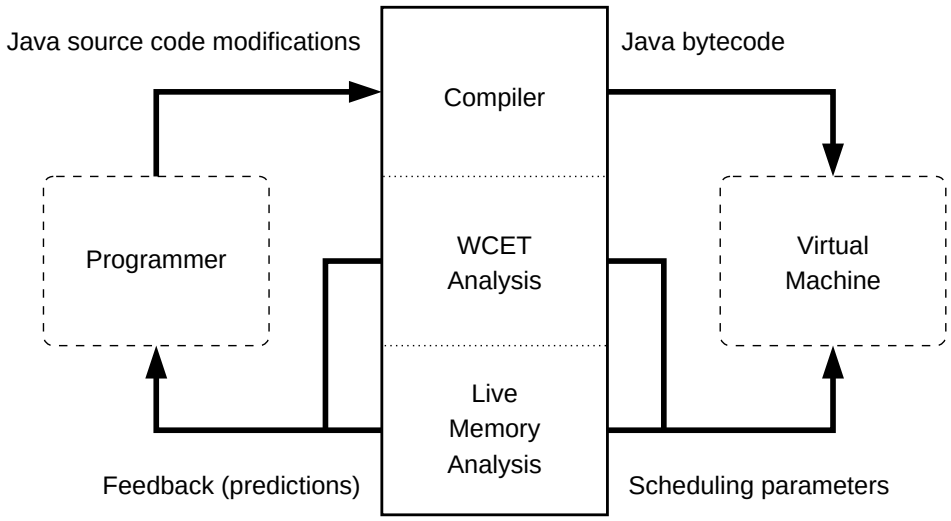


Figure 2 Overall design of Skånerost.

lation, such as name/type information and generated code. This integration thus reduces the complexity of the analysis implementations.

- *Continuous feedback to the programmer.*
- *Compilation to bytecode.*

Skånerost is based on the APPLAB language tool [3, 4]. We use APPLAB's specification language for implementation of all aspects of compilation and analysis: semantic analysis, code generation, WCET analysis, and live memory analysis. The specification language uses Reference Attributed Grammars (RAGs) [7], a special kind of attribute grammars. RAGs allow attributes to be references to syntax nodes, making it easy to express non-local dependencies (such as those between declarations and uses of variables). Non-local dependencies are especially complex to handle in object-oriented languages, where they do not necessarily follow the block structure of programs.

The syntax-directed editor in APPLAB allows an arbitrary attribute (as defined in an attribute grammar) to be inspected by the programmer at any time during development. The WCET of a part of the developed code (such as a loop or a method) is represented by an attribute in the corresponding syntax node. The amount of live memory possibly referenced by a particular declaration in the program is represented in the same way.

We will now outline the internals of the components of Skånerost: compilation, WCET analysis, and live memory analysis.

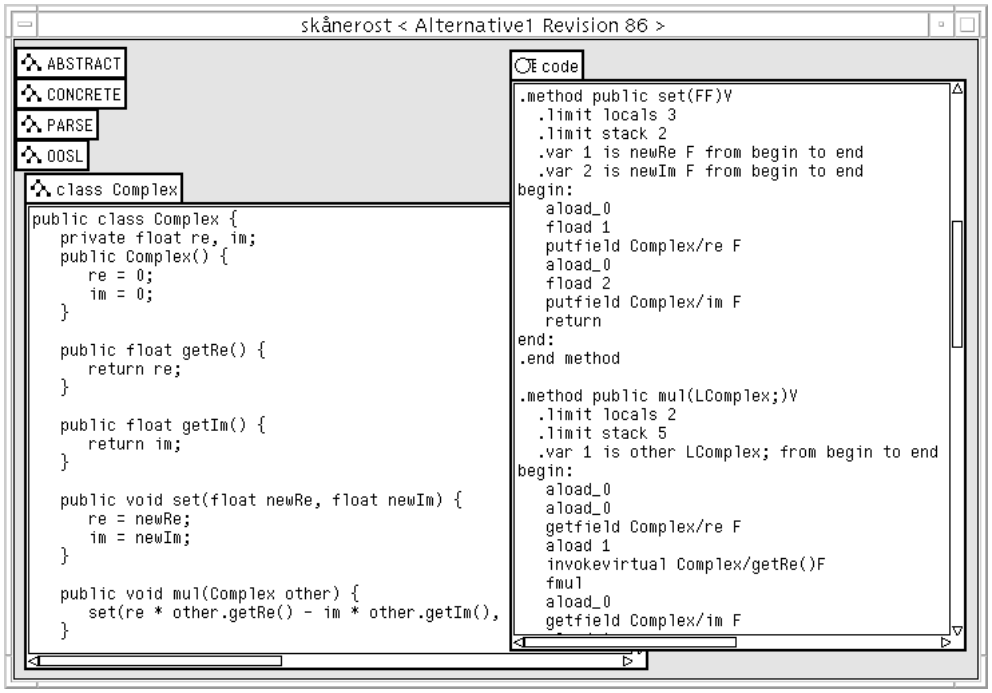


Figure 3 Source code (left) and the corresponding compiled bytecode (right).

4.1 Compilation to Bytecode

Java source code is compiled to an internal representation of Java bytecode. This internal representation is used both for additional analyses and unparsing to a text representation for use by the Jasmin bytecode assembler [15].

One such additional analysis of the internal bytecode representation is used to determine the required stack depth of each method. Since the Java virtual machine is stack-based and an operand stack is allocated in each method frame, the stack depth requirements for each method must be specified in the compiled bytecode. This information is computed by an analysis of the internal bytecode representation.

An example showing Skånerost containing an edited Java class and the compiled bytecode is given in Figure 3.

4.2 Worst-Case Execution Time Analysis

The WCET analysis uses the internal bytecode representation just mentioned. It also uses information from the source code, such as annotations from the programmer. In Figure 4, the source code of a polynomial

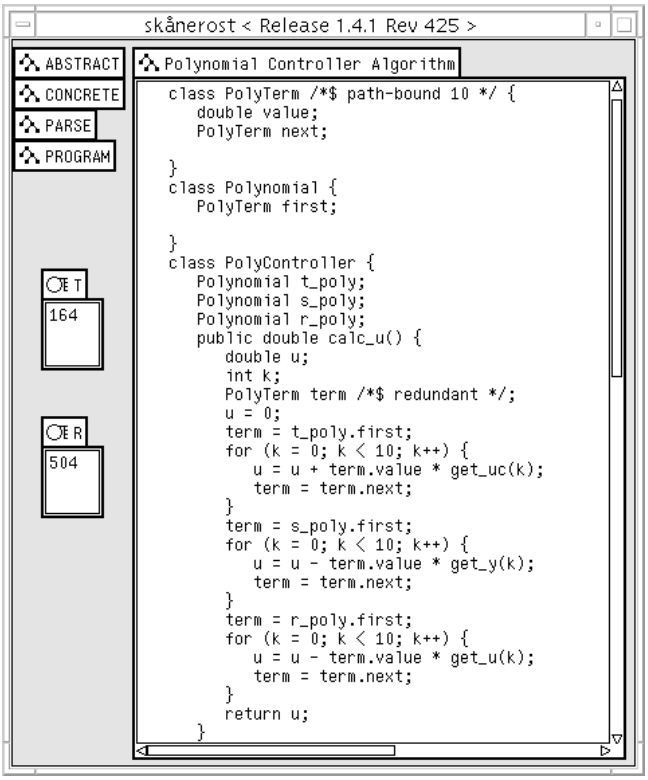


Figure 4 Source code of a polynomial controller and predictions for that code. The *T* window shows the worst-case execution time of the *calc_u* method, and the *R* window shows the amount of memory referenced from a *PolyController* instance.

controller is shown. The output of the controller is computed from the equation

$$R(q)u(k) = T(q)u_c(k) - S(q)y(k)$$

where $u(k)$ is the output at (discrete) time k, $u_c(k)$ is the command signal, y is the measured process output, and $R(q)$, $S(q)$, and $T(q)$ are polynomials in the forward-shift operator [1, Eq. 5.2].

The predicted worst-case execution time of the *calc_u* method is shown in the *T* window. (The exact value depends on the virtual machine timing model mentioned in Section 2.2.) This prediction can be used both as a scheduling parameter and as feedback (regarding the program's ability to meet its deadlines) to the programmer.

4.3 Live Memory Analysis

As discussed in Section 3.2, recursively defined data structures require special treatment in live memory analysis. The class *PolyTerm* in Figure 4 has an annotation (`/*$ path-bound 10 */`), indicating that at most 10 instances of that class appear in sequence (that is, each polynomial in this controller has at most 10 terms). Without this information, the amount of live memory cannot be bounded since the *PolyTerm* class is recursively defined. However, the programmer's domain knowledge about the control algorithm, expressed as an annotation, makes an accurate analysis possible.

The `/*$ redundant */` annotation on the local reference *term* in the *calc_u* method indicates that any object referenced by *term* is always referenced by another, non-redundant reference. Hence, *term* does not contribute to the amount of live memory.

The tools interactively provides worst-case predictions of the memory demands of the program at hand. Analogously to WCET analysis, the obtained prediction can be used both as a scheduling parameter (to compute the amount of execution time required for garbage collection; the exact form of this computation depends on the garbage collector) and as programmer feedback. However, the use of live memory predictions is not restricted to scheduling of garbage collection. It is often important to know how much memory a program requires, particularly in cost-sensitive embedded systems where memory is scarce.

5 Conclusions

The key property of a real-time system is its ability to perform its computations within deadlines. Should such a system turn out to be unable to keep its deadlines, the design or the requirements (or both) must be reassessed. Such changes become more and more expensive as development progresses, and it is thus imperative that such schedulability problems are discovered as soon as possible.

It is highly desirable that the timing properties of the developed program can be continuously observed, and the interactive nature of the tool we have presented allows just that. It provides interactive predictions of time and memory bounds for selected parts of the developed program, along with the generated code.

The target language is Java, a language well suited for embedded real-time systems programming, with respect to both the source representation (a contemporary type-safe object-oriented language) and the portable, dynamically loadable bytecode.

This tool is novel in that these predictions are available directly in the development environment throughout development. It is also novel in its support for a modern, statically typed, object-oriented language with garbage collection and dynamic loading of code.

5.1 Future Work

The present work is related to the development of the Infinitesimal Virtual Machine (IVM), a real-time Java virtual machine designed for a very small memory footprint (tens of kilobytes). The IVM is currently in development at the Department of Computer Science, Lund University. Future plans include development of a timing model for the IVM on a small embedded system, and possibly extending our work to handling more elaborate timing models.

Work on the Skånerost tool continues to support a larger Java subset and integrate analyses further.

Acknowledgments

Boris Magnusson, Klas Nilsson, Anders Ive, Roger Henriksson, and the anonymous reviewers provided valuable comments and suggestions. Elizabeth Bjarnason implemented large parts of APPLAB.

The work of Patrik Persson was financed by ARTES (A network for Real-Time research and graduate Education in Sweden) and SSF (the Swedish Foundation for Strategic Research). The work of Görel Hedin was partially financed by NUTEK (Swedish National Board for Industrial and Technical Development).

References

[1] K. J. Åström and B. Wittenmark. *Computer-Controlled Systems — Theory and Design (Third Edition)*. Prentice-Hall, 1996.

[2] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. To appear in *Proceedings of the 12th EuroMicro Conference on Real-Time Systems*, Stockholm, June 2000.

[3] E. Bjarnason. *Interactive Tool Support for Domain-Specific Languages*. Licentiate Thesis, Department of Computer Science, Lund University, December 1997.

[4] E. Bjarnason, G. Hedin, and K. Nilsson. Interactive Language Development for Embedded Systems. *Nordic Journal of Computing*, Vol. 6, pages 36-55, 1999.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Language Specification*. Addison-Wesley, 1996.

[6] J. Gustafsson. *Analyzing Execution Time of Object-Oriented Programs with Abstract Interpretation.* PhD Thesis, Department of Computer Systems, Information Technology, Uppsala University, Sweden, May 2000.

[7] G. Hedin. Reference Attributed Grammars. *Proceedings of WAGA'99: Second Workshop on Attribute Grammars and their Applications*, pages 153-172. Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.

[8] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*, San Francisco, California, June 1992.

[9] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems.* PhD Thesis, Department of Computer Science, Lund University, September 1998.

[10] L. Ko, N. Al-Yaqoubi, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M. Harmon. Timing Constraint Specification and Analysis. *Software — Practice & Experience*, January 1999.

[11] Y.-T. S. Li, S. Malik, A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'95)*, December 1995.

[12] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An Accurate Worst Case Timing Analysis Technique for RISC Processors. *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'96)*, December 1996.

[13] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM* 20(1), 1973.

[14] T. Lundqvist and P. Stenström. Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques. *Proceedings of*

ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98), Montreal, Canada, June 1998.

[15] J. Meyer. The Jasmin bytecode assembler. *http://mrl.nyu.edu/meyer/jvm/jasmin.html*

[16] Newmonics, Inc. *http://www.newmonics.com*

[17] K. Nilsen. Reliable Real-Time Garbage Collection of C++. *Computing Systems*, 1994.

[18] K. Nilsson, A. Blomdell, and O. Laurin. Open Embedded Control. *Real-Time Systems*, Vol. 14, No. 3, 1998.

[19] P. Persson. Live Memory Analysis for Garbage Collection in Embedded Systems. *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*, Atlanta, Georgia, May 1999. ACM SIGPLAN Notices 34(7), pages 45-54.

[20] P. Persson and G. Hedin. Interactive Execution Time Predictions using Reference Attributed Grammars. *Proceedings of WAGA'99: Second Workshop on Attribute Grammars and their Applications*, pages 173-184. Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.

[21] A. C. Shaw. Reasoning about Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, Vol. 15, No. 7, 1989.

[22] Real-Time Specification for Java™ Experts Group. *http://www.rtj.org*

[23] Sun Microsystems. Jini™ Connection Technology. *http://www.sun.com/jini/*

[24] V. Sundaresan, C. Razafimahefa, R. Vallée-Rai, and L. Hendren. Practical Virtual Method Call Resolution for Java. Sable Technical Report No. 1998-7, McGill University, Canada, 1998.

[25] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing Analysis for Data Caches and Set-Associative Caches. *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, June 1997.