# Deriving Java Virtual Machine Timing Models for Portable Worst-Case Execution Time Analysis

Erik Yu-Shing Hu, Andy Wellings and Guillem Bernat

Real-Time Systems Research Group
Department of Computer Science
University of York, York, YO105DD, UK
{erik,andy,bernat}@cs.york.ac.uk

**Abstract.** Performing worst-case execution time (WCET) analysis on the highly portable real-time Java architectures without resulting in the under utilisation of the overall system has several challenges. Current WCET approaches are tied to either a particular language or target architecture. It should also be stressed that most WCET analysis approaches are usually only considered in relation to procedural programming languages. In this paper, we propose a comprehensive portable WCET analysis approach, and demonstrate how Java virtual machine timing models can be derived effectively on real-time and embedded Java-based systems.

**Keywords :** Real-Time Java, Worst-Case Execution Time (WCET) Analysis, Portable WCET

## 1 Introduction

To be able to support a predictable and expressive real-time Java environment, two major international efforts have attempted to provide real-time extensions to Java: the Real-Time Specification for Java (RTSJ) [4] and the Real-Time Core extensions to Java [6]. These specifications have addressed the issues related to using Java in a real-time context, including scheduling support, memory management issues, interaction between non-real-time Java and real-time Java programs, and device management, among others.

On the whole, timing analysis is crucial in real-time systems to guarantee that all hard real-time threads will meet their deadlines in line with the design. In order to ensure this, appropriate scheduling algorithms and schedulability analysis are required. Typically, most scheduling algorithms assume that the *Worst-Case Execution Time* (WCET) estimation of each thread has to be known prior to conducting the schedulability analysis. Therefore, estimating WCET bounds of real-time threads is of vital importance. Unfortunately, in neither of the real-time Java specifications [4, 6], is there a satisfactory solution to how WCET estimations can be carried out on the Java architecture with portability in mind.

For the most part, current WCET approaches [3, 5, 14, 16, 17] are tied to either a particular language or target architecture. In addition, these analysis approaches are only considered in relation to procedural programming languages. Performing WCET analysis on the Java architecture must take into account not only additional dynamic features, such as dynamic dispatching and memory management, but also the platform independent issue. Therefore, most WCET analysis approaches are not appropriate for the Java architecture, since Java programs are "write once, run every where" or perhaps more appropriate for real-time "write once carefully, run everywhere conditionally" [4].

To our knowledge, only the portable WCET analysis proposed by Bernat et al. [2, 1] has taken account of platform independent features for the Java architecture. Portable WCET analysis uses a three-step approach: high-level analysis (i.e. analysing the Java programs), low-level analysis (i.e. producing a *Virtual Machine Time Mode*[1] (VMTM) for a target platform), and conducting a combination of the high-level analysis with the low-level analysis to compute the actual WCET bound of the analysed code sections. However, the portable WCET analysis approach has only tended to focus on supporting portability, rather than addressing the dynamic features of Java. It also should be noted that portable WCET analysis highly depends on the VMTM of a target platform, and there is also no appropriate solution to show how a VMTM for a particular platform can be built efficiently. Therefore, from a practical standpoint, bringing this approach into engineering practice still has a number of issues to be addressed.

For the above reasons, we introduce a comprehensive portable WCET analysis that takes into account the dynamic dispatching issues [7, 10, 11] and presents how VMTMs can be built for various target platforms. The major contributions of this paper are:

- introducing two measurement approaches that demonstrate how to extract Java VMTMs for portable WCET analysis,
- discussing how VMTM can be derived by a profiling-based approach, and
- presenting how to build a portable benchmark model to extract VMTMs from various target platforms.

The rest of the paper is organised as follows. Section 2 gives a summary of the related work, while Section 3 presents an overview of the comprehensive portable WCET analysis. Section 4 discusses how Java VMTMs can be extracted from various platforms with two different approaches. Following this, Section 5 gives a simple example to evaluate our approaches. Finally, conclusions and future work are presented in Section 6.

## 2 Related Work

In general, there are two principal ways for obtaining the WCET of a program: static analysis and dynamic analysis (a.k.a. measurement approach). Most systems in industry have relied on ad-hoc measurements of execution times when

[1] VMTM is a timing model for the target virtual machine including a list of the worst-case execution time of native methods and Java bytecode instructions.

designing real-time systems [15]. Arguably, measuring an execution time could be an unsafe practice, since one cannot know whether the worst case has been captured in the measurements.

In contrast, a static analysis could give relatively safer results for the WCET analysis [18]. A number of research approaches [5, 14, 16, 19] have demonstrated how to estimate WCET at high-level and low-level analyses. Unfortunately, the above WCET approaches are tied to either a particular language or target architecture. In addition, these analysis approaches are only considered in relation to procedural programming languages.

As processors have tended to be more complex recently, some research approaches [3, 15, 17] have integrated measurement techniques with static analysis to address modern complicated processor issues. However, these approaches have attempted to estimate WCET bounds from applications to the target platform at once. As a result, these techniques cannot take advantage of the platform independent feature supported in Java.

Notable exceptions include the portable WCET analysis proposed by Bernat et al. [2, 1]. This approach has taken into account platform independent features for the Java architecture. However, they have only tended to focus on supporting portability, rather than taking account of the issues connected with the use of dynamic dispatching features. Nor is there an appropriate solution to show how a VMTM for a particular platform can be built efficiently.

Our approach extends the portable WCET analysis approach to take into account dynamic dispatching issues and provide a portable model to build VMTM effectively on real-time and embedded systems. Arguably, there is some additional pessimism in performing the WCET process in this particular way, which counteracts for the added benefits that portability brings [2]. It can be observed that this pessimism can be compensated with the use of gain time reclaiming mechanisms [10, 11] integrated in our approach.

## 3 Overview of the Framework

Following the philosophy of portable WCET analysis [2, 1], our framework (Figure 1), therefore, also uses the three-step approach to be able to offer a comprehensive WCET analysis bearing portability and dynamic dispatching issues in mind.

Note that this framework is part of our on-going work, called XRTJ [2] [9], which extends the current Real-Time Java architecture [4] proposed by the Real-Time Java Expert Group. The XRTJ environment is targeted at cluster-based distributed high-integrity real-time Java systems, such as consumer electronics and embedded devices, industrial automation, space shuttles, nuclear power plants and medical instruments. In the XRTJ environment, to facilitate the various static analysis approaches and provide information that cannot be expressed in either Java source programs or Java bytecode, an extensible and

[2] XRTJ: an Extensible Distributed High-Integrity Real-Time Java Environment

portable annotation class format called *Extensible Annotations Class* (XAC) file is proposed [8]. To generate XAC files, an annotation-aware compiler, named *XRTJ-Compiler* [9], which can derive additional information from either manual annotations or source programs, or both, is also introduced. Furthermore, a static analyser, called *XRTJ-Analyser* [9], is introduced in order to support various static analyses, including program safety analysis and timing analysis.

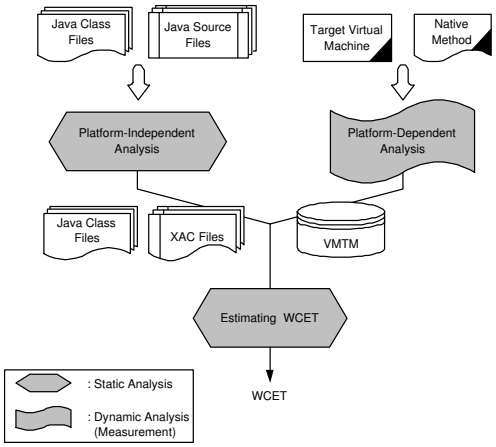


**Fig. 1.** The comprehensive portable WCET framework

The first step of the framework is the *platform-independent* analysis. At this stage, the technique analyses annotated Java programs or Java class files to produce portable WCET information. Manual annotations in our approach are introduced with the characters '//@' for single line and '/*@ ... @*/' for multiple lines [8]. Note that these formats are assumed to be comments in Java. Taking advantage of the knowledge accumulated with the compiler, portable WCET information can be extracted from either source programs or Java bytecode statically. Here, dynamic dispatching methods can also be analysed with our previous approaches [7, 10, 11]. Portable WCET information is computed in the form of so-called *Worst-Case Execution Frequency* (WCEF) vectors by the XRTJ-compiler. WCEF vectors [1] represent execution-frequency information about basic blocks[3] and more complex code structures that have been collapsed during the first part of the analysis. Then portable WCET information can be stored into the XAC files [8]. Note that the static analysis is used in this stage.

In parallel, analysis of the target platform is performed, so-called *platform-dependent* analysis. This takes the form of the definition of a timing model of the virtual machine. This stage performs platform-dependent analysis (i.e. in the context of specific hardware and VM) of the implementation of Java

[3] A basic block is a continuous section of code in the sense that control flow only goes in at the first instruction and leaves through the last one.

bytecode instructions. During this stage, information about the potential effects of pipelines [1] and caches[4] may be captured.

Although the platform-independent analysis can be carried out by a static analysis approach, the use of a static analysis technique to perform the platform-dependent analysis has a number of challenges. It should be noted that when deriving VMTM it is necessary to take into account the implementation aspects of not only the Java virtual machine, but also the operating system. In addition, the analysis should also bear in mind the hardware architecture for various embedded systems. In order to accommodate a diverse set of implementations on the underlying platforms and virtual machines for embedded systems the measurement-based analysis technique is used in our approach. This will be explored further in Section 4.

The final stage is the estimating of the WCET bounds of each thread. In the XRTJ environment, a WCET analysis tool in the XRTJ-Analyser performs the combination of the results of platform-independent analysis with the target VMTM to compute the actual WCET bound of the analysed code sections.

## 4 Deriving Java Virtual Machine Models

Deriving the VMTM of a target platform is crucial in the portable WCET analysis since the results of the analysis are highly dependent on the outcome of the VMTM. Arguably, in the real-time and embedded field, analysing a virtual machine to produce the VMTM of the target platform needs to be efficient and rapid since the development life-cycles of the software built for embedded systems are short and the applications are demanded to be reusable and compatible among various architectures. Therefore, how to efficiently derive VMTMs for different platforms is the key issue for the portable WCET analysis approach. In this paper, we propose two measurement approaches: *profiling-based analysis* and *benchmark-based analysis*; which demonstrate how the VMTM can be extracted from a target platform.

Note that there are several possible ways in which the execution time can be measured, such as using *clock cycle counters* and using *timers*. In our approach, we use the *rdtsc* instruction, which has high resolution and very low overhead at run-time, provided in x86 architecture [12] to extract the time-stamp counter of the processor. Although we only show the use of software approach on the x86 architecture under the Linux platform here, our approach can also be applied to other CPU architectures and operating systems if they support instructions or libraries that can extract the time-stamp counter of the processor. For example, *getrtime*() library routine can be used on the SPARC V9 architecture under Solaris 8 operating systems, and hardware data acquisition interfaces[5] can be used under Windows, Linux and Solaris operating systems.

[4] The analysis of the cache effects is our future work and beyond the scope of this paper.

[5] `http://www.ultraviewcorp.com/`

### 4.1 Profiling-Based Analysis

Observing the behaviour of a system to analyse the specific aspects of applications executing on the system is not novel. An automatic tracing analysis [20] has been proposed to extract temporal properties of applications and operation systems. The approach shows that the empirical analysis can reduce the over-estimation of real-time applications. Accordingly, a profiling-based analysis technique can be applied to deriving a VMTM for a particular platform by instrumenting additional code into the virtual machine.

Even though the idea is relatively straightforward to derive a VMTM, a number of issues need to be addressed to ensure the reliability of the derived VMTM. For example:

- where to insert the instrumenting code,
- how to minimise the side effects of the instrumenting code at run-time, and
- how to avoid the out-of-order execution during the measurement of the specific code section.

Similar to the automatic tracing analysis approach [20], profiling the execution time of each bytecode can be divided into two steps. One is extracting run-time information and the other is analysing it. The former step involves: exploring the context of the virtual machine where temporal information can be derived; and the instrumenting code to extract the time-stamp counter of the processor with very low runtime overhead. For instance, the instrumenting code has to accumulate the instruction mnemonics and the time-stamp counter every time the interpreter fetches a bytecode. The latter step analyses these data and builds up a VMTM for the target platform.

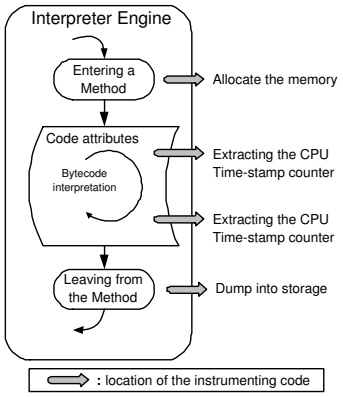


**Fig. 2.** Instrumenting profiling code into an interpreter engine

To be able to trace the run-time information, instrumenting code needs to be provided into the Java virtual machine. The instrumenting mainly depends

on the specific implementation of the JVM. However, on the whole, Java virtual machines conduct the interpretation of Java bytecode by a method-based approach. Therefore, to reduce the memory and run-time overhead needed for collecting the run-time information, the implementation of the profiling-based analysis can refer to a suggested implementation, given in Figure 2. Note that the major aim of the collecting run-time data by method in the interpreter engine is only to reduce the memory and run-time overhead of the instrumenting code, rather than analysing the applications.

As shown in the figure, a small amount of memory, which can be allocated when invoking a method, is necessary to store the collected information during run-time. These accumulated data can be dumped into storages when returning from the method (i.e. finishing the interpretation of the method). Dumping accumulated data at this point can reduce the noise or side effects of the instrumenting code on the measurement results. Here, these data can be analysed with the requirements of the target platform and the VMTM can be built with these analytical data. To avoid out-of-order execution during profiling, a serializing instruction (*cpuid*) can be invoked before extracting the time-stamp counter of the processor.

The experimental implementation of this approach has been carried out on the reference implementations of RTSJ provided by TimeSys [21]. Basically, the instrumenting code, including the serializing and time-stamp counter instructions, is added into the interpreter engine. Before starting the interpretation of a method, a buffer to store run-time information is prepared. Then, the execution time of each bytecode can be measured starting from the opcode fetched to before fetching the next opcode. The run-time information captured by the interpreter is classified by the opcode mnemonics. When leaving the method, the captured run-time information can be conducted with statistical analysis to produce he VMTM. The evaluation of this approach is discussed in Section 5.

### 4.2 Benchmark-Based Analysis

It should be noted that the analysis of the portable WCET analysis approach highly depends on the VMTM of a target platform, and the technique provided in the previous section needs enormous effort to be carried out, including modifications to the execution engine of the target Java virtual machine to derive the execution time of each bytecode. In order to conduct this, it is clear that the source of the virtual machine is necessary. Although deriving the execution time of a single bytecode can be achieved by the previous mechanism, deriving the execution of specific sets of bytecodes is unlikely to be accomplished. Furthermore, the implementations of the previous approach cannot be reused for building the VMTM of a new virtual machine. This means that creating a VMTM for a new virtual machine needs to be started from scratch. Therefore, to be able to apply portable WCET analysis to real-time and embedded systems effectively, two major issues need to be addressed:

– how the instrumenting code can be reused effectively on various platforms without modifying it, and

– how the execution time of a specific set of bytecodes can be measured.

To address these issues, the *benchmark-based analysis* approach is introduced. The aim of this approach is to provide a Java-based benchmark[6] that may produce a VMTM automatically after executing it on the target virtual machine. The principle behind this mechanism is to inject individual or a set of specific bytecodes into the instrumenting code developed in a native method that may access the time-stamp counter of the processor in a Java program. Therefore, the native method using Java native interface (JNI) features in the benchmark can be ported easily to different platforms without modifying the benchmark. However, some issues need to be addressed to achieve these goals:

– where and how specific bytecodes can be inserted into the Java program to measure the execution time of the specific bytecodes, and
– how to maintain the integrity of the Java stack after the injection of additional bytecodes.

To prove the feasibility of this approach and reduce the time needed to develop the whole mechanism, a number of tools have been investigated. Taking advantage of the time-stamp counter instruction (*rdtsc*) [12] supported in x86 architecture, the bytecodes *disassembler* and *assembler* tools provided in the *Kopi Compiler Suite* [13], and the Java native interface feature, the benchmark-based analysis approach can be carried out. The procedure of how the benchmark can be established is given below.

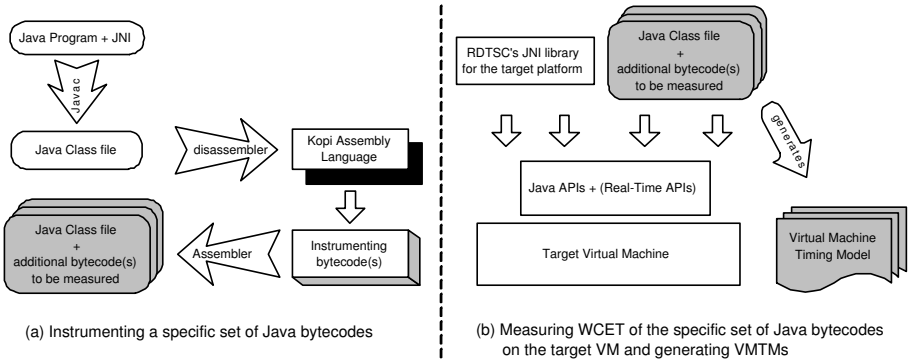


**Fig. 3.** Instrumenting and Measuring of the benchmark-based approach

As shown in Figure 3, a Java program with a native method that can access the time-stamp counter can be translated into Java bytecode by a traditional compiler. Then, the class file can be translated into Kopi assembly language to

[6] The term *benchmark* means a collection of Java programs that are instrumented with particular bytecodes to be measured.

be able to insert a specific set of Java bytecodes easily under text mode. Here, one should note that the integrity of the Java stack of JVM needs to be borne in mind when inserting additional bytecodes. For instance, after executing the *iload* instruction, the virtual machine will load an integer onto the Java stack. Therefore, we need to add complementary bytecodes to remove the integer from the stack in order to maintain the data integrity of the Java stack for the whole program. Some bytecodes may also need to be provided with values or references before executing them, such as *iadd* and *iaload*. As a result, to ensure the data integrity of the Java stack, corresponding complementary bytecodes needed to be added at the *pre-* or *post-* locations of the measurement bytecodes. After injecting the specific bytecodes, the file saved in the Kopi assembly language format can be translated into standard Java class files. As presented in Figure 3, these individual instrumenting Java programs can be combined together into a comprehensive benchmark that can generate VMTM automatically. Then, the individual Java program or the benchmark is ready to be used for measuring the execution time of the specific set of bytecodes on any target platform.

One should note that the major purpose of the benchmark is to produce VMTM that contains a collections of the WCET bounds of WCEF vectors and method calls. Therefore, it is possible that building VMTM to be used in systems in the small can provide a compact benchmark that comprises those WCEF vectors that will only be used on such systems. The benchmark can be executed on any particular target platform with a native method that can access the time-stamp counter of the target platform. This approach can be used to generate the execution time of a specific set of common sequence bytecodes since it allows to insert any combination of bytecodes with this mechanism. It can be observed that generating instrumented Java programs can be automatically conducted by a simple program implementing the above procedure.

**Table 1.** Measurements of the WCET of the instrumenting code

| Experiment | 99.95% | 99.90% | Average |
|---|---|---|---|
| 1. | 334 | 321 | 320 |
| 2. | 326 | 321 | 321 |

**Table 2.** Measurements of the WCET of `iload` with the instrumenting cost

| Experiment | 99.95% | 99.90% | Average |
|---|---|---|---|
| 1. | 366 | 353 | 353 |
| 2. | 363 | 353 | 353 |

An experiment has been carried out on the RTSJ-RI and the preliminary result of the analysis of *iload* bytecode is given below. A Java program developed with a native method that can access the time-stamp counter of the processor is developed to measure the cost of the instrumenting code. The experimental results of the cost of the instrumenting code is given in Table 1. Then, *iload* bytecode instructions is added into the instrumenting code and the corresponding complementary bytecode (i.e. *istore*) is inserted at the post instrumenting code. The measurement of the *iload* has been carried out in a 50000-times loop. The experiment has been conducted several times and two of them are given in Table 2. There is a graph illustrating that the distributions of the measurement

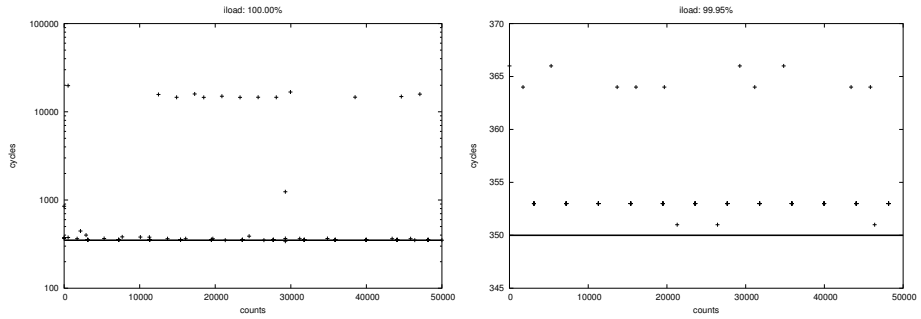


**Fig. 4.** Measurements of the *iload* bytecode with the benchmark-based analysis

of the *iload* instruction is given Figure 4. The *iload:100%* shows the machine cycles of all the measurements in the loop, whereas the *iload:99.95%* presents 99.95 percentage of the measurements of the loop where we assumed that very high execution time of the measurements are influenced by the operation systems and interrupts.

## 5 Evaluation

The evaluation of our analysis is illustrated with an example code of the Bubble Sort algorithm presented in Figure 5. Figure 6 shows the individual basic block of the algorithm with offset numbers. The maximum number of iterations of the outer and inner loops can be assumed as $10 - 1$ and $10(10 - 1)/2$ respectively when the *size* is equal to 10.

The WCEF vectors of the bubble sort algorithm, generated by our prototype compiler during compilation, is given in Figure 7 in text mode. In this example, only 14 different Java bytecodes are generated by the Java compiler. When deriving VMTM it is necessary to minimise the run-time overhead and influences

```
import javax.realtime.*;
public class rth extends RealtimeThread {
  static int Data[];
  public void run() {
    // ...
    bb_Sort(Data);
    waitForNextPeriod();
  }
  public static void bb_Sort(int a[]) {
    int i, j, t;
    int size = 10;
    for(i=size-1; i>0; i--) {
      for(j=1; j<=i; j++) {
        if(a[j-1] > a[j]) {
          t = a[j-1];
          a[j-1] = a[j];
          a[j] = t;
        }
      }
    }
  }
  ...
}
```

**Fig. 5.** The Bubble Sort Algorithm in Java

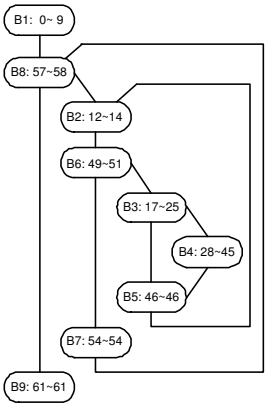


**Fig. 6.** Individual basic blocks with their offset numbers

```
<WCEFVectors>
  ...
  <Method=bb_Sort:([I)V>
    <SubTAG_BODY>
      BB:(0~9)
        bipush:1
        istore:1
        iload:1
        iconst_1:1
        isub:1
        istore_1:1
        goto:1
      BB:(12~14)
        iconst_1:1
        istore_2:1
        goto:1
      BB:(17~25)
        aload_0:2
        iload_2:2
        iconst_1:1
        isub:1
        iaload:2
        if_icmple:1
      BB:(28~45)
        aload_0:4
        iload_2:4
        iconst_1:2
        isub:2
        iaload:2
        istore_3:1
        iastore:2
        iload_3:1
      BB:(46~46)
        iinc:1
      BB:(49~51)
        iload_2:1
        iload_1:1
        if_icmple:1
      BB:(54~54)
        iinc:1
      BB:(57~58)
        iload_1:1
        ifgt:1
      BB:(61~61)
        return:1
    </SubTAG_BODY>
  ...
</WCEFVectors>
```

**Fig. 7.** WCEF Vectors of the bubble sort algorithm in text mode

of background process running in the operating systems including background tasks and interrupters. We addressed these issues by running the test-bed under single user mode on Linux. In addition, other background processes are killed manually to reduce the influences as much as possible. The measurements of the execution time are represented with machine cycle unit in the rest of this example.

**Table 3.** A VMTM derived with the benchmark-based analysis

| Bytecode | 99.95% | 99.90% | Average |
|---|---|---|---|
| aload | 48 | 38 | 36 |
| bipush | 40 | 30 | 30 |
| iaload | 46 | 34 | 36 |
| iastore | 55 | 41 | 31 |
| ifgt | 67 | 47 | 27 |
| if_icmple | 71 | 51 | 52 |
| iinc | 93 | 64 | 62 |
| iload | 37 | 32 | 24 |
| istore | 50 | 38 | 36 |
| isub | 45 | 34 | 35 |
| goto | 36 | 27 | 25 |
| iconst0 | 38 | 35 | 31 |
| iconst1 | 37 | 37 | 32 |

A summary of the VMTM for the Bubble Sort example is shown in Table 3. This table shows the different statistical analysis results of the VMTM carried out with benchmark-based analysis. Each bytecode is measured by 50000 times continuously. As shown in Figure 8, although the VMTM derived with the benchmark-based approach shows rather constant outcomes, the VMTM produced with the profiling-based approach presents relatively pessimistic results if the 99% of the measurements have been taken into account as the WCET

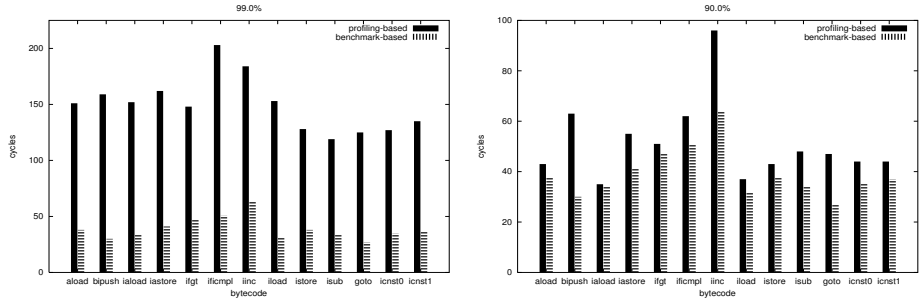


**Fig. 8.** Comparing the profiling-based and benchmark-based analyses

bounds. This can be reasoned that the ad-hoc measurements of the profiling-based analysis can produce pessimism because it derives the execution time of each bytecode from various methods invoked on the VM and most methods are invoked during the initialisation phase of the VM. As a result, some measurements could be the worst-case response time of the bytecode instead of the WCET bounds. However, it can be observed that the 90% percent of the measurements of the profiling-based analysis are very close to the results derived with the benchmark-based analysis. The experiment also shows that the profiling-based analysis has some difficulties to control which particular bytecodes to be measured and the number of the measurements of the bytecodes. Therefore, in order to obtain the reliable measurements with the profiling approach, it also needs to be provided with a large amount of the particular bytecodes needed to be measured.

Using Table 3, three different WCET bounds (i.e. 99.95%, 99.90%, and average) can be estimated. The WCEF of the bubble sort algorithm is obtained as follows:

WCET( bb_Sort() )
= B1+10∗B8+9(B2+B7)+46∗B6+45(B3+B4+B5)+B9

**Table 4.** Comparing the final WCET bounds

| Approach | 99.9% | 90.0% |
|---|---|---|
| End-to-end measurement | 40378 | 39865 |
| Benchmark-based | 42125 | 39908 |
| Profiling-based | 164689 | 51235 |

The final WCET bounds of the algorithm with different approaches (i.e. end-to-end measurement, benchmark-based analysis and profiling-based analysis) have been conducted. The estimations taking account of the 99% and 90% of the measurements in Table 4. Note that the method of estimating the pipeline effects is beyond the scope this paper and the technique proposed in [1] can be integrated into our approach easily with benchmark-based analysis.

## 6 Conclusion and Future Work

Since the aim of portable code is to support hardware interchangeability, the WCET analysis for such portable applications needs to bear portability in mind. The comprehensive portable WCET has been proposed with a three stage approach to analyse the highly portable and reusable Java applications for real-time and embedded systems. In this paper, we have mainly discussed how to derive various VMTMs to facilitate the use of portable WCET [2] in real-time and embedded Java-based applications.

Two approaches (i.e. profiling-based and benchmark-based) have been proposed to derive VMTMs. The major advantage of the profiling-based approach can be extended to integrate with other tracing or profiling techniques, such as POSIX-trace [20], whereas the disadvantages of the approach are that it needs the source code and knowledge of the target VM and it takes time to instrument the additional code into the VM. In contrast, the benchmark-based analysis is highly portable and only needs to provide a native method to access the time-stamp counter of the target processor. However, the benchmark-based analysis is less convenient to integrate with other profiling techniques. Therefore, these techniques can be applied to various applications that depend on the requirements of the systems.

Based on the experimental results, the outcomes of the benchmark-based analysis approach (Figure 8) encourage us to carry on the future work on the use of portable WCET analysis in real-time and embedded Java-based systems, whereas the results of the profiling-based analysis approach reminds that taking account of other run-time issues, such as cache effects and branch prediction issues, can achieve relatively safer and tighter WCET estimations.

There are still a number of issues that need to be addressed in our approach, such as taking into account the timing properties of the RTSJ, cache effects on WCET estimations and extending for just-in-time compiler techniques.

## References

1. I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-Level Analysis of a Portable Java Byte Code WCET Analysis Framework. *Proceedings of the 6th IEEE Real-Time Computing Systems and Applications RTCSA-2000*, pages 39–48, December 2000.
2. G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. *Proceedings of the 6th Euromicro Conference on Real-Time Systems*, pages 81–88, June 2000.
3. G. Bernat, A. Colin, and S. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, December 2002.
4. G. Bollella, J. Gosling, B. M. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull. *Real-Time Specification for Java*. Addison Wesley, 2000.
5. R. Chapman, A. Burns, and A. Wellings. Integrated Program Proof and Worst-Case Timing Analysis of SPARK Ada. *Proceedings of the Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

6. J. Consortium. Real-Time Core Extensions for Java platform. *International J Consortium Specification*, Revision 1.0.14, September 2000. http://www.j-consortium.org/rtjwg/.
7. E. Y.-S. Hu, G. Bernat, and A. J. Wellings. Addressing Dynamic Dispatching Issues in WCET Analysis for Object-Oriented Hard Real-Time Systems. *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC-2002*, pages 109–116, April 2002.
8. E. Y.-S. Hu, G. Bernat, and A. J. Wellings. A Static Timing Analysis Environment Using Java Architecture for Safety Critical Real-Time Systems. *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems WORDS-2002*, pages 77–84, January 2002.
9. E. Y.-S. Hu, J. Kwon, and A. J. Wellings. XRTJ: An Extensible Distributed High-Integrity Real-Time Java Environment. *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications RTCSA-2003*, pages 371–391, February 2003.
10. E. Y.-S. Hu, A. J. Wellings, and G. Bernat. A Novel Gain Time Reclaiming Framework Integrating WCET Analysis for Object-Oriented Real-Time Systems. *Proceedings of the 2nd International Workshop on Worst-Case Execution Time Analysis WCET-2002*, June 2002.
11. E. Y.-S. Hu, A. J. Wellings, and G. Bernat. Gain Time Reclaiming in High Performance Real-Time Java Systems. *Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC-2003*, pages 249–256, May 2003.
12. Intel's Applications Notes. Using the RDTSC Instruction for Performance Monitoring. Intel. http://cedar.intel.com/software/idap/media/pdf/rdtscpm1.pdf.
13. Kopi. The Kopi Project. DMS Decision Management Systems Gmb. Hhttp://www.dms.at/kopi/.
14. S. Lim, Y. Bae, G. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Kim. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.
15. M. Lindgren. Measurement and Simulation Based Techniques for Real-Time Systems Analysis. Dissertation, Uppsala University, Sweden, 2000.
16. F. Mueller. *Static Cache Simulation and its Applications*. Ph.d thesis, Department of Computer Science, Florida State University, July 1994.
17. S. Petters and G. Farber. Making Worst Case Execution Time Analysis for Hard Real-Time Tasks. *In Proceedings of the 6th International Conference on Real-Time Computing Systems and Application RTCSA-1999*, December 1999.
18. P. Puschner and A. Burns. A Review of Worst-Case Execution-Time Analysis. *Real-Time Systems*, 18(2/3):115–128, 2000.
19. A. Shaw. Reasoning about Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.
20. A. Terrasa and G. Bernat. Extracting Temporal Properties from Real-Time Systems by Automatic Tracing Analysis. *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications RTCSA-2003*, pages 483–502, February 2003.
21. TimeSys. Real-Time Java. TimeSys. http://www.timesys.com/prodserv/java/.