# A Modular Worst-case Execution Time Analysis Tool for Java Processors

Trevor Harmon
Electrical Engineering and Computer Science
University of California, Irvine
tharmon@uci.edu

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

Raimund Kirner
Institute of Computer Engineering
Vienna University of Technology, Austria
raimund@vmars.tuwien.ac.at

Raymond Klefstad
Electrical Engineering and Computer Science
University of California, Irvine
klefstad@uci.edu

## Abstract

*Recent technologies such as the Real-Time Specification for Java promise to bring Java's advantages to real-time systems. While these technologies have made Java more predictable, they lack a crucial element: support for determining the worst-case execution time (WCET). Without knowledge of WCET, the correct temporal behavior of a Java program cannot be guaranteed. Although considerable research has been applied to the* theory *of WCET analysis,* implementations *are much less common, particularly for Java.*

*Recognizing this deficiency, we have created an open-source, extensible tool that supports WCET analysis of Java programs. Designed for flexibility, it is built around a plug-in model that allows features to be incorporated as needed. Users can plug in various processor models, loop bound detectors, and WCET analysis algorithms without having to understand or alter the tool's internals.*

## 1 Introduction

The increasing size and complexity of real-time systems are starting to push developers away from C. Despite its status as the most popular language in real-time computing, it is relatively low-level, error-prone, and difficult to scale to large systems.

An increasing number of developers and researchers are looking to Java as a potential alternative. This much newer language offers direct benefits: Compilers for Java catch many errors that C compilers miss; the language definition specifically addresses safety and security issues; and the high-level nature of Java makes it more productive [2].

At the same time, Java is a terrible match for real-time systems. Its combination of automatic garbage collection, underspecified threading semantics, and pervasive object-orientation (i.e., dynamic dispatch) are all impediments to building time-predictable software.

Recent efforts to address some of these issues have focused on the Real-time Specification for Java (RTSJ) [4]. It provides scoped memory to avoid high-latency garbage collection, tightens the threading model to support real-time scheduling, and adds other features that make Java a viable platform for real-time applications.

The RTSJ is only a partial solution, however. It is designed mainly for soft real-time environments that have significant hardware resources. (The Sun Real-time System,[1] for instance, requires an UltraSparc III or higher with at least 512 MB of RAM.) For embedded applications where processing power and memory are at a premium, the RTSJ is infeasible.

The RTSJ also lacks an essential element necessary for hard real-time and safety-critical systems. Though it may provide deterministic scheduling, priority inversion avoidance, and predictable memory allocation, it offers no support for computing the worst-case execution time (WCET) of a task, and without knowing the WCET, no guarantees can be made on the timeliness of the system. The RTSJ's only provision is in its scheduler interface—the `ReleaseParameters` class—which takes as input the WCET of a schedulable object. But even this basic support may be useless in practice, as evidenced by the documentation note accompanying the class: *Cost measurement and enforcement is an optional facility for implementations of the RTSJ.*

[1] http://java.sun.com/javase/technologies/realtime.jsp

### 1.1 Java Microprocessors

In the last few years, a new approach for dealing with WCET analysis in Java has emerged. Rather than fight the increasingly unpredictable behavior of virtual machines and superscalar processors, an alternative strategy eliminates them entirely. The approach relies on specialized processors that understand Java bytecode as their native instruction set. Examples include the aJ-100 [8], the Cjip [11], and the JOP [19, 20].

These Java-specific processors offer several advantages for real-time systems. Predictability is one of the key factors. For example, Java's jump instructions are guaranteed never to target beyond the address range of the declaring method; therefore, a method-based cache, such as the one employed by JOP [17], makes every non-invocation and non-return instruction a cache hit. These characteristics yield a much tighter bound when performing static WCET analysis. Java processors also eliminate the operating system and virtual machine layers, making analysis and safety certification far simpler.

These qualities make Java processors an attractive platform for hard real-time systems. Although moving to such a novel and unique architecture may seem drastic, developers of real-time systems have a tradition of adopting new platforms when special needs arise, as evidenced by the popularity of ARM and PowerPC architectures in embedded devices.

### 1.2 The Volta Project

Despite the advantages of Java-based processors for real-time systems, tools for WCET analysis in the Java domain are virtually non-existent. Only two implementations have ever been created: Javelin [3] and WCA [21], both of which are now unmaintained. Without working implementations to build upon, testing new theories and finding new avenues of research are exceedingly difficult. The lack of implementations also prevents real-time developers from gaining the productivity advantages offered by Java.

To address these problems, we have constructed a suite of tools for developing and analyzing real-time software on Java processors. The suite, called *Volta*, is completely open-source[2] and implemented entirely in Java. It currently consists of a control flow analyzer called Cascade and a static WCET analyzer called Clepsydra.

In prior work [9], we described the Clepsydra tool and its support for *back-annotation*, a technique that automatically annotates every statement in a real-time Java program with its worst-case execution time. Clepsydra is unique in that it can perform this back-annotation interactively, as the program is being written. The Volta project includes a Clepsydra plug-in for jEdit[3] as a demonstration of how back-annotations can be integrated seamlessly into a real-time programmer's development environment.

In this work, we have extended the tool to support WCET analysis across method invocations. The support includes an enhanced control flow analyzer with improved visualization of control flow graphs, as well as a pluggable model of the *method cache*, a novel cache design introduced in the Java Optimized Processor (JOP).

The paper is organized in a bottom-up fashion, starting with the low-level tools and working up to high-level WCET analysis. Section 2 focuses on the Cascade tool, our control flow analyzer, to explain its design and show the improvements it offers in the display of flow information. Section 3 discusses the enhancements made in Clepsydra, our WCET analyzer, to support analysis across method invocations. (Earlier versions of the Cascade and Clepsydra tools could analyze only one method in isolation.) Finally, in Section 4, we evaluate the performance of the Clepsydra tool to support our claim that it is sufficiently fast for interactive analysis of real-time software, even in the presence of method invocations.

## 2 Control Flow Analysis

WCET analysis necessarily begins with knowledge of the exact bytecode that will run on the target device. The analysis must also build up a control flow model of the bytecode. To accomplish these goals, two general tactics have been employed: 1) Write a custom tool that parses Java bytecode and, by analyzing branch targets, groups the bytecode into its constituent basic blocks; or 2) add hooks to a Java compiler so that the abstract syntax tree (AST) it produces can be saved to a file for later analysis by a WCET tool.

Through our research on the Volta tool suite, we have developed a third tactic: decompilation. We observed that the high-level nature of bytecode allows Java decompilers to reproduce a near-perfect representation of the original source code (assuming the bytecode contains standard debugging symbols and has not been obfuscated). Decompilers offer a practical foundation for WCET analysis because they perform almost the same task—that is, analyzing bytecode and building up control flow information—that traditional WCET tools have always done. Moreover, decompilers are usually simpler than compilers and therefore easier to modify for the purpose of WCET analysis.

Decompilers also offer an intriguing side-effect: They map every control flow element (e.g., basic block) to a representative expression in the Java language. By exploiting this feature, analysis tools can annotate control flow

[2]Available at http://volta.sourceforge.net/ under the GNU Public License.

[3]http://jedit.org/

data structures with source code information, making them much easier to comprehend. For example, a node in a control flow graph generated by conventional tools would show only bytecode instructions:

```
iload_1
iload_2
iload_3
imul
iadd
```

With decompiler support, the tool can combine this bytecode sequence with the source code expression it represents:

```
startVelocity + acceleration * deltaTime
```

This feature greatly improves readability of control flow data, yet it would be extremely difficult to implement properly in lower-level languages such as C. In Figure 1, for example, a typical control flow analyzer would show only the bytecode of each basic block, but with decompilation support, the nodes of the graph can be annotated with the Java statements they represent, as shown here.

## 2.1 JODE

To exploit these benefits, our Volta suite includes a decompiler to serve as the foundation for all of its tools. We chose to integrate an existing tool called JODE[4] rather than reimplement one from scratch. We selected JODE for three reasons: 1) It exposes control flow data in a public API; 2) our experiments indicate that it produces more accurate results than other decompilers; and 3) it is distributed under an open-source license, allowing the freedom to modify JODE for the benefit of Volta's higher-level tools.

This freedom is especially important because the original version of JODE discards bytecode during the process of reconstructing source code. Though such details may be unnecessary for a decompiler, they are vital for WCET analysis, and thus we modified JODE to preserve the original bytecode instructions and associate them with their corresponding control flow blocks. We also translated the bytecode into BCEL[5] format (now a *de facto* standard in the Java domain) to enable interoperability with other Java tools.

## 2.2 Cascade

With a strong decompiler as the foundation, we were able to create a unique control flow analyzer for the Volta tool suite. Called Cascade, this tool is built directly on top of JODE and translates the decompiler-specific control flow

[4]http://jode.sourceforge.net/

[5]http://jakarta.apache.org/bcel/

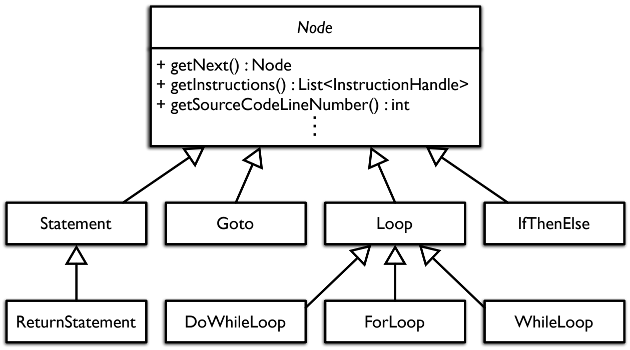

**Figure 2. Our control flow analyzer, Cascade, translates Java programs into control flow graphs and trees consisting of the classes shown in this UML diagram. Each class provides operations for obtaining information about the control flow element, such as the bytecode sequence it represents.**

structures into a general-purpose class hierarchy, as shown in Figure 2. Other tools, such as WCET analyzers, can use these classes directly, shielding them from the complexities of the decompiler. Cascade also provides important reflection services for loading classes, obtaining method handles, computing the static code size of a method, and so on.

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

For the benefit of such tools, Cascade offers flexibility in how it represents control flow data. It provides both a traditional control flow graph as well as a control flow tree that is necessary for tree-based WCET algorithms. For each data structure, Cascade provides an API for operations such as:

- Iterating through the control flow structure
- Detecting loops
- Finding method invocations
- Obtaining the bytecode and human-friendly source code associated with any given node
- Exporting control flow data to SVG, DOT, GML, GraphML, and plain text file formats

The latter feature is particularly valuable because it allows Cascade's output to be fed into other tools for further processing and visualization. For example, the DOT export format allows control flow graphs to be displayed in a Graphviz[6] program, as shown in Figure 1.

[6]http://www.graphviz.org/

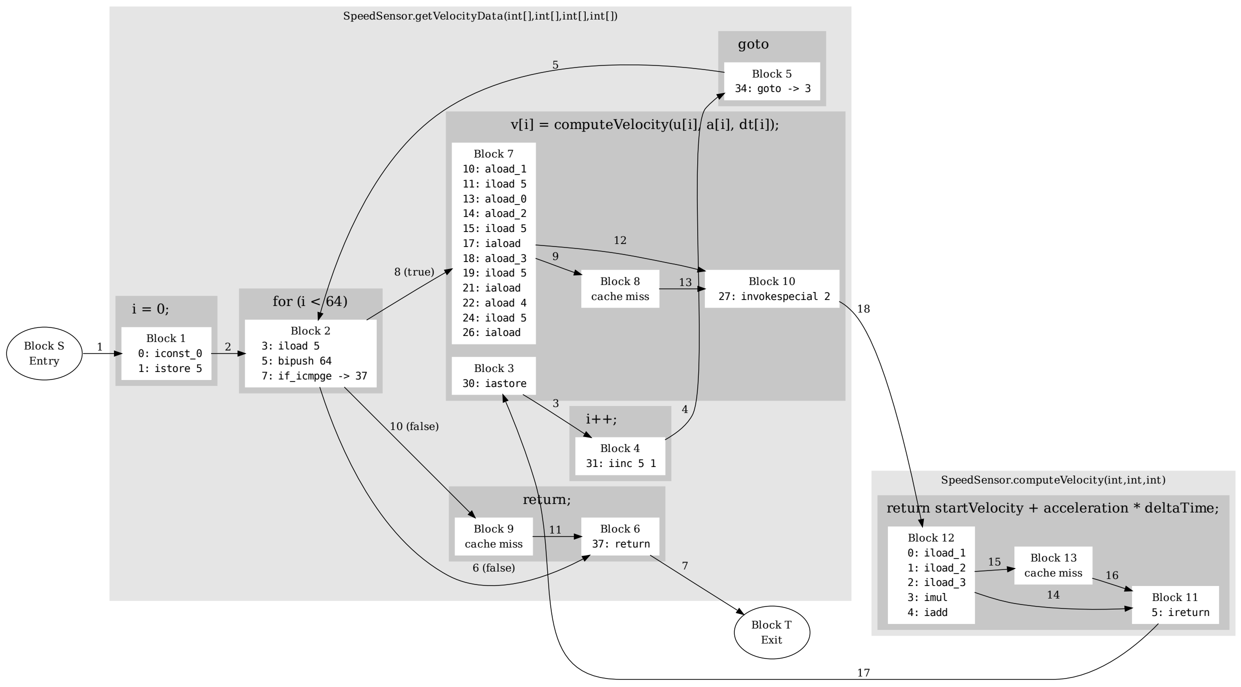

**Figure 1. In addition to programmatic access to control flow via its API, Cascade can generate output for visualization. This figure shows the result of Cascade's analysis of the program in Figure 3. Note how Cascade groups the nodes according to the method in which they belong to enhance readability.**

Note that this figure is not a mock-up; it is the actual output produced by Cascade of the program in Figure 3. The figure also illustrates how Cascade annotates each control flow node in the graph with the corresponding source code provided by the decompiler. Without this extra information, the CFG would be much more difficult to comprehend, and the relationship between the control flow and the original program would be far from obvious. Cascade is the first control flow analyzer to provide such a feature.

## 3 WCET Analysis

In keeping with the modular spirit of the Volta project, our WCET analysis tool is, like Cascade, constructed as an independent program. Called Clepsydra, it implements current WCET analysis theory, including *path analysis*, *exec-time modeling*, and *longest path search* [12]. The following sections describe this theory and its implementation in Clepsydra.

Note that garbage collection is absent from this discussion, despite its traditional importance in Java-based systems. In real-time systems, garbage collection is typically *disabled* for critical tasks and can therefore be ignored. In addition, it is largely a scheduling issue and can be considered independent of WCET analysis on individual threads.

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

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

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

**Figure 3. This program is the basis of Figures 1 and 5. The `@LoopBound` statement is a custom annotation for communicating loop bounds to a WCET analyzer.**

### 3.1 Path Analysis

Prior to calculating a WCET bound, information about the feasible execution paths in a program must be collected. This information is often called flow information or *flow facts*. Identifying precise flow information normally requires some form of control flow reconstruction from the binary code. In the Volta suite, the Cascade tool takes care

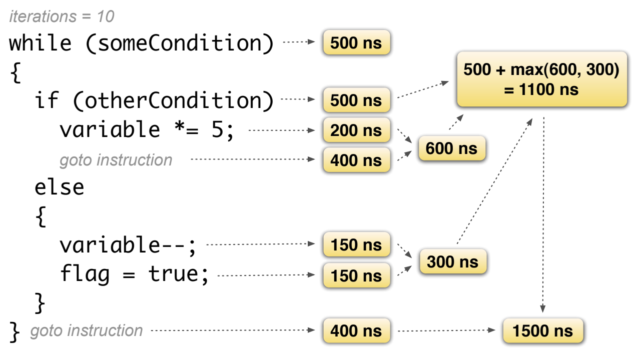

**Figure 4. A tree-based WCET algorithm recurses to the leaves of a control flow tree and returns the sum for each node. When branches are encountered, the worst-case time among all branches is taken, resulting in a WCET of 1500 for the loop body shown here. The body is then multiplied by the maximum number of iterations for a final WCET of 500 + 10 (1500 + 500) = 20,500.**

of this reconstruction.

However, even a perfect reconstruction of the flow model is insufficient. For programs containing loops, their WCET is essentially unbounded; a pure control-flow analyzer without deep semantic analysis has no way of knowing how many times a loop will iterate in the worst case. While recent work has focused on abstract interpretation to find bounds automatically [6, 7], the more common approach is to rely on annotations. These loop bound annotations must be inserted manually, thus they require more effort from the developer and are error-prone, but they make WCET analysis much faster and simpler. (Our prior efforts have focused on adapting Java's built-in annotation mechanism for this purpose [10]. Benefits include "for free" syntax checking, type safety, and support from existing tools.)

## 3.2 Exec-Time Modeling

Once the control flow has been reconstructed and loop bounds obtained, the core of the analysis begins with a process known as *exec-time modeling* [12]. It assigns each instruction in the program an execution time. The nanosecond values in Figure 4, for example, must be derived from some model of the target processor. In addition, knowledge of cache behavior is required for a tight bound on the WCET.

### 3.2.1 Instruction Timing

Exec-time modeling on modern processors is quite difficult. They include multi-level caches, branch prediction, and out-of-order execution that introduce state whose exact value depends on a large execution history. Modeling this history leads to a state explosion for the final WCET calculation. As a result, low-level WCET analysis usually requires simplifications of the CPU model, producing an excessively conservative estimate.

A novel solution, as discussed in Section 1.1, is to rely on a Java-specific processor such as JOP, which strikes a balance between average case performance and ease of WCET analysis. JOP avoids complex features such as pipeline dependencies, prefetch queues, and automatic stack dribbling, as found in other Java processors [15]. As a consequence, there are no timing dependencies across bytecode boundaries, and pipeline analysis [5] can be omitted. The rules to aggregate timing values [22] can be applied without introducing significant conservatism.

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

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

### 3.2.2 Instruction Caching

Simple processors without caching are naturally an easier target for WCET analysis. Yet even in resource-constrained embedded systems, caches are now mandatory due to the growing gap between processor performance and memory access time. To bridge this gap, the design of JOP introduces two time-predictable caches: A *stack cache* [18] to speed up access to variables and operands on the execution stack, and a *method cache* [17] as a special kind of instruction cache.

The stack cache is a simple two-level on-chip memory. The two top-most elements of the stack are held in registers, and the subsequent elements are stored in on-chip block RAM. There is no automatic exchange between on-chip RAM and the main memory, as found in picoJava [15], which would introduce complex timing interactions between instructions. The exchange is under microcode control and can be restricted to method invocation or thread switching.

The method cache stores whole Java methods and is filled only on an invoke or a return instruction; all other instructions are a guaranteed cache hit. This observation greatly simplifies low-level analysis because the context of

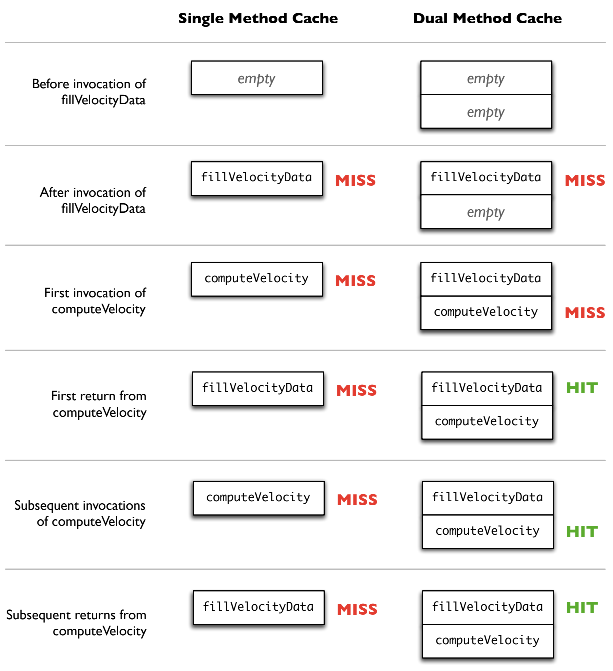

**Figure 5. A purely control-flow analysis can identify guaranteed hits and misses of the method cache in certain situations. This diagram shows one such situation: a leaf method within a loop, as in Figure 3. All invocations of `computeVelocity` are misses in the single method cache, but only the first invocation is a miss for the dual method cache.**

the instruction, such as whether it lies within a loop, can be ignored completely without neglecting the effects of the cache.

JOP's method cache is also unique in that its architecture can vary according to the desired sophistication of WCET analysis. In sharp contrast to block-level instruction caches, its most basic form is the *single method cache* [17], in which the total size of the cache matches the size of the largest method to be executed. In this configuration, every method invocation and return is a guaranteed miss, as shown in Figure 5. (Note that this is still a caching solution because it converts all non-invocation and non-return instructions into cache hits.)

Although the single method cache makes WCET analysis simple and fast, the cache miss on every invocation and return causes programs to run slowly, especially given the large number of method invocations in typical Java software. To increase the cache hit ratio, JOP can also be configured to store more than one method at a time. For example, a *dual method cache* [17] stores two methods at once using a least-recently used replacement strategy. As before, both areas of the cache must be large enough to hold the largest method in the program. (The largest method could end up in either area, depending on the call graph).

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

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

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

Fortunately, other code structures are more amenable to analysis. If, for example, the call to `methodB` were removed, and `methodA` makes no further invocations, then two guarantees can be made: the invocation of `methodA` and the return from `methodA` will *always* be cache hits (except for the first iteration when `methodA` is loaded into the cache, as shown in Figure 5).

By identifying such structures in the code—that is, a loop that executes only one kind of method—analyzers can improve their estimate of the WCET. Our prior work describes how to add constraints to an IPET formulation in order to achieve these improvements [21]. The *variable block method cache* [17] is more WCET-friendly than a standard cache (the analysis depends on method invocations instead of individual instructions) but more complex. The integration of the analysis [13] into Clepsydra is future work.

## 3.3 Longest-Path Search

After identifying feasible paths through *path analysis* and assigning execution times to instructions through *exec-time modeling*, the WCET bound (i.e., the longest path) can be calculated. This phase is also known as *calculation of execution scenarios* [12] because additional information can also be obtained from this final phase, such as best-case execution time. The process normally takes one of two forms: a tree-based algorithm or a graph-based algorithm.

### 3.3.1 Tree-based Algorithms

Tree-based algorithms were among the very first implementations of WCET analysis [16]. They operate by recursively descending the nodes of a program's control-flow tree, returning the execution time for each node. The value returned for the root node is the total WCET.

As the algorithm encounters each control-flow node, it must decide how to compute the WCET based on the node's type. For straight-line code, the time to execute each instruction is simply summed. For branches (`if` and `switch` statements), the path whose execution time is highest—the "worst" path—is taken as the total time. For loops, the maximum number of iterations is multiplied by the WCET of the loop's body. Figure 4 shows an example of this concept for a simple loop.

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

The side effect of this performance advantage is that analysis can be made *interactive*. Just as in Eclipse, where the Java compiler runs in the background to reveal syntax errors during typing, WCET analysis can run in parallel with the text editor, providing immediate feedback to the developer of the worst-case path [9]. Without tree-based algorithms and the simplicity provided by Java microprocessors, this feature would be nearly impossible, given the time complexity of the problem and the speed of today's development workstations.

Despite these advantages, the tree-based algorithm has fallen out of favor among WCET researchers. It suffers from certain drawbacks, such as a susceptibility to the false path (a.k.a. infeasible path) problem, in which data dependencies between two `if` statements can fool the algorithm into computing an overly pessimistic WCET [1]. In general, tree-based algorithms have difficulty handling any type of dependency across sibling nodes due to the nature of tree traversal.

#### 3.3.2 IPET-based Algorithms

As a solution to problems such as the false path, an alternative technique known as *implicit path enumeration* has emerged [14]. These algorithms require a control flow graph, instead of a tree, as input. They operate by finding the maximum possible "weight" (i.e., maximum time) between the source of the graph and its sink. Thus, they treat WCET computation as an instance of the maximum flow problem, solvable by a variety of techniques: Ford-Fulkerson, Edmonds-Karp, integer linear programming (ILP), etc.

WCET is a special case, however, due to the presence of loops in software programs. Most maximum flow algorithms assume an acyclic graph. For this reason, the ILP technique is commonly used to compute the flow. ILP solves the loop problem by defining maximum flow according to *constraints* on the legal flow through the graph. Accounting for a loop is simply a matter of adding an additional constraint to bound the amount of flow—that is, the number of iterations—through the loop.

To illustrate, consider the control flow graph of Figure 1, whose source code is given in Figure 3. An IPET-based algorithm would add constraints for each node in the graph. The constraint on Block 7, for example, would be $edge8 = edge9 + edge12$, indicating that the incoming flow is equal to the outgoing flow. To account for the loop, an additional constraint for Block 2 would be added: $64 \cdot edge2 = edge8$, indicating that the flow through edge 2 is 64 times as large as the flow through edge 8.

Solving the false path problem would be handled in a similar way. A constraint to relate the two `if` statements would be added to the ILP formulation. Determining this constraint is far from trivial, however. Unless the false path can somehow be identified—an NP-complete problem by itself—then the IPET approach can offer no improvement, degrading to the same level of pessimism as a simple tree algorithm. This is an important point to remember: Contrary to conventional wisdom, IPET algorithms are not inherently more accurate than their tree-based counterparts. They improve only when provided with the appropriate flow constraints, a task that is quite difficult to accomplish in practice.

Another disadvantage of IPET-based techniques is their slothful performance. The time required by such algorithms grows exponentially in the worst case as the cyclomatic complexity of the analyzed program increases [9]. (By comparison, tree-based algorithms grow linearly.) Their slowness makes IPET algorithms impractical for the kind of interactive WCET analysis described in Section 3.3.1.

### 3.4 Clepsydra

Clepsydra provides a complete implementation of the theory described in this section, including the method cache modeling. As an additional benefit, it is designed to support a variety of different theoretical approaches because, as discussed in Section 3.3, WCET analysis techniques offer different strengths and weaknesses. Some techniques run fast but may not find a tight bound (e.g., tree-based approaches); others can be made tighter but require exponential running time in the worst case (e.g., implicit path enumeration). No single approach is ideal, and for this reason Clepsydra makes analysis techniques pluggable via the Strategy pattern. Developers can switch between *analysis strategies* with relative ease, allowing the same Clepsydra framework to be used as existing techniques are refined and new ones are created.

This flexibility is particularly important with respect to back-annotation (see Section 3.3.1). When making back-annotation interactive, the speed of the analysis technique is a major factor in usability. A developer cannot afford to wait for a lengthy analysis to complete after every change of the source code. Therefore, Clepsydra offers a *hybrid*

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

  switch (opcode) {
    case SIPUSH:
      return 3;

    case LDC:
      return 7 + readWaitStates;

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

**Figure 6. Clepsydra relies on the Strategy pattern to make processor timing definitions easily swappable. This listing shows a portion of the JOP timing strategy.**

approach: Fast tree-based analysis is used by default for interactive back-annotation, but if the developer suspects the results are too pessimistic, Clepsydra can switch to a slower technique for tighter WCET estimation.

Clepsydra also offers the flexibility to support arbitrary CPUs for exec-time modeling. It provides a JOP processor model by default, and users can plug in a custom *timing strategy* to describe other Java processors, such as the aJile chip. Creating the plug-in requires the implementation of a simple Java interface, as illustrated in Figure 6.

Loop bound determination is factored out via the Strategy pattern, as well. Developers can plug in the default annotation-based strategy supplied by Clepsydra, or they can implement their own without having to understand the details of Clepsydra's design.

## 3.5 Method Cache Analysis

Since our previous work [9], we have extended Clepsydra to support analysis of JOP's method cache. It now supports analysis of whole programs across method invocations. This required a change in Cascade, as seen in Figure 1. Note the extra cache miss blocks; these represent the alternate path in control flow in the event of a method cache miss. Miss blocks were also added to precede return instructions because they can also cause cache misses. The execution time, or "cost," of control flow passing through these blocks is equal to the miss penalty alone (i.e., the number of extra cycles needed to load the method), not the total time for invocation or return.

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

For the dual method configuration, only the simple analysis described in Section 3.2.2 was implemented. When Clepsydra encounters a method invocation, it first checks whether the invocation occurs within a loop and whether it is the only one of its type in the entire loop body. If both conditions are true, the invocation is a miss on the first iteration but a guaranteed hit on all subsequent iterations. Clepsydra uses this fact to add a constraint in the IPET formulation. In the case of Figure 1, for example, it would add the following constraints for the invocation in block 10:

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

where the constant $n$ is the loop bound.

For return instructions, Clepsydra considers only leaves in the call tree. (A leaf is a method that invokes no methods.) Any return instruction in a leaf is always a guaranteed hit (for the dual-block cache), not only within loops. Cascade's API makes testing for this condition easy; it requires only a simple expression:

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

This expression obtains the control flow tree belonging to the return instruction, then checks whether the set of method invocations in that tree is empty. (For a full explanation of `getMethodInvocations`, refer to the Volta API documentation.[7])

Figure 7 shows the result of a complete analysis that puts all of these constraints together. The listing is an ILP formulation by Clepsydra of the CFG in Figure 1.

## 3.6 Tree Algorithm for Method Analysis

As discussed in Section 3.3.2, the IPET approach is computationally expensive. Even for small programs, computing WCET in the presence of method invocations can take ten seconds or more; larger programs may require several minutes even on a fast workstation. For interactive analysis of real-time software, this delay is unacceptable.

To solve this problem, we developed a tree-based algorithm capable of analyzing the dual-method cache described in Section 3.2.2. It offers accuracy that is identical to the IPET algorithm but executes in a fraction of the time. (A concrete performance analysis of the algorithm is provided in Section 4.1.)

[7]http://volta.sourceforge.net/api/index.html

```
+edge1 = 1;
+edge7 = 1;
+edge1 -edge2 = 0;
+edge2 +edge5 -edge6 -edge8 -edge10 = 0;
-edge3 +edge17 = 0;
+edge3 -edge4 = 0;
+edge4 -edge5 = 0;
+edge6 -edge7 +edge11 = 0;
+edge8 -edge9 -edge12 = 0;
+edge9 -edge13 = 0;
+edge10 -edge11 = 0;
+edge12 +edge13 -edge18 = 0;
+edge14 +edge16 -edge17 = 0;
-edge14 -edge15 +edge18 = 0;
+edge15 -edge16 = 0;
-64 edge2 +edge8 = 0;
+3 edge1 -block1 = 0;
+8 edge2 +8 edge5 -block2 = 0;
+38 edge17 -block3 = 0;
+8 edge3 -block4 = 0;
+4 edge4 -block5 = 0;
+21 edge6 +21 edge11 -block6 = 0;
+119 edge8 -block7 = 0;
-block8 = 0;
-block9 = 0;
+75 edge12 +75 edge13 -block10 = 0;
+23 edge14 +23 edge16 -block11 = 0;
+39 edge18 -block12 = 0;
-block13 = 0;
```

**Figure 7. This ILP formulation, shown here in lp_solve format, is produced by Clepsydra when applying the IPET analysis strategy to Figure 3. Note that the cache miss constraints (blocks 8, 9, and 13) evaluate to zero because the methods are so small that their miss penalties are zero.**

Our algorithm is a two-pass variation of the standard tree analysis approach (i.e., recursive descent). In the first pass, we treat all method invocations as if they were cache misses. We then execute a second pass through the control flow tree and examine only the method invocations. For invocations determined to be cache hits, we multiply the number of hits by the miss penalty for that particular invocation. This penalty is actually the *gain time* because the first pass assumed only cache misses. We simply subtract the gain time from the first pass total to arrive at the final WCET calculation. Figure 8 provides a pseudocode listing of this approach. A complete implementation can be found in the Volta distribution.

## 4 Performance Analysis

One of the key benefits of the Volta tool suite is interactive development of real-time software, such as the back-annotation feature described in Section 3.3.1. Given the importance of speed in interactive back-annotation, the running time of the various WCET analysis techniques becomes a prime consideration. Almost all prior work has focused on reducing pessimism in WCET analysis, but there has been little to no emphasis on reducing its execution time. The idea of near-instantaneous analysis offers the potential for a new breed of real-time software development tools, such as the editor plugin described in Section 1.2.

```
getWCET(node)
  if node is null return 0
  if node is if-then-else
    return getExprWCET(if expression) +
         max(getWCET(then branch), getWCET(else branch))
  if node is loop
    return getExprWCET(loop expression) +
           getWCET(loop body) * loop bound
  if node is statement
    return getExprWCET(statement expression)

getExprWCET(expression)
  return the sum of CPU cycles for all instructions
  (assume all invocation instructions are cache misses)

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

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

**Figure 8. The standard recursive descent algorithm in tree-based analysis cannot account for method invocations because method cache hits are not constant across loop iterations. A two-pass variation of the algorithm, shown here in pseudocode, solves this problem.**

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

### 4.1 Speed of Analysis

Our performance measurements with Clepsydra show that this is not yet possible, as evidenced by Figure 9. The chart shows benchmarks for three analysis techniques: the tree technique, the IPET technique using lp_solve,[8] and the IPET technique using GLPK.[9] (The seemingly redundant benchmarks for IPET are designed to rule out the possibility that a lackluster performance of the IPET approach is merely the result of an inefficient ILP solver.)

[8]http://lpsolve.sourceforge.net/5.5/

[9]http://www.gnu.org/software/glpk/

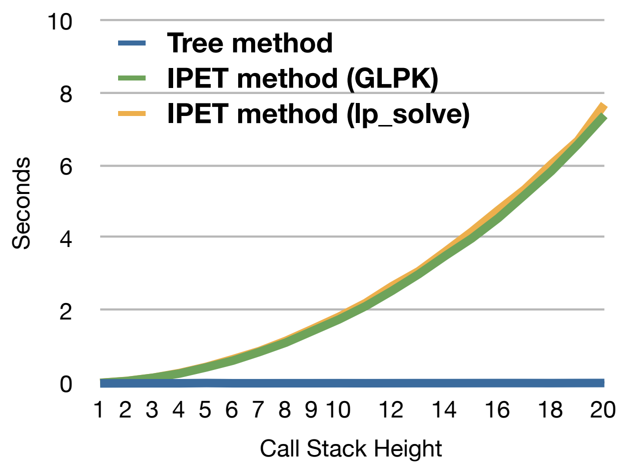

**Figure 9. In the presence of method invocations with cache analysis, the tree technique still outperforms the IPET alternative.**

In prior work [9], we tested how each technique performed as the cyclomatic complexity of a contrived input program increases. It showed that the tree-based technique was the clear winner. Its performance, while linear in growth, appears virtually constant, requiring only a few milliseconds even at high complexity. As expected, the IPET technique is much slower, growing exponentially with program size. For interactive back-annotation, tree-based analysis is clearly the best choice and deserves greater attention from the research community.

In light of our recent improvements in Clepsydra to support method invocations, there was a question of whether the tree-based technique's stellar performance would remain. We evaluated the speed of all three algorithms again using the same hardware and software testbed, this time running a benchmark that invokes methods (see Figure 9). Note that it contains three curves, one for the tree technique and two for the two independent implementations of IPET.)

For the evaluation, we measured the performance of the algorithms under increasing complexity by increasing the height of the call stack. (For example, a method that calls a method that calls a method has a stack height of three.) Although the presence of methods slowed the performance of all techniques, the trends were the same: The tree-based approach is extremely fast while the IPET approach is exponentially slower.

## 4.2 Accuracy of Analysis

Although the primary focus of this work is the flexibility and speed of WCET analysis tools, the accuracy of those tools is certainly a factor as well. We define the accuracy

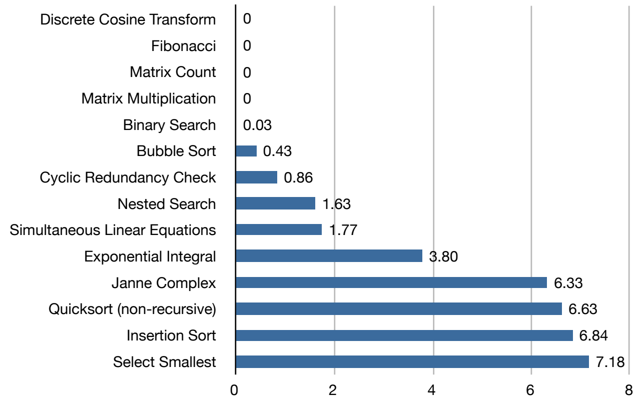

**Figure 10. Clepsydra's pessimism ratio for a variety of WCET benchmarks.**

of a WCET analysis tool in terms of *pessimism*; that is, the amount by which the predicted and measured WCET values differ.

To evaluate the pessimism of our Clepsydra tool, we created a set of fifteen WCET benchmark programs. They are based on a similar suite of benchmarks from the Mälardalen Real-Time Research Center.[10] They are available in the Volta distribution under a public domain license. We offer these benchmarks as a *de facto* standard for evaluating Java-based WCET analysis tools.

Figure 10 shows the results of running the benchmarks on Clepsydra with a 100 MHz JOP processor (Altera Cyclone implementation) as the target.[11] The benchmarks vary widely. Discrete Cosine Transform, Fibonacci, Matrix Count, and Matrix Multiplication exhibit the ideal behavior of 0% pessimism because they are simple loops. The Select Smallest benchmark, a complex piece of code with many nested conditionals and loops, fared the worst at more than 700% pessimism. (That is, the time predicted by Clepsydra was about 7 times larger than the actual worst-case time measured on the JOP.)

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

[10] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[11] The Petri Net benchmark is missing from these results because it contains a single method 20 kilobytes in size, which overflows the method cache in our FPGA version of JOP.

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

## 5 Conclusion

Given the importance of WCET analysis tools, especially in the case of safety-critical real-time systems, it is surprising that some commercial vendors have license agreements that prohibit publication of any user-experience data.[12] Even serious defects in the tools offered by these vendors will be hidden. In contrast, the Volta suite is 100% open source, and we encourage users to evaluate it and report their findings.

## References

[1] P. Altenbernd. On the false path problem in hard real-time programs. In *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems (EURWRTS 2006)*, pages 102–107, Los Alamitos, CA, USA, June 1996. IEEE Computer Society.

[2] E. G. Benowitz and A. F. Niessner. Experiences in adopting real-time Java for flight-like software. In *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, volume 2889 of *Lecture Notes in Computer Science*, pages 490–496. Springer Berlin, October 2003.

[3] G. Bernat, A. Burns, and A. Wellings. Portable worst-case execution time analysis using Java byte code. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (Euromicro-RTS 2000)*, pages 81–88, Los Alamitos, CA, USA, June 2000. IEEE Computer Society.

[4] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison Wesley Longman, January 2000.

[5] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2002.

[6] P. A. Guedes and S. V. Cavalcante. On the design of an extensible platform for flow analysis of Java using abstract interpretation. In *Proceedings of the Third International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*, pages 47–50, July 2003.

[7] J. Gustafsson, B. Lisper, R. Kirner, and P. Puschner. Code analysis for temporal predictability. *Real-Time Systems*, 32(3):253–277, March 2006.

[8] D. S. Hardin. Real-time objects on the bare metal: An efficient hardware realization of the Java virtual machine. *Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, pages 53–59, May 2001.

[9] T. Harmon and R. Klefstad. Interactive back-annotation of worst-case execution time analysis for java microprocessors. In *Proceedings of the Thirteenth IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, 2007.

[10] T. Harmon and R. Klefstad. Toward a unified standard for worst-case execution time annotations in real-time Java. In *Proceedings of the Fifteenth International Workshop on Parallel and Distributed Real-Time Systems*, page 151. IEEE Computer Society, March 2007.

[11] Imsys Technologies. *IM1101C Technical Reference Manual*, 0.25 edition, October 2004.

[12] R. Kirner and P. Puschner. Classification of WCET analysis techniques. In *Proc. 8th IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 190–199, Seattle, WA, May 2005.

[13] R. Kirner and M. Schoeberl. Modeling the function cache for worst-case execution time analysis. In *Proceedings of the 44rd Design Automation Conference, DAC 2007*, San Diego, CA, USA, June 2007. ACM.

[14] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 88–98, New York, NY, USA, 1995. ACM Press.

[15] J. M. O'Connor and M. Tremblay. picoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, 1997.

[16] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

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

[18] M. Schoeberl. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, Denver, Colorado, USA, April 2005. IEEE.

[19] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, Vienna, Austria, January 2005.

[20] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, doi:10.1016/j.sysarc.2007.06.001, 2007.

[21] M. Schoeberl and R. Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.

[22] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Trans. Softw. Eng.*, 15(7):875–889, 1989.

[12]http://www.cs.york.ac.uk/hise/safety-critical-archive/2007/0497.html