Chronos: a Timing Analyzer for Embedded Software

Xianfeng Li

Department of Computer Science and Technology, Peking University, China

Yun Liang

Department of Computer Science, National University of Singapore, Singapore

Tulika Mitra *

Department of Computer Science, National University of Singapore, Singapore

Abhik Roychoudhury *

Department of Computer Science, National University of Singapore, Singapore

Abstract

Estimating the Worst-case Execution Time (WCET) of real-time embedded software is an important problem. WCET is defined as the upper bound b on the execution time of a program P on a processor X such that for any input the execution time of P on X is guaranteed to not exceed b. Such WCET estimates are crucial for schedulability analysis of real-time systems. In this paper, we present Chronos, a static analysis tool for generating WCET estimates of C programs. It performs detailed micro-architectural modeling to capture the timing effects of the underlying processor platform. Consequently, we can provide safe but tight WCET estimate of a given C program running on a complex modern processor. Chronos is an open-source distribution specifically suited to the needs of the research community. We support processor models captured by the popular SimpleScalar architectural simulator rather than targeting specific commercial processors. This makes Chronos flexible, extensible and easily accessible to the researcher.

Key words: Worst Case Execution Time (WCET) Analysis

* Corresponding Authors
Email addresses: lixianfeng@mprc.pku.edu.cn (Xianfeng Li), liangyun@comp.nus.edu.sg (Yun Liang), tulika@comp.nus.edu.sg (Tulika Mitra), abhik@comp.nus.edu.sg (Abhik Roychoudhury).

1 Introduction

Estimating the Worst Case Execution Time (WCET) of a program is an important problem [20,22,14,26]. WCET analysis computes an upper bound on the program's execution time on a particular processor for all possible inputs. The immediate motivation of this problem lies in the design of real-time embedded systems. Typically an embedded system contains processor(s) running specific application programs and communicating with an external environment in a timely fashion. Many embedded systems are safety critical, e.g., automobile, avionics and healthcare monitoring applications. The designers of such systems must ensure that all the real-time constraints are satisfied. Real-time constraints impose hard deadlines on the execution time of embedded software. WCET analysis of the program can guarantee that these deadlines are met. A survey of WCET analysis techniques appears in [17].

Due to its inherent importance in embedded system design, timing analysis of embedded software has been studied extensively. Accurate timing analysis critically depends on modeling the effects of the underlying micro-architecture. Ignoring the micro-architecture can produce extremely pessimistic time bounds. This is particularly so because modern processors employ advanced micro-architectural features such as pipeline, caches, and branch prediction to speed up program execution. Therefore, to obtain safe but tight WCET estimate of a program, we need to model the timing effects of these architectural features and their complex interactions.

In this paper, we present a WCET analysis tool named Chronos [1]. Chronos estimates WCET through static program analysis. It incorporates timing models of different micro-architectural features present in modern processors. In particular, it models *superscalar in-order and out-of-order pipelines, instruction caches, dynamic branch prediction and their interactions.* The modeling of different architectural features is parameterizable. For example, the user can set the line size, number of lines, and the associativity of the instruction cache. The user can also choose among various dynamic branch prediction schemes (including popular schemes such as GAg, gshare, etc.) and set the sizes of the associated hardware structures such as branch history register, branch prediction table etc. Similarly, pipeline parameters such as the issue width, number/type of functional units, sizes of different pipeline buffers etc. can be set by the user.

The input to Chronos is a C program. The front end performs limited data flow analysis at C source code level to determine loop bounds, failing which it requests the user to provide this information. However, the core of the

[1] The name is taken from ancient Greek mythology where Chronos was the personification of time.

analyzer operates on the binary executable of the program. This is because micro-architectural modeling requires details that are only available at the binary executable level [2]. The analyzer disassembles the binary executable to construct the assembly level control flow graph of the program. It then finds the timing estimate of each basic block through detailed micro-architectural modeling. The timing estimates of the individual basic blocks are combined together to estimate the WCET of the program. For this step, we provide a mapping of the loop bounds and other program path related information from the source code to the binary executable level.

What are the distinguishing features of Chronos w.r.t. other WCET analysis tools and prototypes? We note that Chronos captures the timing effects of complex micro-architectural features such as out-of-order pipelines and dynamic branch prediction. One pragmatic issue is whether one needs to model the timing effects of such complex micro-architectural features for obtaining reasonable WCET estimates. We observe that current high-performance processors employ out-of-order execution engines to mask latencies due to pipeline stalls; these stalls may happen due to data dependency, resource contentions, cache misses, branch mispredictions, etc. In the embedded domain, many recent processors employ out-of-order pipelines and other complex features; examples include Motorola MPC 7410, PowerPC 755, PowerPC 440GP, AMD-K6 E and NEC VR5500 MIPS.

Apart from the functionality provided in terms of advanced micro-architectural modeling, Chronos provides several advantages in terms of its usage in research/development. First, *Chronos is a completely open source distribution especially suited to the needs of the research community.* It can be downloaded from

`http://www.comp.nus.edu.sg/~rpembed/chronos`

This allows the researcher to modify and extend the tool for his/her individual needs. Current state-of-the-art WCET analyzers, such as aiT [1], are commercial tools that do not provide the source code. Even most of the research prototypes, such as Cinderella [13], do not make the source code available. The only notable exception in this aspect is HEPTANE [19], which is open source. However, HEPTANE does not support complex architectural features such as out-of-order pipeline and global branch prediction.

Secondly and more importantly, unlike other WCET analyzers, *Chronos is not targeted towards one or more commercial embedded processors.* Instead, it is built on top of the freely available SimpleScalar simulator infrastructure. SimpleScalar is a widely popular cycle-accurate architectural simulator that allows

[2] For example, instruction cache and branch prediction analysis require exact memory addresses of the instructions.

the user to model a variety of processor platforms in software [3]. We target our analyzer to processor models supported by SimpleScalar. *This choice of platform ensures that the user does not need to purchase a particular embedded platform and its associated compiler, debugger and other tools (which are often fairly expensive) in order to conduct research in WCET analysis using Chronos.* Also, the flexibility of SimpleScalar enables development and verification of modeling a variety of micro-architectural features for WCET analysis.

The rest of the paper is organized as follows. The next section gives a high-level view of the WCET analysis technique used in Chronos, as well as the challenges it addresses. Section 3 describes the workflow of Chronos in more details. Based on the technical discussions in Sections 2 and 3, Section 4 walks through a sample usage of the tool by showing some user-interactions, screenshots and intermediate analysis results. Section 5 narrates some results obtained using Chronos, in the context of the recently organized "WCET Tool Challenge 2006" for comparing and evaluating existing WCET analysis tools. Section 6 compares Chronos to existing WCET analysis tools. Finally, Section 7 provides conclusions and some discussions on Chronos.

2 WCET Analysis Technique

The execution time of a program is determined by the program path taken during execution, as well as the timing of instructions along that path. Consequently, Worst-Case Execution Time (WCET) analysis should take care of these two issues. For instruction timing, Chronos performs microarchitecture modeling to capture the timing effects of performance enhancing microarchitectural features. This work is done at the level of basic blocks, that is, it returns the upper bound on execution time of each basic block in the program's control flow graph.

With the knowledge of execution times of basic blocks, Chronos represents the execution time of the whole program using an Integer Linear Programming (ILP) formulation, and uses an ILP (or simply a Linear Programming) solver to find the WCET estimate. Formally, let $\mathcal{B}$ be the set of basic blocks of a program. The program's WCET is given by the following objective function

$$\text{maximize} \sum_{B \in \mathcal{B}} N_B * c_B$$

where N_B is an ILP variable denoting the execution count of basic block B and c_B is a constant denoting the WCET estimate of basic block B. The linear

constraints on N_B are developed from the flow equations based on the control flow graph. Thus for basic block B,

$$\sum_{B' \to B} E_{B' \to B} = N_B = \sum_{B \to B''} E_{B \to B''}$$

where $E_{B' \to B}$ ($E_{B \to B''}$) is an ILP variable denoting the number of times control flows through the control flow graph edge $B' \to B$ ($B \to B''$). Additional linear constraints are also provided to capture loop-bounds and any infeasible path information known to the user.

The main functionalities provided by Chronos are in the domain of complex micro-architectural modeling, such as modeling of out-of-order (as well as in-order) pipelines, instruction cache and branch prediction. All of this is done to get tight estimates for the constants c_B — the maximum execution time of the individual basic blocks. The main obstacle to achieve this modeling lies in the timing anomaly problem [16]. Let us consider an instruction I with two possible latencies l_{min} and l_{max} such that $l_{max} > l_{min}$. The variation of latency could be due to different reasons: cache hit/miss for a load instruction, variable number of cycles taken by an arithmetic instruction like multiplication etc. Let us assume that the execution time of a sequence of instructions containing I is g_{max} (g_{min}) if I incurs a latency of l_{max} (l_{min}). The latencies of the other instructions in the sequence are fixed. A timing anomaly happens if either $(g_{max} - g_{min}) < 0$ or $(g_{max} - g_{min}) > (l_{max} - l_{min})$. In the presence of timing anomaly, techniques which generally take the local worst case for WCET estimation no longer guarantee safe bounds. Even in the absence of caches and branch prediction, we cannot avoid timing anomaly if the pipeline executes instructions out-of-order. As a result, we cannot estimate the WCET of even a sequence of instructions by assuming the maximum latency of each of the instructions. Instead, all possible instruction schedules need to be considered. For a piece of code with N instructions and each of which has K possible latencies, a naive approach, which examines each possible schedule individually, will have to consider K^N schedules. The recent work [12] shows a simple example of a code fragment with variable latency instructions where timing anomaly is exhibited. Similarly, it is not safe to assume that the worst case timing behavior of a sequence of instructions results from cache misses for all the instructions.

How does Chronos bypass the timing anomaly problem to efficiently estimate WCET of each basic block in the presence of complex micro-architectural features? The basic idea of finding the WCET estimate of a basic block B without enumerating instruction schedules is as follows. We observe that the worst-case timing behavior of B occurs from maximum resource contention among instructions in B, that is, each instruction being delayed by maximum number of other instructions. We produce very coarse estimates for the time

intervals at which instructions in B can start/finish execution by initially assuming that any instruction in B can delay the other instructions, except for contentions ruled out by data dependencies. The estimates allow us to rule out certain contentions — if the earliest time at which instruction I is ready for execution occurs after the latest time at which J finishes, clearly I cannot delay J. This allows us to further refine the estimates, thereby ruling out more contentions. The process continues until a fixed point is reached. The WCET of the basic block B is the maximum time between the fetch of B's first instruction and commit of B's last instruction.

The above is only a brief sketch of our pipeline modeling. More details of our out-of-order pipeline modeling can be obtained from [12]. Needless to say, *Chronos can model the timing effects of simple in-order pipelines as well.* The modeling of cache and branch prediction appears in [11,18]. The integration of cache/branch prediction modeling with pipeline modeling appears in [12]. By varying and configuring the different micro-architectural features, we can model the timing effects of different processors. This gives us the flexibility to estimate the execution time of a given program on different processors without actually having access to the processors.

In summary, given a C program and a processor configuration, Chronos returns an estimated Worst-Case Execution Time (WCET) for the program. The estimated WCET is *guaranteed* to be not less than the program's actual execution time for any input. How do we know whether the estimated WCET is a *tight* estimate, that is, the estimated WCET does not substantially exceed the actual WCET of the program? Finding the *actual WCET* is difficult even for programs with few paths in the presence of complex micro-architectural features. For example, even for a program with a single path, finding the actual WCET in the presence of out-of-order pipelines can be difficult since the program path can have variable latency instructions and the exact worst case can only be determined by exhaustively considering all instruction schedules. For this reason, we measure the accuracy of the estimation with the help of architectural simulation. In other words, we simulate the given program using several data inputs that are likely to lead to longer execution times. We call the result obtained through simulation *Observed WCET*, which is guaranteed to be less than the actual WCET. The *Estimated WCET*, on the other hand, is guaranteed to be more than the actual WCET. Thus

$$Estimated\ WCET \geq Actual\ WCET \geq Observed\ WCET$$

Ideally, we would like to compare the Estimated WCET with the actual WCET to find the accuracy of our analysis. Since we do not know the actual WCET, we conservatively compare the *Estimated WCET* with the *Observed WCET* to assess the accuracy of the WCET estimation. If the Estimated WCET is close to the Observed WCET, clearly this means that the Estimated WCET is

close (possibly even closer) to the actual WCET. We now explain the detailed workflow of Chronos.

3 Analyzer Internals

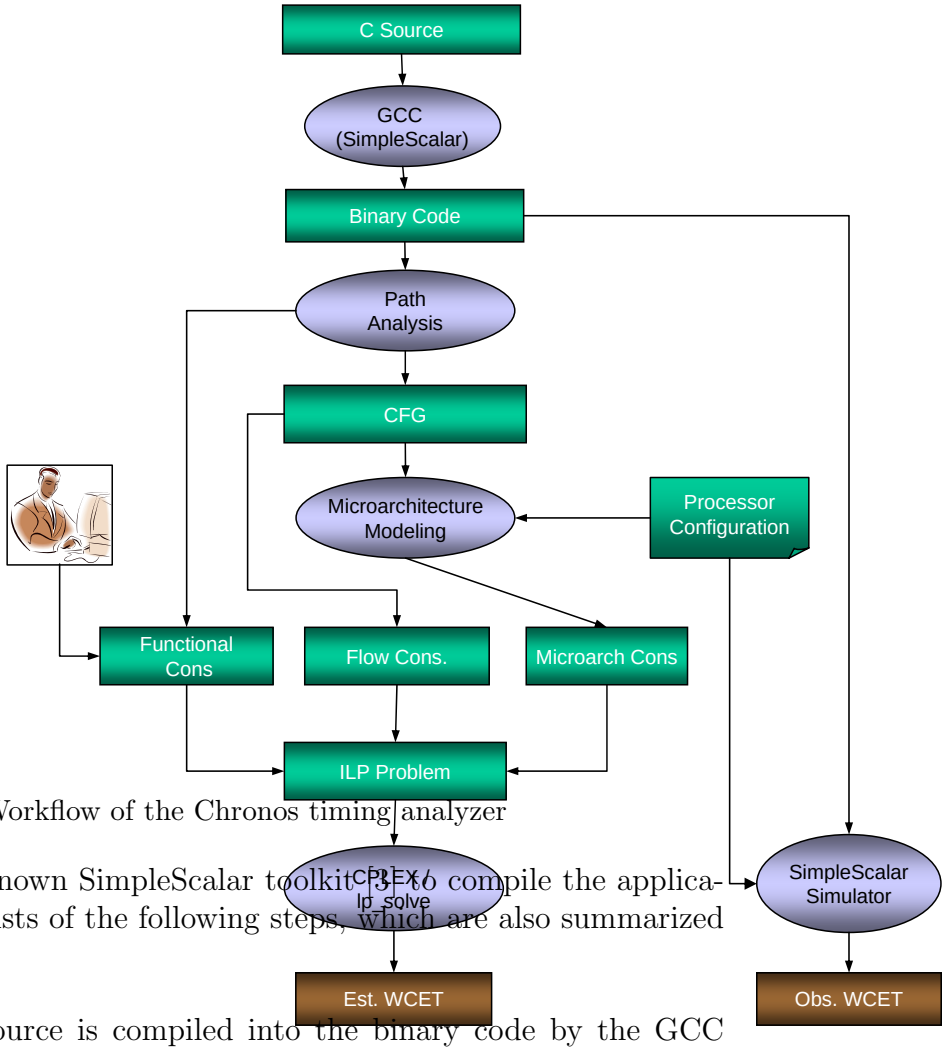

Fig. 1. Workflow of the Chronos timing analyzer

Chronos uses the well-known SimpleScalar toolkit [3] to compile the applications. Its workflow consists of the following steps, which are also summarized in Figure 1.

- First, the program source is compiled into the binary code by the GCC compiler of the SimpleScalar toolset. This GCC version yields binary executable corresponding to an instruction set architecture (ISA), which is a superset of MIPS ISA.

- Chronos reads in binary executable and reconstructs the control flow graph (CFG) by disassembling the binary. The control flow information is represented as linear constraints which are called "flow constraints". Chronos performs a lightweight dataflow analysis to find out the basic blocks of the program whose execution counts are independent of the program input. Once this is found, we use the SimpleScalar profiler to find out the execution counts of these input independent basic blocks.

  In addition, Chronos allows the user to input constraints such as loop bounds and some other flow facts through a graphical interface, and these constraints are called "user constraints". This step corresponds to program path analysis (see Figure 1).
- Based on the processor model, which can be configured by user via the tool's graphical interface, Chronos performs micro-architectural modeling (see Figure 1). This yields (1) time bounds for each basic block's execution under certain execution contexts; (2) constraints on the occurrences of these execution contexts (instruction cache state, branch prediction information etc.). These are shown as "Micro-architectural Cons." (constraints introduced by micro-architectural modeling) in Figure 1. Combined with the flow constraints and user constraints, a complete Integer Linear Programming (ILP) problem is formulated by Chronos.
- The tool invokes either *CPLEX* [5], a high-performance commercial ILP solver, or *lp_solve* [2], a free Linear Programming solver, to solve the ILP problem. This yields the *Estimated WCET*. It is important to note here that even though ILP constitutes the back-end of Chronos, this tool is not an ILP-only one like Cinderella [13]. The core of our micro-architectural modeling method is achieved by a fixed-point analysis of pipeline/cache behavior.
- In addition to the estimated WCET, an observed WCET can be obtained via simulation using the SimpleScalar toolset with the same processor configuration as that used in estimation. This yields the *Observed WCET*, which can then be compared against the *Estimated WCET* (produced by micro-architectural analysis and ILP solving).

4 Sample Usage

We now walk-through a sample session in Chronos to describe the usage of the tool. The example is `insertsort` taken from the WCET benchmark set maintained by the Mälardalen WCET research group [21]. Figure 2 shows the dialog for selecting `insertsort`.

Step 1: Compilation and disassembling. The source code of `insertsort` is shown on the second pane in Figure 3 (some irrelevant lines in the source

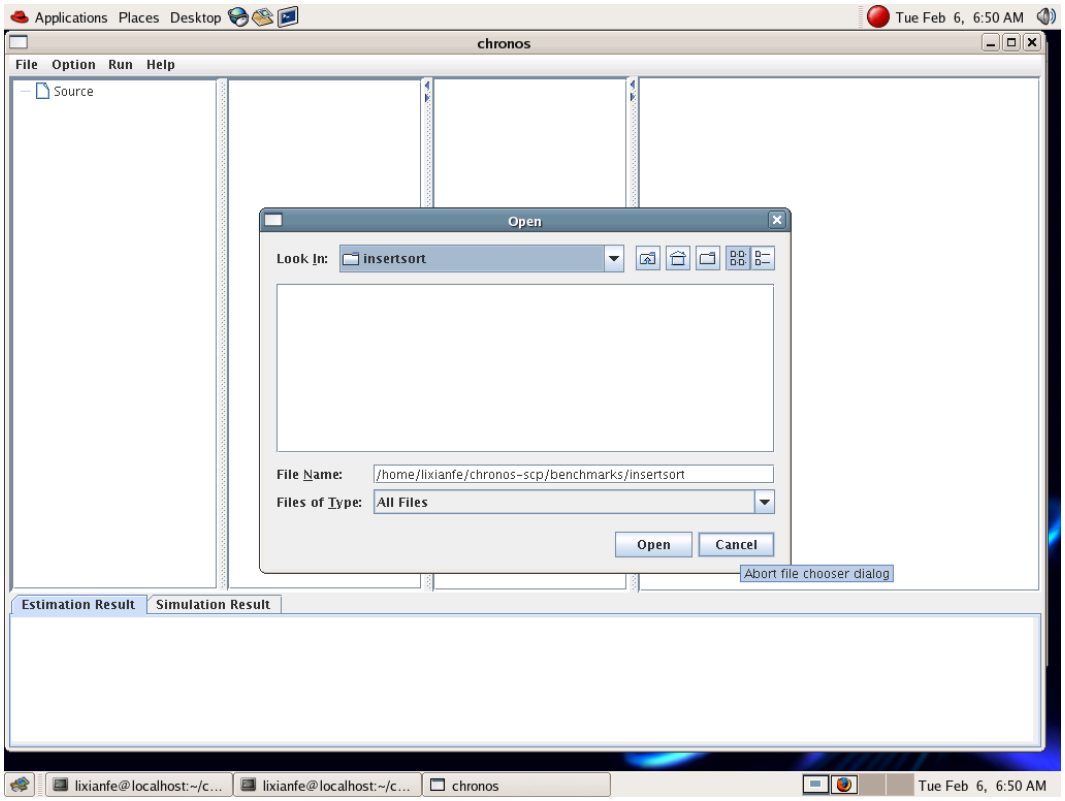

Fig. 2. Chronos: benchmark selection

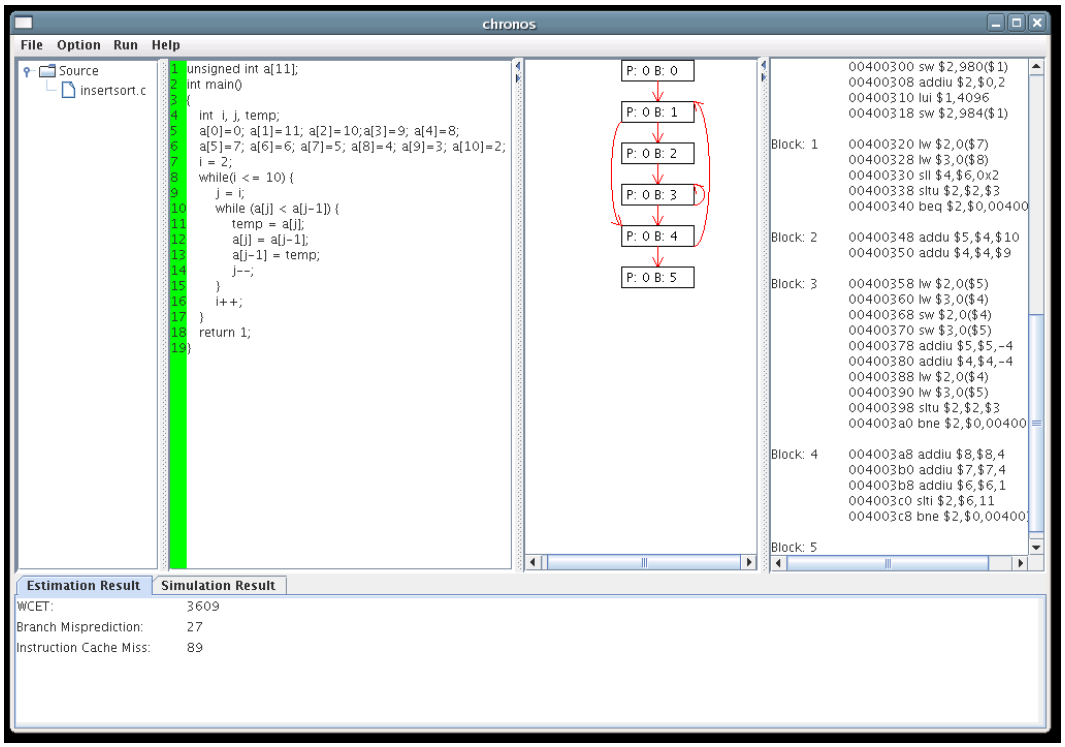

Fig. 3. Chronos: main window

code have been removed for better view). Once `insertsort` is loaded, SimpleScalar GCC is invoked by Chronos for compilation. Subsequently, the assembly code (shown on the right pane in Figure 3) is obtained by objdump in SimpleScalar toolset. The association between source code and assembly code is provided by the SimpleScalar toolkit. It is very useful in practice, since it allows the user to add functional constraints about the program at the source code level. These constraints automatically get translated to constraints on execution of basic blocks in the assembly-level control flow graph.

Step 2: Path analysis. Chronos then performs program path analysis. It constructs `insertsort`'s control flow graph (shown on the third pane in Figure 3), and formulates a set of control flow constraints. These constraints, as part of the final ILP file `insertsort.lp` , are shown in Figure 4(a). For example, the line $b4-d4\_5-d4\_1 = 0$ captures the flow constraint that the execution count of basic block 4 (denoted as $b4$) is equal to its outgoing flows ($d4\_5$ and $d4\_1$ denote the transfers from block 4 to block 5 and block 1 respectively). Similarly, the line $b4 - d1\_4 - d3\_4 = 0$ captures the flow constraint that the execution count of basic block 4 is equal to its incoming flows. Note that the block identifiers in Figure 4(a) actually denote "transformed blocks". Chronos transforms the individual control flow graphs of each procedure into a global control flow graph by traversing the procedure call graph of the program.

Chronos also performs a light-weight data flow analysis to discover additional constraints like *input-independent loop bounds*. In this case, it finds that the outer loop iterates nine times, and generates a constraint $b1 = 9$ accordingly. The number of iterations of the inner loop, however, is dependent on the content of the array to be sorted. Therefore, the user needs to provide an upper bound for the inner loop. Chronos allows the user to give constraints at the source code level. Suppose the user can determine that the inner loop is entered no more than 45 times, he or she then gives such a constraint: $line_{10} <= 45$. This constraint will be converted into a basic block level constraint $b3 <= 45$ by Chronos. Note that the basic block level constraint refers to the assembly level control flow graph, so we need to consider the mapping between source and assembly code for this step.

The following steps correspond to microarchitecture modeling, which are decomposed into branch prediction analysis, instruction cache analysis, and pipeline analysis.

Step 3: Branch prediction analysis. Given the branch predictor configuration, Chronos performs branch prediction analysis based on the technique in [18] to bound the number of mispredictions at each branch. Figure 4(b) shows a small part of the branch prediction constraints. For example, the line

$d1\_2 - dc1\_2 - dm1\_2 = 0$ means that the control flow transfers from block 1 to block 2 can be divided into two cases: the branch at the end of block 1 is correctly predicted (denoted by $dc1\_2$) or mispredicted (denoted by $dm1\_2$). The other branch prediction constraints that further bound $dc1\_2$ and $dm1\_2$ are not shown in Figure 4(b). The reader is referred to our earlier paper [18] for the technical details.

Step 4: Instruction cache analysis. Given the instruction cache configuration, Chronos performs instruction cache analysis based on the technique in [12] to bound the number of cache misses. This basically involves a static categorization of the memory blocks into "cache hit" or "unknown". To reduce the pessimism in such categorization we categorize a memory block under different control flow contexts (such as different levels of loop nest). The static cache behavior categorization affects the ILP constraints for WCET calculation, which are shown in Figure 4(c). For example, $b3 - b3.0 - b3.1 - b3.2 = 0$ means the execution count of block 3 can be divided into three scenarios in terms of cache misses of memory blocks (a memory block is a sequence of instructions in a basic block that are mapped into the same cache line). The occurrence counts of these three scenarios are given by $b3.0$, $b3.1$, $b3.2$. Roughly speaking, a cache scenario of a basic block corresponds to hit/unknown categorization of each of its memory blocks. The reader is referred to [12] for further details.

The variable $d2\_3.0$ in $b3.0 - d2\_3.0 - d3\_3.0 = 0$ denotes the count of control flow transfers from block 2 to block 3 with the cache miss scenario corresponding to $b3.0$. The other constraints that further bound variables like $b3.0$, $d2\_3.0$ are not shown in Figure 4(c). The reader is referred to our [12] for technical details of our cache modeling.

Step 5: Pipeline analysis. Given the pipeline configuration, the results of branch prediction analysis and instruction cache analysis, Chronos performs pipeline analysis to estimate the execution time upper bounds of basic blocks. In fact, since a basic block's pipeline schedule varies significantly with branch prediction and/or cache behavior, it is necessary to consider branch prediction/cache while performing pipeline analysis of a basic block.

Figure 4(d) is the result of our pipeline analysis. The whole expression denotes the overall execution time of `insertsort`, where each term is the contribution of one basic block. For example, in the term "1 $dc1\_2.0$", the variable $dc1\_2.0$ denotes the execution count of basic block 2 under the following context: (1) it is reached from block 1; (2) the branch at the end of block 1 is correctly predicted; and (3) it is executed under cache miss scenario 0 of block 2. The coefficient, 1, is the predicted execution time upper bound of block 2 under this context. These coefficients are provided by our pipeline modeling.

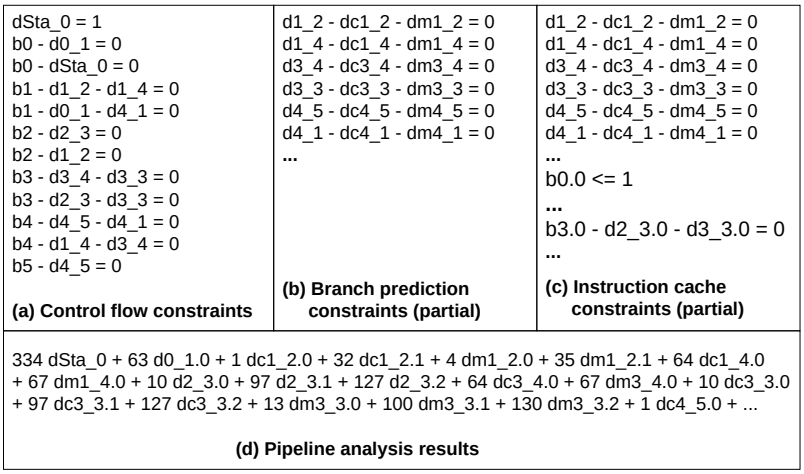

Fig. 4. Fragment of ILP problem constructed for the `insertsort` program

The final ILP file, `insertsort.lp`, contains the following.

- Flow constraints
- Loop bounds inferred and/or provided by user
- Instruction cache and branch prediction constraints
- Objective function denoting execution time of the program (this uses the results from pipeline analysis)

Now Chronos submits `insertsort.lp` to lp_solve, a publicly available linear programming solver, and dumps out the WCET returned by lp_solve, as shown on the bottom pane in Figure 3. Chronos also supports the commercial ILP solver CPLEX for the back-end constraint solving. CPLEX allows for much more efficient WCET calculation and thus provides a more scalable back-end to Chronos.

5 The WCET Tool Challenge Experience for Chronos Team

In September 2006, a tool challenge was organized among the WCET analysis tools. The effort was initiated and conducted by a working group based on discussions at the Timing Analysis group under the ARTIST2 Network of Excellence for Embedded Systems. The purpose of the challenge was to test the maturity of state-of-the-art WCET analysis tools. Thus, the event was in the true sense "WCET Tool Challenge" rather than "WCET Tool Competition". The results of the event were discussed and published at the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA) in November 2006. Chronos was one of the five entrants to this challenge. In this section, we articulate some of the experiences gained from this effort. We briefly describe the results from Chronos, and the

lessons we learnt. The interested reader is referred to [6] for more details.

The Challenge consisted of checking each WCET tool against fifteen medium sized benchmarks and two programs from a fly-by-wire application, resulting in a total of seventeen programs. The medium-sized benchmarks were drawn from the Mälardelaen benchmark suite [21] and the fly-by-wire application was taken from the Papabench suite. [3] For each program, the tools were tried in several modes such as — no annotations (analysis proceeds completely automatically), minimal annotations (the user provides basic information such as loop bounds) and maximal annotations (the user can provide any annotation provided the WCET analysis tool can exploit it for estimation). *The tools were tested on the given benchmarks by an independent evaluator employed by the WCET working group.*

We found that Chronos successfully estimated fifteen (15) out of the seventeen (17) programs. The estimation was done of various processor configurations — processors with no cache and perfect branch prediction, processors with cache and global branch prediction etc. Two of the seventeen benchmarks could not be handled by Chronos. One of them is because Chronos cannot handle recursive programs, except those programs where a recursive procedure calls itself. The other program's estimation was not considered because of rather specific technical reasons (the Graphical User Interface of our tool was not equipped to handle compilation of programs where all the program files were not in the same directory).

The results from the Tool Challenge gave us some useful lessons, as well as confidence in our tool. We found that Chronos is one of the only two tools (the other one being the commercial tool aiT) which provides *Observed WCET* as well as *Estimated WCET* values. The other tools simply provide the *Estimated WCET* and hence there is no way to gauge how tight the estimate is. The WCET analysis times from Chronos were also well within acceptable limits (typically few seconds), making it feasible to integrate such tools into a compiler infrastructure.

Overall, we found that the loop bound inferencing in our tool needs improvement. Since loop bounds are required for WCET analysis, any loop bound that cannot be automatically inferred by Chronos needs to be provided by the user as annotation. In future, we are planning to hook up Chronos with an external constraint solver which will perform an offline loop bound inferencing along the lines of [8].

[3] See (`http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97`

6 Related Work — Existing WCET Analysis tools

There exist some commercial and research prototype tools for WCET analysis. However, with the exception of one or two, none of these tools are open source. Moreover, most of these tools cannot model complex architectural features that are becoming common in modern embedded processors (e.g., out-of-order execution and dynamic branch prediction) for timing analysis.

The most well-known and extensively used commercial WCET analyzer is the aiT tool [1] from AbsInt Angewandte Informatik, and it is the only WCET analyzer that has been used to model out-of-order execution. Here we take an overview of its underlying techniques, and show how the same example – `insertsort` would work on aiT. The discussion is based on information from aiT website and a few technical publications [10,23]. This tool also derives an upper bound on the execution time of each basic block, and uses these upper bounds to find the program's WCET by solving an ILP problem. The microarchitectural modeling technique, however, is different from ours. What aiT uses, as suggested by its name, is the theory of abstract interpretation [4], which conservatively derives program properties (in this case, properties about cache behavior and so on). In the following we describe how aiT would work on `insertsort` targeted to an out-of-order processor. This will expose some key differences between the two analyzers.

Like Chronos, aiT works on program binary. It first takes in the binary of the program, and reconstructs the control flow graph (CFG); this corresponds to Step 2 in Section 4. It then conducts an analysis called *value analysis*, which statically computes the ranges of data values and addresses. This information is used for data cache analysis to identify non-conflicting memory accesses. Since data cache is not modeled in Chronos, it does not have a corresponding step. Next, aiT performs *loop bound analysis*, which determines upper bounds for simple loops. Conceptually, this is very similar to our light-weight data flow analysis, and we expect it to discover the bound on `insertsort`'s outer loop as Chronos does.

When it comes to micro-architecture modeling, the two analyzers (Chronos and aiT) work very differently. First, we have not seen any detailed discussion on branch prediction analysis in aiT – Step 3 in Chronos. In particular, for *dynamic branch prediction*, we do not know how the number of mispredictions is bounded via abstract interpretation in aiT. We conjecture that statically classifying (via abstract interpretation) the three conditional branches of `insertsort` as either mispredicted or correctly predicted might be too coarse.

In aiT, cache analysis (Step 4 in Chronos) and pipeline analysis (Step 5 in Chronos) can be performed either in separation or in integration. The work

[23] presents a separated approach which is used to model the PowerPC 755 processor, while [10], which models the same processor, describes an integrated approach. The integrated approach is able to achieve better accuracy at the expense of much higher complexity. Overall, aiT's instruction cache analysis is not significantly different from that of Chronos – both tools try to classify for each instruction whether it is always a hit in some execution context, or otherwise.

What is fundamentally different in the two tools (Chronos and aiT) is the pipeline analysis. At the basic block level, aiT estimates the execution time of a basic block via cycle-wise evolution of pipeline states along program points. Although pipeline state abstraction w.r.t. timing-irrelevant aspects (*e.g.*, register values) is enforced, it still has to maintain a large number of pipeline states. Furthermore, for WCET estimation, aiT maintains the pipeline states for each basic block. The tool iteratively populates/updates these states until there are no changes to the pipeline state space of any basic block. For complicated pipelines, this approach suffers from a huge state space explosion. As reported in [23], when working on a 3+ GHz Pentium 4, aiT spent 12 hours per task on average for the avionics program, and its memory consumption was close to 3 Gbytes – nearing the limit of current 32-bit architectures.

Since aiT is not an open-source software, and SimpleScalar processor model is not in the supported list, we are unable to conduct a quantitative comparison between aiT and Chronos. However, it should be noted that aiT is a commercial tool coming with a number of powerful supporting tools like aiSee, PAG, StackAnalyzer, etc. These supporting tools not only help in WCET analysis but also in performance debugging and visualization.

Another commercial WCET analyzer is the Bound-T tool [15] which also works on program binary. It concentrates mainly on program path analysis and does not model cache, complex pipeline and/or branch prediction. Even in path analysis, the main focus of the tool is in inferring loop bounds – for which it depends heavily on user assertions. Bound-T has been targeted towards Intel 8051 series micro-controllers, Analog Devices ADSP-21020 DSP, and ATMEL ERC32 SPARC V7-based platforms. Again, unlike Chronos, Bound-T is not open-source.

Among the research prototypes, HEPTANE [19] is an open-source WCET analyzer. HEPTANE models in-order pipeline, instruction cache and branch prediction; but it does not include any automated program flow analysis. Symta/P [24] is another research prototype that estimates WCET for C programs. It models caches and simple pipeline; but does not support modeling of complex micro-architectural features such as out-of-order pipelines and branch prediction. The SWEET tool [7] primarily focuses on loop bound inferencing and flow analysis. Cinderella [13] is an Integer Linear Programming (ILP)

Tool	**Caches**	**Pipeline**	**Periphery**
aiT	I/D cache, direct/SA, LRU, PLRU, pesudo round robin	in-order/out-of-order	PCI bus
Bound-T	-	in-order	-
SymTA/P	I/D cache, direct/SA, LRU	-	-
Heptane	I-cache, direct/SA, LRU, locked cache	in-order	-
Vienna	jump-cache	simple in-order	-
SWEET	I-cache, direct/SA, LRU	in-order	-
Florida	I/D cache, direct/SA	in-order	-
Chalmers	split L1 SA, unified L2 cache	multi-issue superscalar	-
Chronos 2.0	I-cache, direct/SA, LRU	multi-issue superscalar, in-order/out-of-order, dynamic branch prediction	-

Table 1
Comparison of WCET tools in terms of support for architectural features (SA stands for set associative).

based research prototype developed at Princeton University. The main distinguishing feature of this tool is that it performs both program path analysis and micro-architectural modeling by solving a (huge) ILP problem. However, this formulation makes the tool less scalable because the ILP solving time does not always scale up for complex micro-architectures. Also, Cinderella mostly concentrates on program path analysis and cache modeling; it does not analyze timing effects of complex pipelines and branch prediction.

Apart from the above-mentioned tools, several other research groups have developed their own in-house timing analysis prototypes incorporating certain novel features (*e.g.*, TU Vienna research prototype, Chalmers research prototype). One notable effort is by the research group in Florida State University. Their work involves sophisticated flow analysis for inferring infeasible path patterns and loop bounds [9]. However the tool is currently not available for use/download, that is, it is an in-house research effort.

Table 1 taken from [25] compares the architectural features supported by different static analysis based WCET tools. Chronos 2.0 and aiT are the only two WCET tools that support complex out-of-order pipeline. Chronos also accurately models different dynamic branch prediction schemes. Furthermore,

Chronos and aiT are the only two tools which give both estimated and observed WCET, thereby providing an idea about the estimation tightness.

7 Discussion

In this paper, we have presented Chronos, a Worst-case Execution Time (WCET) analysis tool for real-time embedded software. It takes as input the program binary, disassembles it and performs static analysis on the assembly code. The static analysis involves program flow analysis as well as micro-architectural modeling. Currently, there exist commercial tools as well as research prototypes for WCET analysis. The main distinguishing features of Chronos w.r.t. these tools are as follows.

- Chronos is *open-source* unlike most existing WCET analyzers. The user can change the micro-architectural analysis routines to model new processor platforms. Thus, he/she can always re-use the routines doing standard stuff — control flow graph extraction, disassembly etc. Moreover, if a new processor has a completely different pipeline structure (for example) we can change the core pipeline analysis with minimal change to the cache/branch prediction analysis. In contrast, most existing WCET analyzers only allow a user to set architectural parameters in a limited way (*e.g.* set the cache line size, cache associativity etc.)
- Chronos is built on the top of the popular Simplescalar architectural simulator. Simplescalar allows the user to flexibly model different architectures for simulation. By building Chronos on top of Simplescalar, we provide the users a similar advantage but for timing analysis — he/she can quickly model different processor platforms and perform WCET estimation of a given application.
- Last but certainly not the least, Chronos supports accurate modeling of *advanced micro-architectural features* such as out-of-order pipelines and dynamic branch prediction (both local and global). This increases the scope of the tool's applicability — the timing effects of more processor platforms can be estimated by Chronos.

The Chronos tool is available from

`http://www.comp.nus.edu.sg/~rpembed/chronos`

In future releases, we plan to enhance the tool by working on program path analysis (more powerful loop bound inferencing) as well as micro-architectural modeling (modeling of new features like data caches).

Acknowledgments We would like to thank Vivy Suhendra for her help during the WCET Tool Challenge 2006. This work was partially supported by University Research Council (URC) project R252-000-171-112 from National University of Singapore (NUS).

References

[1] AbsInt, aiT: Worst case execution time analyzer, `http://www.absint.com/ait/` (2005).

[2] M. Berkelaar, lp_solve: (Mixed Integer) Linear Programming Problem Solver, Available from `ftp://ftp.es.ele.tue.nl/pub/lp_solve`.

[3] D. Burger, T. Austin, The SimpleScalar Tool Set, Version 2.0, Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, `http://www.simplescalar.com/docs/users_guide_v2.pdf` (Jun. 1997).

[4] P. Cousot, R. Cousot., Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints., in: ACM Symposium on Principles of Programming Languages, 1977.

[5] CPLEX, The ILOG CPLEX Optimizer v7.5, commercial software, `http://www.ilog.com` (2002).

[6] J. Gustafsson, The WCET Tool Challenge 2006, in: B. S. Tiziana Margaris, Anna Philippou (ed.), 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06), 2007.

[7] J. Gustafsson, et al., Swedish execution time tool (SWEET) (2006).

[8] C. Healy, M. Sjodin, V. Rustagi, D. Whalley, R. Engelen, Supporting timing analysis by automatic bounding of loop iterations, Real-Time Systems 18 (2/3).

[9] C. Healy, D. Whalley, Automatic detection and exploitation of branch constraints for timing analysis, IEEE Transaction on Software Engineering 28 (8).

[10] R. Heckmann, M. Langenbach, S. Thesing, R. Wilhelm, The Influence of Processor Architecture on the Design and the Results of WCET Tools, Proceedings of the IEEE 91 (7).

[11] X. Li, T. Mitra, A. Roychoudhury, Modeling control speculation for timing analysis, Journal of Real-Time Systems 29 (1).

[12] X. Li, A. Roychoudhury, T. Mitra, Modeling out-of-order processors for WCET analysis, Journal of Real-Time Systems 34 (3).

[13] Y.-T. S. Li, Cinderella 3.0 WCET analyzer, `http://www.princeton.edu/~yauli/cinderella-3.0/` (1996).

[14] Y.-T. S. Li, S. Malik, Performance Analysis of Real-time Embedded Software, Kluwer Academic Publishers, 1999.

[15] T. Ltd., Bound-t execution time analyzer, http://www.bound-t.com (2005).

[16] T. Lundqvist, P. Stenström, Timing anomalies in dynamically scheduled microprocessors, in: IEEE Real-Time Systems Symposium, 1999.

[17] T. Mitra, A. Roychoudhury, Worst Case Execution Time and Energy Analysis, in: Y. Srikant, P. Shankar (eds.), The Compiler Design Handbook, CRC Press, 2007.

[18] T. Mitra, A. Roychoudhury, X. Li, Timing analysis of embedded software for speculative processors, in: ACM SIGDA International Symposium on System Synthesis (ISSS), 2002.

[19] I. Puaut, Heptane static WCET analyzer, http://www.irisa.fr/aces/work/heptane-demo/heptane.html (2003).

[20] P. Puschner, C. Koza, Calculating the maximum execution time of real-time programs, Journal of Real-time Systems 1 (2).

[21] M. research group, WCET benchmark programs, http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[22] A. Shaw, Reasoning about time in higher level language software, IEEE Transactions on Software Engineering 1 (2).

[23] J. Souyris, E. Pavec, G. Himbert, V. Jegu, G. Borios, R. Heckmann, Computing the worst case execution time of an avionics program by abstract interpretation, http://www.absint.com/aiT_airbus.pdf.

[24] J. Staschulat, Symta/p: Symbolic timing analysis for processes, http://www.ida.ing.tu-bs.de/research/projects/symta/home.e.shtml (2005).

[25] R. Wilhelm, The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools, ACM Transactions on Embedded Computing Systems(To appear).

[26] R. Wilhelm, Why AI + ILP is good for WCET, but MC is not, nor ILP alone, in: Verification, Model Checking and Abstract Interpretation (VMCAI), LNCS 2937, 2004.