Cinderella: A Retargetable Environment for Performance Analysis of Real-Time Software

Yau-Tsun Steven Li[1], Sharad Malik[2], Andrew Wolfe[2]

[1] Hewlett-Packard Company, 1501 Page Mill Rd, MS 6U-J, Palo Alto, CA 94304, USA
[2] Dept of Electrical Engineering, Princeton University,
Princeton, NJ 08544, USA.

Abstract. Real-time systems are characterized by the presence of timing constraints that a task must be completed within a given deadline. In this paper, we present a complete environment for determining best-case and worst-case execution time of a program when running on a given hardware. Our analysis technique is unique in that it allows user to annotate complex program path information and at the same time, models cache memory and pipeline accurately. This results in tight estimations even for complicated programs running on modern hardware. The technique has been implemented on a timing analysis tool — `cinderella`[3], which provides retargetable back-ends for analyzing programs written in different languages and executed on different hardware. We present some experimental results of using this tool.

1 Introduction

The execution time of a program running on a given system may vary significantly according to different input data and initial system states. In many cases it is essential to determine the extreme case (best case or worst case) execution time of a program. This information is needed in many real-time operating systems for task scheduling. It is also needed in the hardware/software partitioning step in embedded system designs.

The *actual* extreme case execution time of a program cannot be determined unless all feasible input data and system states are simulated. Since a large number of simulations is required, this method is impractical. Instead, our objective is to determine a *tight bound* on all feasible execution times of a program. This bound is denoted as the **estimated bound** of the program. We tackle the problem of determining tight estimated bound by dividing it into two smaller ones:

Program path analysis This analyzes the structure of the program statically and determines the set of paths that corresponds to the extreme case program execution time. Since the number of program's feasible paths is in general exponential in its size, no path enumeration is allowed in this analysis. Also, many of the statically feasible program paths are never executed in practice. A mechanism for the programmer to mark infeasible paths is essential in tightening the estimation.

Microarchitecture modeling This determines the extreme case execution times of known sequences of instructions and passes them to program path analysis. The presence of modern microarchitecture features, such as pipelines and caches, varies the instruction execution time significantly. The instruction execution time is no longer constant and it depends on the execution trace. Hence, this analysis interferes with program path analysis.

[3] In recognition of her hard real-time constraint — she had to be back home at the stroke of midnight!

Both problems are *equally important* in determining tight estimated bound of the program. In solving the above problems, we also need to consider the retargetability issues so that the solution can be applied to a wide range of programs running on different hardware. In the following, we will describe our analysis in determining the estimated worst case execution time (WCET) of the program. The analysis for the best case execution time is similar.

2 Related Work

Early researchers [11, 15, 17, 18, 20] in this area adopted simple microarchitecture modeling where instruction execution times are assumed to be constant and independent of each others. They focused on program path analysis and proposed different techniques to eliminate false program paths. In particular, Park [17] recognized the use of regular expressions to annotate various path information. However, the analysis of regular expression is complicated and some pessimism approximations are used.

More recently, many techniques have been proposed to model pipelines [4, 7, 13, 16, 21] and caches [2, 3, 9, 13, 14, 19] with various success. In modeling these microarchitecture features, the importance of path annotation are neglected. As a result, these techniques could only handle simple programs with fixed loop bounds (e.g. matrix multiplication routines). For more complicated programs, like sorting routines, they generated loose estimations.

Most researchers claimed their methods to be retargetable. However, there are no timing analysis tools that actually implement the retargetable framework. Only a few retargetable tools exist in pipeline modeling [5, 16].

3 Program Path Analysis

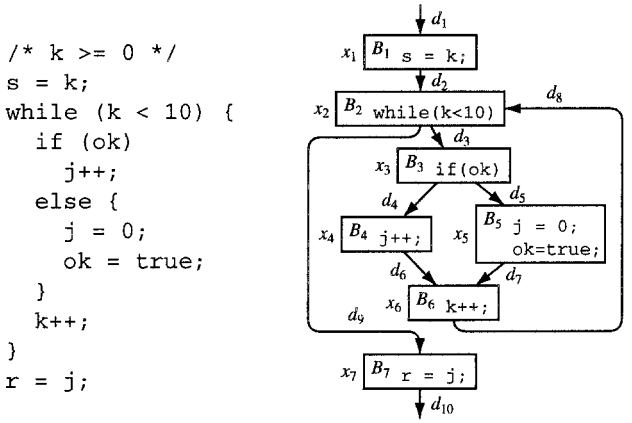
In this analysis, we assume that instruction execution times are all constant. This assumption will be removed in Sect. 4. Our analysis technique uses the counting approach to compute the estimated WCET. The method converts the problem of solving the estimated WCET into a set of **integer linear programming** (ILP) problems in which the estimated WCET, and the worst case execution counts of the instructions are solved for.

For each basic block [1] B_i in the program, we let variable x_i be its execution count and constant c_i be its single execution time. For program with N basic block, the total execution time is:

$$\text{Total execution time} = \sum_{i=1}^{N} c_i x_i. \tag{1}$$

The possible values of x_i's are constrained by the program structure and the program input data. This constraints are represented by a set of linear constraints. The linear constraints are divided into two parts: (i) **structural constraints**, which are derived automatically from the program's control flow graph (CFG) [1], and (ii) **functionality constraints**, which are provided by the user to specify loop bounds and other path information. Fig. 1 shows a simple code fragment and its CFG. Each edge in the CFG is labeled with a variable d_i which serves both as a label for that edge and as a count of the the number of times that the program control passes through that edge. Analysis of the CFG is equivalent to a standard network-flow problem. Structural constraints can be derived from the CFG from the fact that, for each node B_i, its execution count is equal to the number of times that the control enters the node (inflow), and is also equal to the number of times that the control exits the node (outflow):

$$x_i = \sum d\_inflow = \sum d\_outflow \tag{2}$$



(a) Code (b) Control flow graph

Fig. 1. An example showing how the structural and functionality constraints are constructed.

The loop bound information must be provided by the user using functionality constraints. Otherwise, the estimated WCET is unbounded. In this example, since k is positive before it enters the loop, the loop body will be executed at most 10 times each time the loop is entered. This information can be represented by the functionality constraints: $0x_1 \leq x_3 \leq 10x_1$.

Additional path information can also be described by functionality constraints. We have been able to show that the functionality constraints are more powerful than Park's IDL [17] in describing path information [12]. As a simple example, the `else` statement (B_5) can be executed at most once inside the loop. This information can be specified as: $x_5 \leq 1x_1$.

4 Microarchitecture Modeling

Microarchitecture modeling models CPU pipeline and cache memory — two dominant microarchitecture features that affect the execution time of an instruction. In modeling these features, our analysis technique is unique in that the program path analysis model described in previous section is retained.

4.1 Pipeline Modeling

The pipeline modeling is relatively easy and straightforward. We model the pipeline within each basic block and add up the execution time each instruction spent in the execution stage of the pipeline [8]. In determining the estimated WCET of the basic block, we assume the pipeline is flushed at the end of the basic block. This model is used by many researchers [4, 16, 21]. Our experiments in modeling the pipeline of Intel i960KB processor showed that it is accurate. Detailed results will be given in Sect. 6.

4.2 Cache Modeling

Cache is much harder to model than pipeline. In a pipeline, the instruction execution time depends only on a few of its preceding instructions. But with the presence of cache memory, it depends on all instructions that are mapped to the same cache set. A *global* analysis is required

for accurate cache modeling. If not properly modeled, the cache analysis will introduce more pessimism than the pipeline analysis. Direct mapped instruction cache analysis will be described first. This is followed by set associative instruction cache analysis.

The goal of cache modeling is to determine for each instruction, the number of fetches that result in cache hits/misses. We first partition each basic block into smaller units (called *l-blocks*) that are aligned with the instruction cache line. Suppose a basic block B_i is partitioned into n_i l-blocks, they are denoted as $B_{i.1}, B_{i.2}, \ldots, B_{i.n_i}$. Since the cache controller always fetches a line of code whenever this is a miss, an l-block $B_{i.j}$ is either in the i-cache completely, or not in it at all. These two cases correspond to two possible execution times of the l-block, which are represented by constants $c_{i.j}^{hit}$ and $c_{i.j}^{miss}$ respectively. We let $x_{i.j}^{hit}$ and $x_{i.j}^{miss}$ be integer variables that represent an l-block $B_{i.j}$'s hit and miss counts. Given these variables, the total execution time of the program can be refined as:

$$\text{Total execution time} = \sum_{i=1}^{N} \sum_{j=1}^{n_i} (c_{i.j}^{hit} x_{i.j}^{hit} + c_{i.j}^{miss} x_{i.j}^{miss}). \tag{3}$$

Since l-block $B_{i.j}$ is inside the basic block B_i, its total execution count is equal to x_i. Hence

$$x_i = x_{i.j}^{hit} + x_{i.j}^{miss}, \qquad j = 1, 2, \ldots, n_i \tag{4}$$

Eq. (4) links the new cost function (3) with the structural constraints and the functionality constraints, both of which remain unchanged. In addition, the cache activities can now be described in terms of the new variables $x_{i.j}^{hit}$'s and $x_{i.j}^{miss}$'s.

For any two l-blocks mapped to the same cache set, we say that they *conflict* with each other if their address tags [8] are different. Otherwise, they are called *non-conflicting* l-blocks.

Direct Mapped Instruction Cache Analysis Consider a simple case. For each cache set, if there is only one l-block $B_{k.l}$ mapped to it, then only its first execution *may* result in a cache miss, therefore,

$$x_{k.l}^{miss} \leq 1. \tag{5}$$

When a cache set contains two or more conflicting l-blocks, the hit/miss counts of all the l-blocks mapped to this set will be affected by the execution sequence of these l-blocks. In this case, a **cache conflict graph** (CCG) [12] is constructed. A CCG captures the control flow of a set of l-blocks mapped to the same cache set. Each edge of CCG is labeled with a p variable to count the number of times that the control passes through that edge. Suppose that in Fig. 1(b), basic blocks B_1, B_4 and B_5 are partitioned into l-blocks and l-blocks $B_{1.1}$, $B_{4.1}$ and $B_{5.1}$ are mapped to the same cache set and they conflict with each other, the CCG is shown in Fig. 2(a). The control flow from one l-block $B_{i.j}$ to the other $B_{k.l}$ is represented by a variable $p_{(i.j,\ k.l)}$.

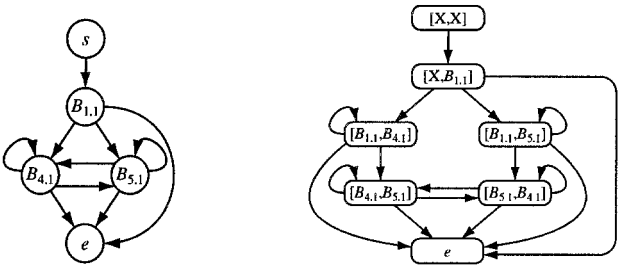
A set linear constraints can be derived from the CCG to link with the structural and functionality constraints, and also to describe cache hit/miss constraints. At each node $B_{i.j}$, the sum of control flow going into the node must be equal to the sum of control flow leaving the node, and it must also be equal to the total execution count of l-block $B_{i.j}$. Therefore, two constraints are constructed at each node $B_{i.j}$:

$$x_i = \sum p\_inflow = \sum p\_outflow \tag{6}$$

Due to the existence of x_i's, this set of constraints is linked to the structural and functionality constraints.

All self loops in CCG indicate cache hits. Therefore,

$$x_{i.j}^{hit} = p_{(i.j,\ i.j)} \tag{7}$$



(a) Cache Conflict Graph (b) Cache State Transition Graph

Fig. 2. The CCG captures control flow of l-blocks mapped to the same cache set. The CSTG, shown in 2-way set associative i-cache, represents all feasible cache states and their transitions due to the flow of these l-blocks. In this example, the graphs are constructed when l-blocks $B_{1,1}$, $B_{4,1}$ and $B_{5,1}$ in Fig. 1 conflict with each other.

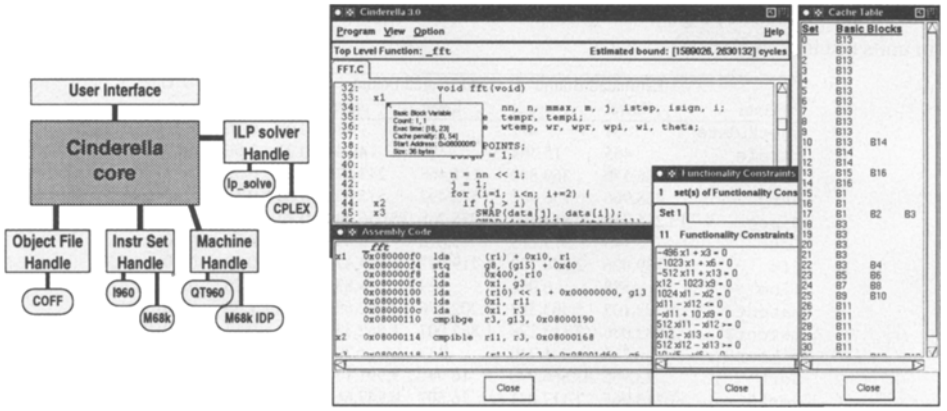
The above linear constraints are the **cache constraints** for direct mapped i-cache. These constraints, together with (4), the structural constraints and the functionality constraints, are passed to the ILP solver with the goal of maximizing the cost function (3). Because of the cache information, a tighter estimated WCET will be returned. The CCGs are network flow graphs and thus the cache constraints are typically solved rapidly by the ILP solver. For programs with function calls, the functions are treated as if they are inlined [12].

Set Associative Instruction Cache Analysis The modeling of set associative i-cache is very similar to that of direct mapped i-cache. Only the direct mapped i-cache constraints (5)–(7) are replaced by a set of new ones.

A **cache state transition graph** (CSTG) [12] is constructed to model all feasible cache state transitions of a cache set. Each node of the graph contains a state $[B_{i.j}, B_{m.n}]$ showing two l-blocks in the least and most recently used entries. For 2-way set associative i-cache with least recently used (LRU) replacement policy, a CSTG is shown in Fig. 2(b). A transition from state $[B_{i.j}, B_{k.l}]$ to $[B_{k.l}, B_{m.n}]$ represents an execution of l-block $B_{m.n}$. Similar to the CCG, self loops in CSTG represent cache hits. In addition, transition from $[B_{i.j}, B_{m.n}]$ to $[B_{m.n}, B_{i.j}]$ also results in cache hits. A p-variable is associated with each edge of the graph to represent the number of transition. Based on x's and p's, a new set of flow equations and cache hit linear constraints can be generated.

5 Implementation and Retargetability Issues

The above analysis technique has been implemented in our timing analysis tool `cinderella`. The tool features several retargetable back-ends (Fig. 3(a)) so that it can be easily ported to model different hardware, as well as programs written in different source language. The core performs program path analysis and cache analysis. The *object file handler* reads the executable file of the program directly and utilizes debugging information to map binary code to source code so that path information can be entered at source level. This approach allows programs written in different languages and compiled by different compilers to be analyzed. The *instruction set handler* decodes the binary code and passes information to the core for building control flow graph and also to the *machine handler*, which models instruction pipelines and provides instruction timings and cache configurations to the core. The separation of instruction set decoding and machine



(a) Block diagram (b) Snapshot

Fig. 3. `Cinderella`'s retargetable back-ends and its graphical user interface.

Table 1. Set of benchmark examples, their descriptions, source file line sizes and Intel i960KB binary code sizes.

Program	Description	Lines	Bytes
`check_data`	Check if any element in an array is negative, from Park [17]	23	88
`circle`	Circle drawing routine, from Gupta [6]	100	1,588
`des`	Data Encryption Standard	192	1,852
`dhry`	Dhrystone benchmark	761	1,360
`djpeg`	Decompression of 128×96 color JPEG image	857	5,408
`fdct`	JPEG forward discrete cosine transform	300	996
`fft`	1024-point Fast Fourier Transform	57	500
`line`	Line drawing routine, from Gupta [6]	165	1,556
`matcnt`	Summation of 2 100×100 matrices, from Arnold [2]	85	460
`matcnt2`	`Matcnt` with inlined functions	73	400
`piksrt`	Insertion sort of 10 elements	19	104
`sort`	Bubble sort of 500 elements, from Arnold [2]	41	152
`sort2`	`Sort` with inlined functions	30	148
`stats`	Calculate the sum, mean and variance of two 1,000-element arrays, from Arnold [2]	100	656
`stats2`	`Stats` with inlined functions	90	596
`whetstone`	Whetstone benchmark	196	2,760

timing model allows one to model a family of processors easily. Currently, we have implemented modules for modeling programs running on Motorola M68000 and Intel i960KB processors. The tool also features a user-friendly graphical interface (Fig. 3(b)). It can be downloaded from its WWW home page (`http://www.ee.princeton.edu/~yauli/cinderella/`).

6 Experimental Results

We have analyzed a large set of programs to validate our analysis technique. The programs are shown in Table 1. Some of them comes from other researchers. Others are much more complicated software benchmarks and application programs.

Table 2. Benchmark program analysis results. Estimated bounds and measure bounds are shown in units of clock cycles.

	Estimated Bound		Measured bound		Pessimism [a]		CPU Time
Program	lower	upper	lower	upper	lower	upper	(sec.)
check_data	34	471	34	430	0.00	0.10	(0, 0)
circle	465	15,364	585	14,483	0.21	0.06	(0, 0)
des	86,570	369,840	111,468	243,676	0.22	0.52	(6, 4)
dhry	458,966	756,961	575,492	575.622	0.20	0.32	(2, 0)
djpeg	13,225,736	70,414,320	14,975,268	35.636.948	0.12	0.98	(4, 6)
fdct	6,145	9,115	7,616	9.048	0.19	0.01	(0, 0)
fft	1,589,026	2,630,132	1,719,832	2,204,472	0.08	0.19	(0, 0)
line	578	6,088	929	4,836	0.38	0.26	(0, 0)
matcnt	1,722,105	5,463,383	2,202,276	2,202,698	0.22	1.48	(0, 0)
matcnt2	1,482,086	2,113,328	1,862,007	1,862.333	0.20	0.13	(0, 0)
piksrt	236	1,740	337	1,705	0.30	0.02	(0, 0)
sort	13,965	27,866,978	16,942	9,991,172	0.18	1.79	(0, 0)
sort2	13,965	7,117,043	16,507	6,747,664	0.15	0.05	(0, 0)
stats	1,008,085	2,213,764	1,158,142	1,158,469	0.13	0.91	(0, 0)
stats2	894,017	1,235,696	1,060,118	1,060.380	0.16	0.17	(0, 0)
whetstone	5,970,554	10,546,246	6,935,612	6,935,668	0.14	0.52	(0, 0)

[a] Pessimism is calculated as: $lower = \frac{Mea.\ lower - Est.\ lower}{Mea.\ lower}$, $upper = \frac{Est.\ upper - Mea.\ upper}{Mea.\ upper}$.

We studied each program carefully, determined its loop bounds and path information, and used `cinderella` to compute the estimated bound. To validate this estimation, we determined each program's extreme case input data sets and then used logic analyzer to measure the execution time of the program when running on an Intel QT960 evaluation board [10] containing a 20 MHz i960KB processor. The processor has an on-chip 512 bytes direct-mapped i-cache and a 4-stage execution pipeline.

Table 2 shows the results of analysis. All estimated bounds bound their corresponding measured bounds. For most programs, the estimated bound is very close to the measured bound. We have also conducted other experiments to validate program path analysis and cache analysis separately [12]. These experiments indicate that when given enough path information, our analysis technique is very accurate. A few programs have looser estimation. For programs `des` and `djpeg`, this is because the extreme case input data set could not be determined, random input data sets were used and as a result, the measured bound might not be close to the actual bound of the program. For programs `matcnt`, `sort` and `stats`, the reason for loose estimation is that i960KB processor features 4 register windows, which were not modeled in our tool. Conservation assumptions were used in modeling the execution times of call and return instructions and large pessimism will occur for programs with lots of small function calls. We inlined the frequently called functions in these programs. The results are programs `matcnt2`, `sort2` and `stats2`. Their estimated bounds are much tighter. The ILP problems were solved by a commercial ILP solver CPLEX. They were solved efficiently on a Silicon Graphics Indigo2 workstation containing a 150 MHz MIPS R4400 processor with 256 MB main memory.

7 Concluding Remarks

In this paper, we have described an efficient and powerful method based on integer linear programming to determine the execution time bounds of real-time programs. When compared with other existing methods, ours offers more powerful path annotation mechanism and more accurate cache modeling. Even better, the ILP formulation allows both to be applied simultaneously.

This results in very tight estimations even for complicated programs. Our implementation is more complete than others. `Cinderella` features a user-friendly graphical interface and retargetable back-ends. We have conducted extensive experiments to validate that our tool is capable of analyzing large and complicated programs accurately.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *Proc. of the 15th IEEE Real-Time Systems Symposium*, Dec 1994.
3. S. Basumallick and K. Nilsen. Cache issues in real-time systems.. In *Proc. of ACM PLDI Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, Jun 1994.
4. S. Bharrat and K. Jeffay. Predicting worst case execution times on a pipelined RISC processor. Technical report, Dept of Computer Science, University of North Carolina at Chapel Hill, Apr 1994. TR94-072.
5. D. Bradlee. *Retargetable Instruction Scheduling for Pipelined Processors*. PhD thesis, University of Washington, 1991.
6. R. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. PhD thesis, Stanford University, Dec 1993.
7. C. Healy, D. Whalley, and M. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proc. of 16th IEEE Real-Time Systems Symposium*, Dec 1995.
8. J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, 2nd Ed.* Morgan Kaufmann Publishers, Inc., 1996. ISBN 1-55860-329-8.
9. Y. Hur, Y.-H. Bae, S.-S. Lim, S.-K. Kim, B.-D. Rhee, S.-L. Min, C.-Y. Park, M. Lee, H. Shin, and C.-S. Kim. Worst case timing analysis of RISC processors: R3000/R3010 case study. In *Proc. of 16th IEEE Real-Time Systems Symposium*, Dec 1995.
10. Intel Corp. *QT960 User Manual*, 1990. Order Number 270875-001.
11. E. Kligerman and A. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Trans. on Software Engineering*, Sep 1986.
12. Y.-T. Li. *Performance Analysis of Real-Time Embedded Software*. PhD thesis, Princeton University, 1997.
13. S.-S. Lim, Y.-H. Bae, G.-T. Jang, B.-D. Rhee, S.-L. Min, C.-Y. Park, H. Shin, K. Park, and C.-S. Kim. An accurate worst case timing analysis technique for RISC processors. In *Proc. of the 15th IEEE Real-Time Systems Symposium*, Dec 1994.
14. J.-C. Liu and H.-J. Lee. Deterministic upperbounds of the worst-case execution times of cached programs. In *Proc. of the 15th IEEE Real-Time Systems Symposium*, Dec 1994.
15. A. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat. Evaluating tight execution time bounds of programs by annotations. In *Proc. of the 6th IEEE Workshop on Real-Time Operating Systems and Software*, May 1989.
16. K. Narasimhan and K. Nilsen. Portable execution time analysis for RISC processors. In *Proc. of ACM PLDI Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, Jun 1994.
17. C.-Y. Park. *Predicting Deterministic Execution Times of Real-Time Programs*. PhD thesis, University of Washington, Aug 1992.
18. P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, Sep 1989.
19. J. Rawat. Static analysis of cache performance for real-time programming. Master's thesis, Iowa State University of Science and Technology, Nov 1993. TR93-19.
20. A. Shaw. Reasoning about time in higher-level language software. *IEEE Trans. on Software Engineering*, Jul 1989.
21. N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst-case execution times. *Journal of Real-Time Systems*, Oct 1993.