Instruction Duration Estimation by Partial Trace Evaluation

Author(s):
Corti, Matteo; Gross, Thomas

Publication date:
2004

Permanent link:
https://doi.org/10.3929/ethz-b-000158503

Instruction Duration Estimation by Partial Trace Evaluation

Matteo Corti
ETH Zurich
Departement Informatik
CH 8092 Zürich

Thomas Gross
ETH Zurich
Departement Informatik
CH 8092 Zürich

Abstract

Hard and soft real time systems require, for each process, the worst-case execution time (WCET), which is needed by the scheduler's admission tests and subsequently limits a task's execution time during operation. A worst-case execution time analysis is usually performed in two distinct steps: first the program is analyzed to extract semantic information and determine maximal bounds on the number of iterations for each basic block. In a second step the duration of the different program's instructions is computed with respect to the used hardware platform. Modern systems with preemption and modern architectures with non-constant instruction duration (due to pipelining, branch prediction and different level of caches) hinder a fast and precise computation of a program's WCET. We present a technique to approximate the instruction duration on modern processors using precise block bounds. Instead of simulating the CPU behavior on all the possible paths we apply the principle of locality limiting the effects of a given instruction to a restricted time allowing us to analyze large applications in linear time.

1. Introduction

A central aspect of the worst-case execution time (WCET) analysis is the computation of the various instructions duration for a given hardware platform. Modern architectures feature several factors making the precise computation of the instruction duration difficult: one or more pipelines, several cache levels, instruction reordering and branch prediction are an example. Systems with a dynamic set of processes and a preemptive scheduler add an additional layer of difficulty since we are not able to predict when a context switch will take place. Our approach to WCET analysis consists in a two-phase analysis: a precise and conservative semantic analyzer computes bounds for the program's basic blocks (see Section 2.1), while a second tool, the instruction length estimator, computes an approximation of the WCET based on the maximal number of block iterations specified by the first tool (see Section 3).

2. System overview

Our worst-case execution time approximator is composed by two tools: a machine independent semantic analyzer and a language independent instruction duration estimator. The static semantic analyzer is integrated as a module in an ahead-of-time Java bytecode to native compiler. We translate the bytecode to an internal representation in static single assignment form (SSA) [3], and perform several analyzes and optimizations. The static semantic analyzer is able to discover false paths, to bound the minimal and maximal number of iterations for several loops, and to discover the minimal set of targets for every method call. The analyzer then includes the information about the program's behavior in the output assembler file in form of (text) comments, which are processed by the instruction duration estimator.

2.1. Semantic analysis

The semantic analyzer performs several steps to derive the minimal and maximal number of iterations for each program's basic block. In a first phase we execute an abstract interpretation pass to derive ranges of possible values for local variables. Abstract interpretation is performed only on paths which do not cross loop boundaries ensuring a fast termination of the algorithm. This limitation does not allow us to bound loops, but we are able to reduce the set of possible values for many local variables at a loop's entry points: This information is then used by the static loop bounder to better understand the behavior of a loop's induction variables. The partial abstract interpretation pass is also useful to detect infeasible (or false) paths, which can be then excluded by the worst-case execution time estimator. After this first pass we bound the maximal number of loop iterations looking at the behavior of the loop's induction variables. Our technique is based on the Florida method [5] to find the bounds for the loop's header block. Using precise structural information we propagate the bounds to every basic block, taking care of the different execution counts for every block in the loop body. If a loop cannot be bounded, e.g., because it depends on the value of a method parameter, we inline the enclosing method and mark the caller for a further seman-

tic analysis pass. Specializing the unbounded methods we are able to analyze the loops in the caller context, keeping the loop bounding analysis and abstract interpretation pass simple and not parametrized. Method inlinig is performed recursively until either the loop is bounded or the user specified maximum inline depth is reached. Loops that cannot be bounded automatically (e.g., loops whose maximum number of iterations depends on the input set) are identified and the user is required to annotate the code manually specifying the bounds. For every dynamic call we determine, with a whole program analysis, the set of possible targets. This excludes the possibility to use reflection, which is not a desirable language feature in a predictable environment. Although recursion has been shown to be acceptable with some constraints in real-time programs [1], we limit our analysis to non-recursive programs by not allowing cycles in the call graph. At the end of the semantic analysis we embed the results in the class assembler representation as textual comments. These annotations indicate the control-flow graph structure, the minimal and maximal number of iterations per block, the false paths and the possible targets for each method call.

2.2. Instruction duration estimator

The instruction duration estimator is a second tool that takes as input the annotated assembler files from our compiler and analyzer and computes the WCET estimate on the given hardware platform. The class files are parsed and the control-flow graph is reconstructed from the embedded comments. Since we do not allow recursion we traverse the call graph in reverse topological order and estimate the WCET of the program's methods using partial trace evaluation (see Section 3).

3. Partial trace evaluation

The duration of an instruction on a modern processor is dependent on a lot of features such as data and instruction caches, data dependencies, and branch prediction. On a dynamic system where the number and kind of running processes is not known, and where the scheduler supports preemption, it is infeasible to precisely predict the state of the CPU at an arbitrary point in time. Several techniques try to precisely compute the instruction duration for pipelined systems [4, 14, 10, 8, 9, 16] while others extended the idea to the instruction cache behavior prediction [4, 8, 13]. A precise and conservative WCET estimation for highly dynamic systems with a preemptive scheduler normally requires analyses using path enumeration, which do not scale well for large applications. For this reason we implemented a first WCET approximator for the PowerPC processors where the length of each instruction was determined by static information from the CPU vendor combined with some run-time performance statistics [2]. Promising results convinced us that a mixed approach using safe bounds on the maximal number of basic block iterations combined with a conservative approximation of the instruction length could help to derive usable WCET estimation for large applications used in non-safety-critical soft real-time systems.

3.1. Locality

Our approach tries to compute a fair estimation of the instruction duration without a precise and complete knowledge of the set of running processes on systems which support task preemption. We assume that the effects of an instruction on the pipeline and caches will fade over time and that they will be no more relevant after a certain number of executed instructions. In other words the effects on the caches and pipelines of an instruction i on a given program trace are relevant only for a limited set of n instructions, where n is an arbitrary number defined experimentally. For every possible execution trace going through an instruction i we define a partial trace as the set of the last n instruction before i on this particular trace. Using the principle of locality a partial trace also defines the set of instructions that could influence the duration of i on the given execution trace. The duration of an instruction i on a given partial trace t ($duration(i,t)$) is computed simulating the instruction cache and pipelines behavior for the given set of n instructions and looking at the number of cycles needed to execute i (See 3.2). We define $T(i,n)$ to be the set of unique partial traces of length n that end in i (i.e. i is the last instruction of the partial trace). We define the WCET of an instruction i as: $WCET(i) = \max_{t \in T(i,n)} (duration(i,t))$. I.e. we compute the duration of i on every partial trace of length n which ends on i and we conservatively consider the worst case only. To limit the number of traces that have to be simulated, some optimizations are possible. Partial simulation results (e.g., the state of the pipeline) are cached and can be reused if the initial part of the two traces is the same. This is particularly effective if we force traces to begin at basic block boundaries. Furthermore the same trace simulation can be extended to the following instructions as long as no jumps are encountered.

3.2. Partial trace simulation

To simulate the instruction cache and pipelines behavior of a partial trace we implemented a basic simulator using a simplified Intel Pentium II processor [6] model. For every clock tick we evaluate the state of the different CPU units keeping track of the instructions we are interested in. The following paragraphs describe which processor elements are considered in the model and how our simulator handles them.

Pipeline The fetch and decode unit fetches one 32-byte cache line from the instruction cache and decodes it to the internal micro-ops. Every IA-32 instruction is decoded to one or more pre-programmed micro-ops which are then passed to the instruction pool. The unit can decode up to six

mirco-ops per clock tick. The instruction pool holds from 20 to 30 micro-ops, which are then dispatched to the correct execute unit in any order. The Pentium II processor has several execution units divided among five ports (pipelines). Several execution units can execute micro-ops on each pipeline, but only one micro-op per cycle can be fed to a port. To keep the simulation fast and simple the current implementation does not precisely model the five execution pipelines and we do not perform a data dependency analysis: we estimate the latency of every instruction with heuristic methods. The retirement unit has the task to reorder the micro-ops and can process up to three instructions per clock tick.

Memory operations Partial trace simulation does not allow to simulate the data cache behavior since at the beginning of the instruction sequence the memory and data cache status are unknown. For this reason we do not currently model data dependencies in memory and we assume that a write to a store buffer will not influence the instruction duration. Assuming that the system has enough memory to handle the process and that no swap space is present, our model uses a statistical approximation of the memory accesses, computing the cost of a read in the following way:

$$\begin{aligned} c_{read} &= p_{L1hit} \cdot c_{L1} + (1 - p_{L1hit}) \cdot (c' + c_{L2hit}) \\ c' &= p_{L2hit} \cdot c_{L2} + (1 - p_{L2hit}) \cdot (c_M + c_{L2_hit}). \end{aligned}$$

Where p_{Lnhit} is the probability to have a hit on the cache level n, c_{Ln} is the cost of a cache hit on level n and c_M is the cost of a memory read. The values for c_{L1},c_{L2} and c_M are determined statically by the Intel specifications [6] while the global hit probabilities for the whole program are determined statistically by measurements performed on a test run of the analyzed application.

Branch prediction To estimate the effects of branch prediction we use the bounds on the maximum number of basic block iterations computed with the static semantic analysis (see Section 2.1). We consider the probability of a branch from a block b to a block s as:

$$p_{branch(b,s)} = \frac{max(iter_{edge(b,s)})}{max(iter_b)}$$

where $iter_{edge(b,s)}$ is the number of times we will jump from b to s and $iter(b)$ is the number of times that block b will be executed. Comparing the expected CPU behavior described in the documentation with our predicted branch direction we can estimate how often a branch misprediction will occur.

3.3. Longest path

Once we have the worst-case execution time of each instruction we can easily compute the duration of a basic block b as the sum of the durations of all the instructions i it contains: $WCET(b) = \sum_{i \in b} WCET(i)$. Since, due to pipelining, an instruction could be spread over the execution of two basic blocks we define the duration of a basic block as the time from the beginning of its first instruction to the clock tick before the execution of the first instruction of the next block. To be able to compute the longest path from a method source to the method exit we have to transform the control flow graph representation removing cycles: For every loop we remove the back edges and replace them with an edge to all the possible loop's exit points. We then update the weight of each loop's block, multiplying it with the block's maximum number of iterations (the initial weight of a block is defined as its worst-case execution time). We obtain an acyclic representation of the control flow graph that retains information about block iterations in the form of nodes weights. Note that cycles are removed from the representation only, and that we do not perform any loop unrolling on the actual code. To find out the longest path from the entry to the exit block of the graph which is not a false path, we use a best-fit algorithm based on what presented in [15] which is able to perform the path enumeration in logarithmic time when the number of false paths is small. Method calls are handled by traversing the call graph in depth-first order and inserting the maximum duration of all the possible callees at the corresponding call site.

3.4. Complexity

The limited length of the partial traces limits the number of paths, and consequently the number of instructions, that we have to consider for simulation. For a program with N instructions and B basic blocks our analysis needs: $O(2^{n \cdot \frac{N}{B}} \cdot N)$ steps. Since n (the length of a partial trace) and $\frac{N}{B}$ (the average number of instructions per basic block) are constant, the total running time is in the asymptotic case linear.

4. Results

Testing the soundness of a WCET predictor is a tricky issue since, for complex examples, the real maximum execution time is difficult or even impossible to measure. To compare the estimated values with the measured time we must force the execution of the longest path, which is not normally known. The easiest way to validate such a tool is therefore the comparison of the results for small known synthetic applications where the real longest path is known or computable by hand. The first half of Table 1 shows the estimated WCET and the longest measured execution time for a set of small benchmarks. For these small tests we know, by hand, which is the longest path and we can easily force its execution. This means that we can directly compare our estimation to the longest measured run time, which corresponds to measured worst-case execution time of the benchmark. The results are encouraging and show the soundness of our technique: We are able to compute a conservative

estimation of the WCET with errors less then 25% in a fast and simple way. All the tests where performed on a 1-GHz Intel Pentium III-based PC. The longest observed running time were measured by using the CPU on-chip performance counters. The second half of Table 1 shows the results for some bigger applications. JavaLayer [7] is a pure Java library that decodes, converts and plays MP3 files (in our benchmark we decode some sample MP3s to raw audio data). SciMark [11] is a composite Java benchmark measuring the performance of numerical kernels occurring in scientific and engineering applications (FFT, SOR, sparse matrix multiply, Monte Carlo integration and dense LU matrix factorization) and _201_compress (we replaced the file input with a 4KB chunk of random bytes) is part of the SPEC JVM98 benchmarks [12]. In case of big applications we

Benchmark	Measured	Estimated
Division	$1.545 \cdot 10^9$	$1.400 \cdot 10^9$ (10.351%)
Jacobi	$1.075 \cdot 10^{10}$	$8.788 \cdot 10^9$ (22.351%)
MatrixInversion	$1.553 \cdot 10^9$	$1.419 \cdot 10^9$ (9.402%)
MatrixMult	$2.732 \cdot 10^9$	$2.667 \cdot 10^9$ (2.448%)
_201_compress	$9.45 \cdot 10^9$	$1.11 \cdot 10^{10}$ (117%)
javalayer	$2.67 \cdot 10^9$	$1.30 \cdot 10^{10}$ (487%)
scimark2	$2.47 \cdot 10^{10}$	$1.42 \cdot 10^{11}$ (579%)

Table 1. WCET estimations.

are not able to compare our estimation with the measured WCET since the input generating the longest execution time is not known and we cannot force it at run time. A manual inspection of the code is not feasible due to the size of the test applications (JavaLayer has 10'000 lines of code). We run the program with different input sets and we compare our estimation with the maximum measured process execution time. For this three applications we compute conservative estimations which are not too far from the measured WCET. To validate our locality principle we run several tests of the same application with increasing values of n (the length of a partial trace in instructions). After a certain value (around 50–100 instructions) the total estimated length does not vary any more. This validates our principle of locality showing that the effects of an instruction are limited to a small part of the analyzed program.

5. Concluding remarks

This paper describes an approach to compute a realistic estimation of the WCET for large applications on highly dynamic systems with a preemptive scheduler. The maximum number of iterations for each basic blocks is precisely computed by a semantic analyzer, while the length of the single instructions is approximated using the principle of locality, limiting the effects of an instruction on pipelines and instruction caches to a short program snippet (a partial trace). We are able to compute a fair estimation of the WCET in linear time allowing to handle big and complex applications. The tool has room for improvements especially in the instruction cache and pipeline model, but this prototype shows that is possible to get usable results for big analyzed programs in a fast and effective way.

6. Acknowledgments

This work was funded, in part, by the NCCR "Mobile Information and Communication Systems", a research program of the Swiss National Science Foundation, and by a gift from Intel's Microprocessor Research Laboratory.

References

[1] J. Blieberger and R. Lieger. Real-time recursive procedures. In *Proceedings of the 7th EUROMICRO Workshop on Real-Time Systems*, pages 229–235, Odense, Denmark, June 1995.

[2] M. Corti, R. Brega, and T. Gross. Approximation of worst-case execution time for preemptive multitasking systems. In *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Vancouver, Canada, June 2000.

[3] R. Cytron et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[4] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Trans. Computers*, 48(1):53–70, Jan. 1999.

[5] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proc. 4th Real-Time Technology and Applications Symp.*, pages 12–21, Denver, Colorado, June 1998.

[6] Intel Corporation. *Intel Architecture Optimization, Reference Manual*, 1999.

[7] JavaZOOM. Javalayer. http://www.javazoom.net/javalayer/javalayer.html.

[8] Y.-T. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond directed mapped instructions caches. In *Proc. 17th IEEE Real-Time Systems Symp.*, pages 254–263, Washington, D.C., Dec. 1996. IEEE.

[9] S.-S. Lim, J. Han, J. Kim, and S. Min. A worst case timing analysis technique for multiple-issue machines, Dec. 1998.

[10] T. Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers Univ. of Technology, Göteborg, Sweden, June 2000.

[11] B. Pozo, R. Miller. Scimark2. http://math.nist.gov/scimark2/.

[12] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1996.

[13] H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proc. 19th IEEE Real-Time Systems Symp.*, pages 144–153, Madrid, Spain, Dec. 1998. IEEE.

[14] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, May 2000.

[15] S. Yen, D. Du, and G. S. Efficient algorithms for extracting the k most critical paths in timing analysis. In A. Press, editor, *Proc of the 1989 26th ACM/IEEE Conf. on Design Automation*, pages 649–654, Las Vegas, NV, June 1989.

[16] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, Oct. 1993.