Techniques for Estimating Worst-Case Program Execution Times with Unexpected Hardware Architecture Changes or Unknowns

William Y. Jhun
Computer Science Honors Paper
Ashok K. Agrawala, faculty sponsor
Spring 2001

I. Introduction

In situations where the execution time of a piece of code is critical to the stability and performance of a system (e.g., hard real-time systems), the expected execution time and variance must be calculated prior to the configuration of the system or, at the very least, prior to the occurrence of the event that utilizes this code. In other words, for an event to be scheduled to occur at a predictable time (and assuming that this event cannot be triggered by a timer interrupt, which would be far too grainy and have long, varying latencies between occurrence of the interrupt and execution of the interrupt handler), previous events must be guaranteed to finish by a certain amount of time before the next event. Given that there may be some known amount of overhead before the next event can begin (say a call to the real-time scheduler), not only can the two events not overlap in time, but there must exist some minimal time between the completion of the previous event and the beginning of the next. Should the timing policies between any consecutive pair of events be breached, stability and accuracy of the real-time system will be compromised. To prevent these types of conflicts, estimations about the Worst-Case Execution Time (WCET) for every critically time-bound piece of code must be made. The aim of analyzing and determining the possible execution times for a piece of code is to come up with a "safe" WCET number that will never underestimate the actual execution time while keeping it as "tight" as possible by minimizing overestimation [EES00].

For software that is designed to run on a single hardware configuration (i.e., one that is sufficiently available for software profiling or one that can be completely simulated), finding a WCET could be done through statistical analysis or close analysis of microarchitecture implementation (taking various prediction failures, cache misses and other hazards into account) in addition to program data and control flow analysis. Such data collection and analysis suggests that knowledge of the target microarchitecture is available at compile-time, and sufficient modeling and simulators for the target platform are available to calculate reasonable timing characteristics. This paper will discuss not only the techniques that are used to analyze and profile execution times, but also the scenarios whereby limited knowledge of the hardware implementation is available. The purpose of this paper is to propose some techniques for better approximating WCET in these scenarios.

II. Trends in Modern Hardware Architecture

As computer systems become increasingly fast and affordable with time, their respective architectures become more complex due to hardware optimizations, dataflow/control predictors, and increasingly complex I/O and memory subsystems. Starting with designs in the 1950's-1960s, instruction pipelining introduced the notion that the execution time for a certain type of instruction may not be a constant due to varying latencies caused by instruction dependencies. Once pipelining became popular again in the late 1970's (the 80x86 processor had a primitive form of pipelining and many new RISC designs of the 1980s utilized extensive pipelining in initial implementations), additional mechanisms such as delay slots and branch predictors were universally implemented in workstation to server level processors [HP96, 112-116]. In an attempt to assure backward compatibility, newer Intel 80x86 architectures used a mechanism to translate x86 instructions to internal, fixed-length, pipeline-able pseudo-instructions. Modern commercial processors, such as the Intel Pentium II, use implementations of dynamic rescheduling and out-of-order (o-o-o) execution which even further complicate the task of predicting execution latencies and sequences. Additionally, many of the implementation techniques used in commercial processors are proprietary; details of their operation are often restricted information and/or unpublished.

ARM, Motorola 680x0, MIPS, and Intel 80x86 Instruction Set Architectures (ISAs) remain baseline architectures for embedded processors of today. Availability, affordability, and consistency are all properties that motivate the use of these ISAs for embedded applications. Motorola's Dragonball and Fireball processors and AMD's exceptionally small and low-power 80486 and Pentium class processors are examples of popular, inexpensive embedded processors modeled after commercial ISAs. The high availability and choice of compilers, programs (and their programmers), operating systems and existing hardware for these

Chip Family	Number Sold
ARM	151 million
Motorola 68k	94 million
MIPS	57 million
Hitachi SuperH	33 million
x86	29 million
PowerPC	10 million
Intel i960	7.5 million
SPARC	2.5 million
AMD 29K	2 million
Motorola M-Core	1.1 million

Table 1: 1999 32-bit Microcontroller Sales [Hal00]

architectures make systems built upon these architectures easier to develop, develop for, and implement, resulting in a strong correspondence between availability and time-to-market. With this relationship in mind, it is not unreasonable to expect future embedded processors to adapt to future mainstream processors. Though one might argue that embedded applications will not require enough speed increase of the hardware to prompt moving from existing embedded architectures, the history of computer software suggests that the functionality and resource requirements of these applications will grow to fill the potential resources that exist [TS97]. An example of this can be the fact that using an embedded 8-bit microprocessor to control a set-top cable box may have been a major advancement fifteen or twenty years ago, but today a similar box is expected to have internet functionality and be able to digitally record and playback video programs. Consider the theory of using Sun's Java Technology as a language to control appliances: as an exaggerated example, would a lowly 8-bit Zilog Z80 processor be expected to run a Java virtual machine? So long as there will be hard real-time systems and applications that grow in complexity and processing capacity, the architectures they run on will continue to evolve accordingly.

III. Techniques for Measuring WCET

In continuation of analyzing software execution times and behavior, consider a simple description for a piece of code: its Worst Case Execution Time. Approximating a WCET for a piece of code requires analysis of the data and control flow of the program in addition to the effect of hardware architecture design features on the execution of the compiled code.

Program Flow Analysis

Program flow analysis involves studying a program's dynamic behavior by permuting all possible paths of execution through a program (including analysis on function calls, loop iterations, conditionals, etc) [EES00]. This is done with the aid of manual annotations and valid, realistic input datasets for the program. Given that many of these programs are designed to be atomic, elemental units (which may often not depend on interrupt-driven I/O but instead depend on a simple messaging abstraction, as in the case of MARUTI [SSA95]), providing a simulated "user" environment for profiling execution can be more straightforward than for non-real time, full-blown applications. This process generates frequency data for basic blocks (i.e. direct line of code into which no other code branches into and no instruction branches out). Once execution time boundaries can be computed for these basic blocks, general execution time statistics can be generated by applying these time boundaries to the basic block frequencies found earlier. Such a technique is known as automatic flow analysis [EES00]. In some cases, known execution times can be manually annotated in the original language of the program. Sometimes, timing restrictions and temporal ordering can be specified in a language, leaving the analysis and profiling to the compilation stage. MCL, the Maruti Configuration Language (as a complement to the Maruti Programming Language), is such an example. In MCL, one may specify that the "interval" between the end of one action and the start of another be bounded by a maximum and/or minimum [SSA95]. The lower bound of this interval suggests that the tightness of a WCET estimation (perhaps complemented by a BCET estimation) be accurate. In combination with an MCL specification, an MPL program will be compiled with these constraints in mind, allowing the compiler to analyze execution times and determine if the specified restrictions can be met or not.

Low-Level Analysis and Simulation

Though flow analysis can describe the typical (as well as best and worst case) behavior of a program in terms of the ordering of basic blocks (control path), the interaction of the compiled program with the target hardware must also be modeled. Most methods for this evaluation fall under two categories: profiled or measured, requiring statistics of actual execution, or analytical, requiring close and accurate modeling of the underlying hardware. Techniques of the latter form include feeding code through a well-modeled pipeline (simulating all possible hazards and instruction scheduling methods), simulating execution with different cache models (varying size of cache, page replacement scheme, page mapping scheme, etc), memory latencies, etc. [Lim95]. Profiling and measurement are not guaranteed methods and can be costly to implement, thus making it a decreasingly attractive solution for WCET analysis [Lim95].

Initially, execution analysis considers the basic blocks of code. Here, measurements of straight lines of code give ballpark execution times that will most likely not show much variance (at this stage, analysis is only to the scope of the code; outside influences like context switches, interrupts, etc. are not taken into account). However, this analysis is not sufficient in itself and cannot be added incrementally when combined with analysis of flow between basic blocks, since the execution time for a basic block depends on the surrounding blocks [Lim95]. In other words, the state of the system upon entering a basic block must be taken into account. Thus, findings from flow analysis can then be used to re-calculate actual execution times of a single path. This would take into account pipeline flushing across basic blocks, compulsory and conflict cache behavior, etc [Lim95].

IV. Options for WCET Estimation With Limited Knowledge of System Parameters

As stated earlier, situations may arise whereby a hard real-time system is designed for a common, inexpensive general hardware platform, say, a Pentium-class PC compatible. Though extensive analysis and profiling of a specific system will allow for tighter execution time bounds, resulting in a generally higher performance system, Worst-Case Hardware Configuration (WCHC) scenarios may help increase the range of hardware configurations on which an already compiled execution-time sensitive program can run. This essentially increases the WCET for a program, thereby increasing the range of safety and decreasing the tightness, resulting in code that can run on more platforms while possibly sacrificing efficiency. Once the WCET has been adjusted such that it can reliably account for execution on a wide range (though a well-specified criteria) of hardware platforms, further attempts can be made to tighten the WCET boundary at runtime.

Expanding WCET to Account for Variation in Hardware Configurations

- **Cache Memory Variation**

In the methods described to this point, general assumptions have been made about machine behavior to allow for a clear understanding of how a program's execution can be modeled. One of these assumptions was that the analyzed programs would not be pre-empted and the resulting cache flushing would not occur. In reality, pre-emptive scheduling requires these conditions to happen.

In practice, due to the unreliable behavior that it introduces, real-time systems often either do not use cache memory or disable it upon booting up. When cache is used, however, cache interference can be classified in two ways: intra-task interference, involving only conflicts caused by memory blocks in the same task, and inter-task interference, involving memory block conflicts when interleaving one task with another [Lee96]. The first can be analyzed through techniques described above, whereas the latter requires analysis of the current scheduling. In general, the worst case "response time" R_i of task i due to cache conflicts can be measured in the following equation developed by Lee *et. al.* in [Lee96]:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + PC_i(R_i)$$

Where C_i represents the worst-case execution time of task *i* without cache preemption involved, *hp(i)* represents the set of tasks of higher priority than i, T_j is the period of task *j* and $PC_i(R_i)$ is the total cache-conflict related preemption delay for task i [Lee96]. In this recurrence, R_i is initially set to C_i. In the case of many different possible cache memories, R_i should be selected to reflect the least favorable cache system (i.e., whichever one produces the most number of cache misses (longest calculated response time). Upon evaluating this response time, it would then be up to the system to determine the feasibility and improvement in execution time to perform cache pre-loading (provided a consistent footprint could be generated in order to pre-load this footprint before preemption).

- **Processor Speed Variation and Processor Type Variation**

Worst-case instruction execution timings provided by the chip vendor may be far too pessimistic to provide efficient bounds. However, in the case where the vendor of the chip is not even known, it may be wise to collect the worst case timing data for all possible vendors. In the situation where hardware configuration cannot be known ahead of time, at least the major components must be from a strict list of acceptable devices. In this case, a processor may need to be one of three major chip vendors for this ISA. Otherwise, how could any guarantees be made about the execution time of a task?

- **Memory Type Variation**

In this case, rather simply, any time a "cache miss" or cache preload takes effect (the statistics for these should have already been computed), the cost of loading a page from memory must be calculated. Assuming no DMA or bus hazards, memory access could be profiled at startup (provided uniform access times for the architecture) or specified by the chipset or BIOS. Thus, the cost of a cache miss would be bloated by relatively slower memory than expected.

Techniques for Tightening WCET

- **Distribute the system and real-time applications in an intermediate, un-optimized state**

  By offering the software in its generic, WCHC form, special configuration tools can be used to perform a "system analysis" which will analyze the system features, profile the execution of critical code fragments and accordingly adjust WCET estimations and re-calculate scheduling calendars and performance guarantees.

- **Identifying specific hardware features at runtime/alternate code fragments**

  After attempting to identify key hardware components on system startup, the system may recognize certain features that will allow optimization on-the-fly. For example, if the system was compiled for use with a specific brand and model processor, and it is well known that the processor is functionally identical regardless of clock speed, re-calculation of timing constraints may be possible. This would require that the execution time estimation be a function of this parameter (clock speed) and that it can be recomputed on-the-fly. Alternately, upon recognizing certain hardware features, alternate code fragments could be linked in which were compiled with these features in mind.

- **Global and local low-level analysis at runtime**

  Consider a case where a program is not available at the time of system installation/configuration or system startup. Perhaps the program is a standalone application or downloaded code for use with RPC or Java RMI, for example. Ideally, this code should contain information and results about an initial software flow analysis, since this part is relatively platform independent (assume for the sake of argument that it is a computationally-intensive routine that has no inherent I/O dependencies).Upon acquiring this code from a source which may not have any knowledge of the target system, the system could apply a very large and very safe WCET to the code, thereby guaranteeing total safety but lacking useful or optimal guarantees about execution time. While the code is running, the system can profile its execution time and other characteristics.

  This technique unfortunately caters to iterative operations where the bulk of the program is being run continuously (generating sufficient profiling statistics). For programs that are scheduled to run infrequently but are designed to be bound by a short execution time, this technique is useless. An alternative would be to have a profiling/analytical engine that runs whenever there is free CPU time. This engine would attempt to gradually lower the WCET long before the program is scheduled to run, thereby allowing later events to be scheduled in time slots that the original WCET would have forbidden (by lengthening the expected execution time of the event containing the program in question).

V. Conclusion and Possible Future Work

A system with loosely-bounded execution timings may limit the guaranteed performance of a system, but would require only a general specification of a hardware platform, allowing for a system that can properly run on a wide range of inexpensive, off-the-shelf hardware. Upon installing and configuring the system on a particular hardware configuration, system analysis could then open these timing constraints, allowing a higher period of critical events (should that be beneficial) or more events to be dynamically scheduled into the system.

Pursuing this work would require a highly portable, hard real-time system that can allow complete manual annotations for tasks, including a specified WCET (i.e., bypassing the execution time analysis stage of the system). This would allow experimental, loose values to be used and tested across many platforms. Additionally, special configuration software to "fine-tune" these values would need to be written. This software would need to detect the types of all critical devices in the system, perform on-the-fly low-level analysis of task execution times, including response times due to cache preemption with other tasks. Though this may be a lengthy procedure, it may only need to be executed once, or as the hardware platform changes. With these two main elements, a system with safety and performance nearing, if not equal to, a system pre-tuned for a specific platform would be possible.

Acknowledgements

Special thanks to my mentor, Dr. Ashok K. Agrawala, for helping me tune the focus of this paper, to Bao Trinh, for suggesting ideas and reflecting on experience with cache pre-loading, and to Dr. Manoj Franklin, for reviewing my hypothesis and discussing the practicality of such analysis.

References

[EES00] Jakob Engblom, Andreas Ermedahl, Mikael Sjődin, Jan Gustafsson, Hans Hansson. Worst-Case Execution-Time Analysis for Embedded Real-Time Systems. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS'00),* November 2000.

[Hal00] T. R. Halfhill. Embedded Market Breaks New Ground. *Microprocessor Report, January 17, 2000.*

[HP96] David A. Patterson, John L. Hennessy. Computer Architecture: A Quantitative Approach. San Francisco: Morgan Kaufmann Publishers, Inc. 1996.

[Lee96] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, Chong Sang Kim. Analysis of Cache-related

Preemption Delay in Fixed-priority Preemptive Scheduling. Proceedings of the 17th IEEE Real-Time Systems Symposium, 1996.

[Lim95] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, Chong Sang Kim. An Accurate Worst Case Timing Analysis for RISC Processors. Dept. of Computer Engineering, College of Engineering, Seoul National University, 1995.

[SSA95] Manas Saksena, James da Silva, Ashok Agrawala. Design and Implementation of Maruti-II. University of Maryland at College Park, Department of Computer Science. 1995.

[TS97] Clyde Edwin Taylor, Ronald Neal Schroeder. Today's RISC Microprocessors and Real-time Concerns. Industrial Computing Society, 1997.