On the Difficulty of Building a Precise Timing Model for Real-Time Programming *

Albrecht Kadlec, Raimund Kirner

Institut für Technische Informatik,
Technische Universität Wien, Austria
{albrecht,raimund}@vmars.tuwien.ac.at

Abstract. For real-time computing it is important to know the worst-case execution time (WCET) of all time-critical software operations in order to ensure timeliness of the system. The calculation of a precise upper bound of the WCET relies on the availability of an adequate timing model of the target hardware.
Within this article we explore the different mechanisms of modern processors that lead to complex timing models. We explore the different types of memory elements within a processor that resemble the state of the processor. Further, we compare the compile-time knowledge and run-time knowledge and discuss the consequences of offline (compiler) and online (hardware) code optimization. The main consequence of this hardware exploration is that real-time computing needs co-design of compilation, timing analysis, and processor optimizations to improve temporal predictability of the system.

1 Introduction

To prove that a real-time computer system is able to meet its deadlines, the worst-case execution time (WCET) [1] of each time-critical task has to be known. In WCET analysis, a timing model of the real-time program has to be constructed to calculate the longest execution path [2]. This timing model depends on the concrete timing of the target platform hardware.

In case of simple processors where the execution time of an instruction is constant, one simply has to determine the execution time of each instruction. The WCET of a program can be calculated by applying a set of hierarchical timing calculation rules, called *timing schema* [3]. However, the processors nowadays used in embedded computing are often much more complex. Hardware features like caches, pipelines, or branch predictors are also used in embedded processors to achieve high peak performance.

Research in WCET analysis has shown that it is quite feasible to separately model moderately complex hardware features, e.g., instruction caches [4]. However, even instruction caches can become almost impossible to analyze if they use an unpredictable replacement strategy [5, 6]. Things are becoming worse, if the timing behavior of the different hardware features is not composable, i.e., the mechanisms influence the timing behavior of each other. For example, *long timing effects* may occur, where the timing variation of an instruction can have an effect on instructions executed much later [7].

* This work has been partially supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Compiler-Support for Timing Analysis" (COSTA) under contract P18925-N13.

Still worse, the interaction of different hardware components can also result in counter-intuitive behavior, like *timing anomalies* [8–10]. It has already been pointed out in [11] that there is a misconception thinking that static analysis may provide accurate results even in case of complex hardware components.

Within this paper we aim to provide guidelines about the temporal analyzability of different hardware mechanisms of modern processors. In Section 2 we give a general discussion of why modern processors are increasingly difficult to analyze. In Section 3 we classify the internal states of a processor with regard to analyzability. A comparison of program optimizations done in software (compiler) and in hardware is given in Section 4.

2 Sources of Complexity

By discussing the sources of complexity we want to obtain hints, how fast the complexity of timing analysis actually grows. This information is valuable for the planning of further timing analysis development.

In the following discussions we will use the acronyms $BCET$, $ACET$, $WCET$ to denote the $\{Best \| Average \| Worst\}$ $Case\ Execution\ Time$. For the $WCET$ we will further differentiate between the actual runtime $WCET$, the theoretically analyzable $WCET_A$ and the $WCET_C$ – the actual $WCET$-bound that can be computed by an analysis tool with a reasonable amount of analysis time. For these measures, the following inequations hold: $BCET \leq ACET \leq WCET$ $WCET \leq WCET_A \leq WCET_C$

2.1 Compile (= Analysis) Time versus Run Time Knowledge

While the offline compile time analysis does have a much wider scope of analysis than the optimizations done in hardware, the type of knowledge is totally different than at runtime: it is *offline knowledge*: everything that depends on input data is unknown and sets of alternatives must still be considered – like for control flow. The hardware on the other hand has no or very limited look-ahead, but can peruse already computed data values – *online knowledge*. An especially important example is alias analysis: even the most complex global alias analysis cannot disambiguate all memory accesses. The hardware just needs a comparator for the already computed addresses. The scope of the offline optimization is however much greater (e.g.: code motion), justifying the greater effort. The key observation is that "potential aliasing" is the problem for analysis and is difficult to handle in software due to incomplete compile time knowledge, while true aliasing is actually very rare but easily detected and handled in hardware using already computed values.

To characterize it in few words: offline knowledge is rather large and inherently imprecise, whereas online knowledge is rather small, but more precise. Thus optimizations that are based on *run-time* knowledge are difficult to predict at *compile-time*, yielding high complexity for timing analysis.

2.2 Functional Orthogonality does not imply Timing Orthogonality

Hardware is designed as a set of functionally independent blocks that can be designed independently – with a limited amount of common planning. For the proof of functional correctness, these functional blocks can be considered independently, as each handles a distinct set of functionality. For timing performance, the average case is

most important. Thus the hardware design process concentrates on the average case often impairing the worst case.

As soon as timing variations are introduced, all mutually influencing functional blocks must be considered together for the analysis of guaranteed timing behavior, as a variation in timing may cause following timing variations. Given such interactions, the whole system must be analyzed as a whole - leading to a significant rise in complexity.

3 Classification of Processor State

The combination of the observations of Section 2.1 and Section 2.2 adds another layer of complexity, as the combined analysis is necessary, but without precise detailed knowledge being available. Still all timing behavior is closely related to processor state: a stateless processor is trivially free of timing variations (except maybe following physical variations). Thus it is beneficial to classify the different parts of processor state to assess the type and amount of its contribution to the complexity of the timing analysis. This may help to decide trade-offs in hardware design: a specific feature should be omitted from a timing-oriented design, if its contribution to analysis complexity is not justified by its contribution to performance.

In the following we factor out specific properties that can be used to judge the effect of a given instance of processor state:

Explicit Architectural State – The most prominent example are the architectural register files. As this state is architecturally visible, it is handled explicitly by code generation and by timing analysis and thus no unforeseen timing variations can occur. The same is true for pipeline registers in an explicitly pipelined architecture (e.g.: Digital Signal Processor – DSP): As pipeline hazards are exposed on the architectural level, they are explicitly known at compile time and analysis time and handling can be moved to software alone.

Implicit, Hidden, Implementation-specific State – This kind of state is excluded from the architectural description and thus to a great deal hidden from the tool chain. However, the classification as "implementation detail" is incorrect with respect to the timing behavior of the implementation. Examples of hardware features with implicit state are: pipelines, caches, hidden registers (e.g.: for renaming)

Although bad in nature, there are some additional properties of this hidden state, that can make a big difference for the complexity of timing analysis:

Stability: This property tells, whether knowledge of the state stabilizes and converges again after unknown events. Then, after a limited sequence of known events, the state is known again.

Resilient/stable – local influence: Even if a state is reached that is not analyzable with offline knowledge, after a limited number of events, the state is again known and analyzable. Examples are true Least-Recently-Used (LRU) caches and local-only branch prediction.

Fragile/unstable – global influence: If the precision of knowledge does not increase again after an unknown event but rather degrades even further although the following events are known, we call it unstable. The reason of this behavior is due to algorithms, that have a shared state component that retains history data, reusing data from previous events, thus linking otherwise independent events with each other in the context of timing analysis. The general pattern can be described as the mixing of otherwise

independent events for efficiency-of-hardware-implementation reasons. Examples are the replacement history bits in pseudo LRU and Pseudo Round Robin (PRR) caches, the global history shift register of branch prediction algorithms.

Distribution / Connectivity: The number of clients, that use a feature clearly has an influence of the distribution of timing variations:

Shared / central – global influence: A feature or device that has more than one client which are served from the same common state, clearly connects its clients in terms of timing behavior: events from one client can now influence the other ones. This leads to a fast spreading of timing variations into logically disconnected client units. Example are shared 2^{nd} and 3^{rd} level caches that serve instruction-, data- and translation-lookaside-buffer accesses, so that a complex common analysis is required, which then in turn suffers from phase ordering problems between code and data accesses.

Unshared / decentral – local influence: e.g.: Harvard-architecture caches only impose a feedback on shared pipeline state via instruction *delays*. Code and data accesses do not directly influence each other.

Decision Density & Granularity: A fine granularity means a potentially high density in time of – for the timing analysis with its limited knowledge – adversary events. This means a fast buildup of (potential) effects, if the events are independent: the interval $[BCET, WCET]$ is drifting farther apart with each event. The sheer number of events leads to a very large search space, if detailed analysis is required for preciseness reasons, raising the complexity of the analysis.

4 Influence of Optimizations in Hardware and Software

Simple optimizations improve on all of the timing measures more or less equally. Examples are simple expression simplification but also induction variable elimination or non-speculative partial redundancy elimination: the $BCET$ is most likely improved along with $ACET$, $WCET$ and $WCET_C$. Advanced optimizations use heuristics and/or profile feedback to guess the typical average case to improve the $ACET$, ignoring the effect on all other timing properties. Focussing on the worst-case execution time properties, the negative effect of optimizations can be three-fold:

1. Most often there is a direct increase in the (rare) $WCET$.
   That is the trade-off that is accepted, when focusing on the $ACET$ alone.
2. Then there can be an *indirect increase of the* $WCET_A$:
   Here, timing analysis simply does not have enough knowledge at code generation time to prove beneficial cases that occur at run time and are exploited by hardware, leading to pessimistic overestimations that degrade $WCET_A$.
3. Last, there can be an *indirect increase of the* $WCET_A$ through *second-order effects*:
   An increase of complexity can create new phase-ordering problems that further compromise the $WCET_A$.

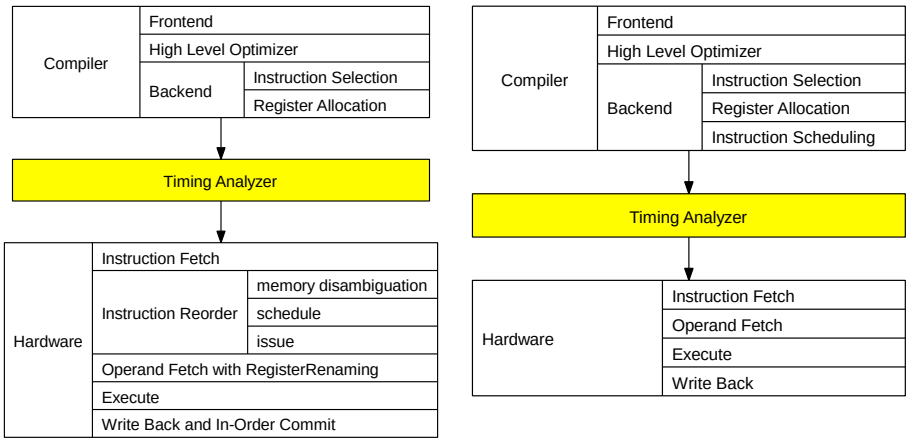
$WCET_C$ may degrade faster than $WCET_A$ when the specific analysis method used by the employed tool is especially sensitive to the problem changes.

To control the effect of optimizations, a compiler framework should actually be designed with these influences in mind. That means classification of each optimization so that $WCET$-increasing optimizations can be inhibited for $WCET$-aware compilation. This avoids increasing $WCET$ or $WCET_A$. However $WCET_C$ can still suffer, as the compiler is not aware of which analysis tool will be used.

Thus, as a second goal for a *WCET*-aware tool chain, *WCET*-analysis should be integrated into code generation, so that even such tool-specific degradations can be avoided – either by not doing the offending optimizations or by improving the specific timing analysis in the context of this optimization.

As a synergistic effect, many potential *WCET*-increasing optimizations can be allowed again in case they do not lie on the critical path and do not lead to a change in the critical path. In addition, execution-time-balancing optimizations are enabled as a new class of optimizations, i.e., given good estimates, it is possible to slow down an uncritical fast path to improve the critical worst-case path.

4.1 The Inappropriate Hardware / Software Interface



(a) HW optimizations after analysis

(b) Optimizations done in software before timing analysis

Fig. 1. Phase Ordering of Timing Analysis and Optimizations

Figure 1(a) explains the fundamental phase ordering problem introduced by hardware optimizations: Timing analysis happens after SW-optimizations, but before HW-optimizations. The former can be accounted-for, the latter must be forecasted with incomplete (analyze time = compile time!) knowledge, inevitably resulting in preciseness. Thus timing analysis preciseness is actually a function of hardware complexity, demanding simplified hardware as depicted in Figure 1(b), if the preciseness is to be improved.

When implicit implementation state is changed to explicit architectural state, the main loss is the loss of binary compatibility - not an issue in the embedded real time domain. The loss in *usable* performance is probably also not very high: i.e. the performance guaranteed after timing analysis may actually increase, as the elimination of pessimism in timing analysis may easily outweight the small benefit of online optimizations over their offline counterparts.

4.2 The Mismatch of Hardware Reality and Analysis Reality

The current situation for timing analysis capabilities versus complexity of current hardware is highlighted in Table 1. The table shows, that there is a wide gap between properties of current hardware [12] and the current capabilities of timing analysis. While

the most complex CPUs are typically not used in embedded real time systems, most CPUs that are actually used, have one or more problematic features, like a Pseudo Least-Recently-Used (PLRU) cache replacement strategy [6].

	timing analysis capability	*current hardware*
Caches		
levels	1 level	up to 3 levels
separation	separated (Harvard)	2^{nd} & 3^{rd} level combined
associativity	2-way	up to 16-way
replacement	LRU [5]	PLRU, PRR
Branch Prediction		
history	local 2-bit saturating	mixed local / global
locality	local only	mixed local / global / tournament

Table 1. Analysis Capabilities and Current Hardware

According to Moore's law, transistor count doubles every 18 months due to hardware improvements, improving performance in turn. According to Proebsting's law, performance doubling due to compiler technology happens only every 18 years. While both "laws" are heavily disputed and criticized for their extrapolation, they nevertheless express the industrial reality of the past thirty years.

While compiler technology employs a lot of phases that separately handle distinct aspects in order to reduce overall tool complexity and compile time, timing analysis is more and more forced to integrate such phases to keep its precision high. Furthermore, timing analysis may be forced to be merged into the code generation tool chain to improve overall WCET results. Thus it is obvious that the complexity of timing analysis is rising at least as fast as the complexity of compilers. Thus it is highly unlikely, that analysis precision can catch up with the imprecision caused by already present hardware features and even less so with future features.

The obvious alternative is to employ co-design of code generation, timing analysis and hardware design, i.e., add only hardware features that do not unduly raise the complexity of timing analysis. This way a better balanced system can be designed with better overall *WCET* properties of the software / hardware stack consisting of code generation tools, timing analysis tools and hardware.

5 Summary and Conclusion

Within this article we explored the different mechanisms of hardware features in order to classify the internal state of a processor. The results are aimed to provide insights into the sources of processors' complex timing behavior. They may help hardware designers in designing more predictable embedded processors.

Analyzing the internal state of the processor, it is the implicit, hidden portion of the processor state that causes the main challenges of constructing a precise and tractable timing model of the processor. The explicit state fraction of the processor state is easier to handle, because first, it is adequately documented and second, it can typically be calculated by analyzing only local instruction sequences of the program code instead of having to analyze the whole program execution. Another challenge is the dynamic code optimization performed by a processor. Timing analysis has to keep track of the execution history in order to reason about the processor state. However, since abstractions have to be used in practice to make the analysis tractable, it is not possible to keep a detailed execution history.

Given the complexity that stems from imprecise offline knowledge, interaction of otherwise disconnected features in the timing domain, fast propagation and distribution through highly interactive/interconnected features, large search spaces stemming from high decision density, *timing analysis complexity* is growing much faster than *hardware design complexity*. With this development, there will be no light on the horizon for timing analysis to keep up in modeling the temporal behavior of the processors. On one side it is relatively easy to develop complex hardware due to the functional modularity of the hardware building blocks. But on the other side, WCET analysis becomes increasingly complex due to the growing challenge of constructing a feasible, precise timing model of the processor. As a possible way out of this dilemma, co-design of compilation, timing analysis, and processor optimizations has been proposed.

References

1. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckman, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenstrom, P.: The worst-case execution time problem - overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) ((Accepted January 2007))
2. Kirner, R., Puschner, P.: Classification of WCET analysis techniques. In: Proc. 8th IEEE International Symposium on Object-oriented Real-time distributed Computing, Seattle, WA (2005) 190–199
3. Puschner, P., Koza, C.: Calculating the maximum execution time of real-time programs. The Journal of Real-Time Systems **1** (1989) 159–176
4. Mueller, F.: Timing analysis for instruction caches. Real-Time Systems Journal **18** (2000) 209–239
5. Heckmann, R., Langenbach, M., Thesing, S., Wilhelm, R.: The influence of processor architecture on the design and results of wcet tools. Proceedings of the IEEE **91** (2003) 1038–1054
6. Berg, C.: PLRU cache domino effects. In: Proc. 6th International Workshop on Worst-Case Execution Time Analysis, Dresden, Germany (2006)
7. Engblom, J., Jonsson, B.: Processor pipelines and their properties for static WCET analysis. In: Proc. 2nd Embedded Software Conference, Grenoble, France (2002) LNCS 2491, Springer Verlag.
8. Lundqvist, T., Stenström, P.: Timing analysis in dynamically scheduled microprocessors. In: Proc. 20th IEEE Real-Time Systems Symposium (RTSS). (1999) 12–21
9. Wenzel, I., Kirner, R., Puschner, P., Rieder, B.: Principles of timing anomalies in superscalar processors. In: Proc. 5th International Conference of Quality Software, Melbourne, Australia (2005)
10. Reineke, J., Wachter, B., Tesing, S., Wilhelm, R., Polian, I., Eisinger, J., Becker, B.: A definition and classification of timing anomalies. In: Proc. 6th International Workshop on Worst-Case Execution Time Analysis, Dresden, Germany (2006)
11. Kirner, R., Puschner, P.: Discussion of misconceptions about WCET analysis. In: Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, Porto, Portugal (2003) 43–46
12. Hennessy, J.L., Patterson, D.A.: Computer Architecture - A Quantitative Approach. 4th edn. Morgan Kaufmann (2007)