DOI:10.1145/3375545

The story of the development of a sound, static method for worst-case execution-time analysis.

BY REINHARD WILHELM

Real Time Spent on Real Time

THE GENERAL SETTING for worst-case execution time (WCET) analysis is that a set of hard real-time tasks is to be executed on a given hardware platform. Hard real-time tasks have associated deadlines within which they must finish their execution. The deadlines may be given by periods. *Timing verification* must verify these timing constraints are satisfied. Traditionally, timing verification is split into a *WCET analysis*, which determines upper bounds on the execution times of all tasks, and a *schedulability analysis*, which takes these upper bounds and attempts to verify the given set of tasks when executed on the given platform will all respect their deadlines.

The problem to determine upper (and potentially also) lower bounds on execution times underwent a transition in the 1990s:

▸ In the old days, textbooks about the realization of real-time systems would strongly argue against the use of execution platforms with caches, pipelines, and such. For previously used architectures with instructions that had constant execution times, WCET analysis methods using *timing schemata*[23] were the method of choice. Timing schemata describe how (bounds on) the execution times of a programming-language construct were composed from the (bounds on) the execution times of its components. These methods would thus do structural induction over the structure of a program and determine bounds for ever bigger parts of the program. Worse yet, industry's "best practice" was, and unfortunately partly still is, to do some end-to-end measurements, ignore some unwelcome outliers, if optimism prevailed, or add some safety margin, if more problem-awareness dominated.

▸ The introduction of performance-enhancing architectural components and features such as caches, pipelines, and speculation made methods based on timing schemata obsolete. Execution times did not compose any longer because instruction execution times were now dependent on the execution state in which they were executed. In the composition *A;B*, the execution time of statement *B* depended on the execution state produced by statement *A*. The variability of execution times

» key insights

- **WCET searches a huge state space for a longest path. Adequate abstraction of the execution platform is key to cope with the complexity of the analysis, and Abstract Interpretation provides the theoretical foundation for a sound and efficient WCET analysis.**
- **The Timing Predictability of an architecture determines the efficiency of WCET analysis and the precision of its results.**
- **Some performance-enhancing features ruin timing predictability and at the same time open the door to hardware security-attacks like Spectre and Meltdown.**

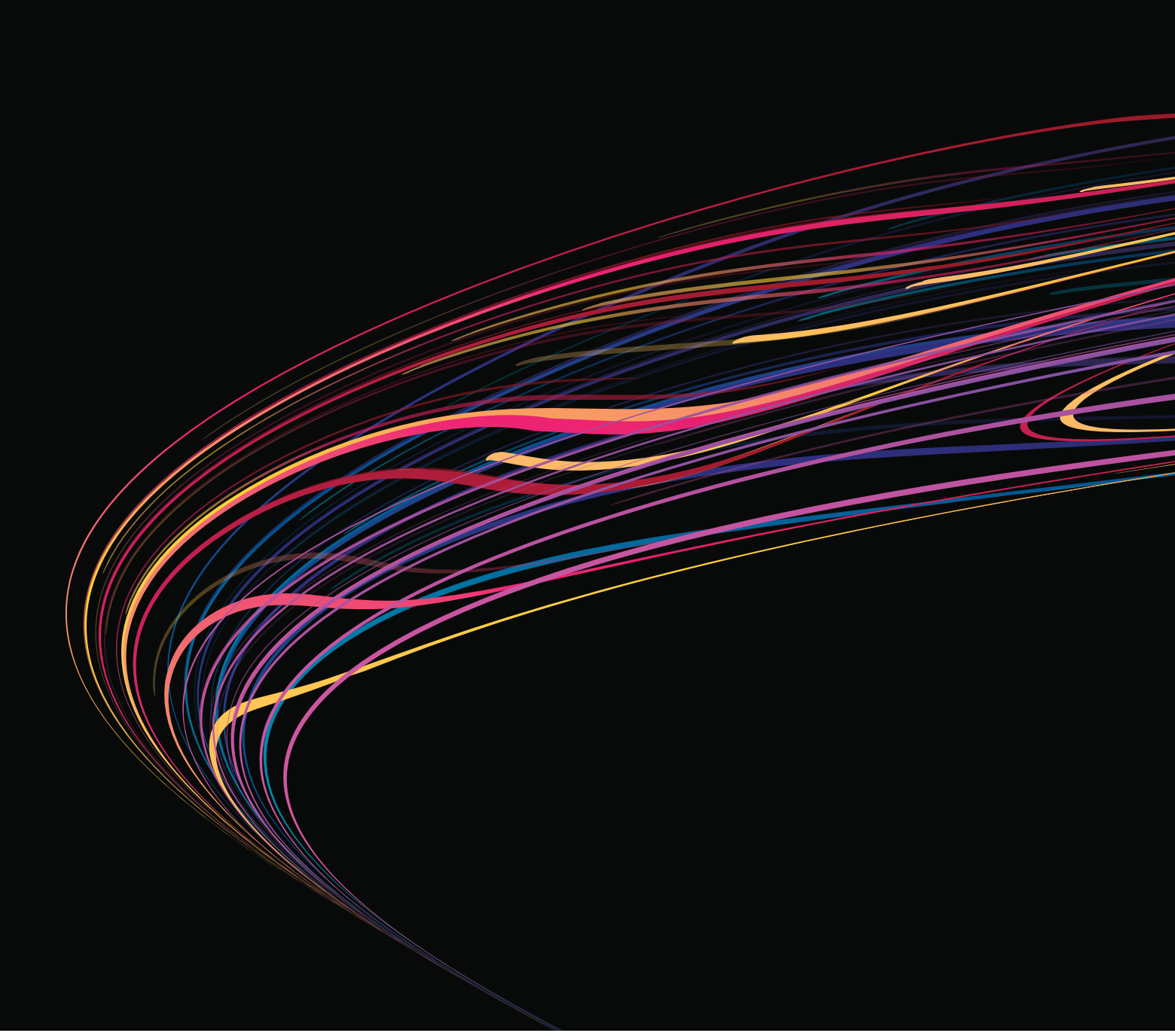

IMAGE BY MARYNOVA

grew with several architectural parameters, for example, the cache-miss penalty and the costs for pipeline stalls and for control-flow mispredictions.

The introduction of multicore execution platforms into the embedded real-time domain made the problem still more difficult. These platforms typically have shared resources, and the interference on these shared resources complicates the determination of upper execution-time bounds for tasks executed on such a platform. A few words about terminology. From the beginning we aimed at sound WCET-analysis methods. The results of a sound WCET analysis are *conservative*, that is, they will never be exceeded by an execution. We consider being conservative as a Boolean property. Often conservative is used as a metric, being more conservative meaning being less accurate. For an unsound method, however, it does not make sense to speak about being more or less conservative. It is not even clear whether being "more conservative" means moving toward the real WCET from below or moving further away from the real WCET by increasing overestimation. The second, quite important property, referred to when mentioning conservatism, is *accuracy* of the results of a WCET analysis.

WCET analysis can be seen as the search for a longest path in the state space spanned by the program under analysis and by the architectural platform. The analysis is based on the assumption that the analyzed programs terminate, that is, all recursions and iterations are bounded. We are not confronted with the undecidability of the halting problem. In trying to determine bounds on recursion or iteration in a program, our tool might discover that it cannot determine all of the bounds and will ask the user for annotations. All WCET bounds are then valid with respect to the given annotations.

This state space is thus finite, but too large to be exhaustively explored. Therefore, (safe) overapproximation is used in several places. In particular, an abstraction of the execution platform is employed by the WCET analysis. We will in the following cover static

analyses of the behavior of several architectural components.

Cache Analysis

Our engagement in timing analysis started with the dissertation work of Christian Ferdinand at around 1995. I had proposed several thesis topics, which he all rejected with the argument, "This is of no interest to anybody," meaning irrelevant for industrial practice. When I proposed to develop an analysis of the cache behavior he answered, "This may actually be of interest to somebody." He was able to convince himself (and me) very quickly that an idea we had would work. He used the program-analysis generator PAG, conceived by Martin Alt and realized by Florian Martin in his Ph.D. thesis,[19] to implement a prototype cache analysis for caches with a *least-recently used* (LRU) replacement strategy. This was, and still is, WCET researcher's dearest replacement policy. Our first submitted article on cache analysis[1] confirmed Ferdinand's appreciation for the subject. It received enthusiastic reviews as for the relevance of the problem we had solved and for the elegance of the solution.

Unlike existing methods, Ferdinand designed two different abstract domains for cache analysis, a must and a *may* domain.[1,6,8] An abstract *must* cache at a program point indicates which memory blocks will be in all concrete caches whenever execution reaches this program point. There is an underlying assumption that program execution is not disturbed by interrupts or preemption. Otherwise, the impact of interrupts or preemptions must be taken into account by analyzing the maximal cache impact through a CRPD analysis.[2]

An abstract must cache state computed at a program point represents an over-approximation of the set of concrete cache states that may reach this program point. All the concrete cache states in this overapproximation have as common contents the memory blocks that are sure to be in the concrete cache when execution reaches this program point. In LRU caches, memory blocks logically have an *age*. The age is between 0 and the *associativity*—1 of the cache (set) and corresponds to the relative position in the access sequence of cached memory blocks, extended by some way to indicate *not cashed*. The associativity of a fully associative cache is equal to its capacity. For a set associative cache it is the size of a cache set, that is, the number of memory blocks fitting into each set. It was (for us) easy to see our analysis should perform a kind of intersection and associate with the elements in the resulting set their maximal age from the incoming *must* caches, whenever control flow merges. Abstract *must* caches can be used to predict cache hits.

An abstract *may* cache at a program point indicates which memory blocks may be in a concrete cache whenever execution reaches this program point. In analogy, our analysis uses union at control-flow merge points and associates the minimal incoming age with the elements in the union. Taking the complement of a *may* cache gives the information which memory blocks will be in no concrete cache arriving at this program point. It thus allows to predict cache misses. In Lv,[18] we called such analyses *classifying analyses* as their results allow to classify some memory accesses as either definite hits or definite misses.

In contrast, *persistence analyses* are called *bounding analyses*. They aim at bounding the number of cache reloads of memory blocks. Intuitively, a memory block is called persistent if it suffers at most one cache miss during program execution. We defined such a persistence analysis, which was too beautiful to be correct. The tempting idea was to add another cache line with age *associativity* in which all memory blocks were collected that had been replaced in the cache at least once and to compute the maximal position (relative age) for all memory blocks that may be in the cache. The analysis would thus use union and maximal age at control-flow merge points. Our mistake was that we ignored the capacity constraints of the caches. Our analysis could collect more memory blocks in an abstract cache than would fit into the concrete cache and thereby underestimate the age. Luckily, this cache-persistence analysis was never implemented in AbsInt's tools. The error was later corrected by Cullmann and Huynh.[4,15]

Let me jump out of the story to some later developments: Jan Reineke has clarified the semantic foundations of persistence analysis.[21] All types of persistence are identified by the occurrence of certain patterns in memory-access traces, while categorizing cache analyses, that is, *must* and *may* analyses only abstract from the last state in a trace. Reineke also identified a whole zoo of different persistence analyses. End of excursion.

Understanding Our Approach

We developed the cache analysis with the goal of classifying memory accesses as either definite cache hits or definite cache misses. The difference to competing approaches was that we could describe our cache analyses as abstract interpretations.[3] This meant we defined the following:

- *domains of abstract cache states* with a *partial order* representing which domain elements contained better information than other elements,
- corresponding to this partial order, a *join function*, used to combine incoming abstract domain elements at control-flow merge points, for example, some kind of intersection for the must-cache analysis, and,
- *abstract cache effects* for each memory access, describing the update of abstract cache states corresponding to this memory access.

This stood in stark contrast to the state of the art in cache analysis, which would typically give a page of pseudo-C code and claim that this code would implement a sound cache analysis, of course without any correctness arguments!

Now, that we had solved one subproblem of WCET analysis, it was time to reflect more deeply what the essence of our method was, and we identified the following central idea behind our WCET-analysis method:

- Consider any architectural effect that lets an instruction execute longer than its fastest execution time as a *timing accident*. Typically such timing accidents are cache misses, pipeline stalls, bus-access conflicts, and branch mis-predictions. Each such timing accident had to be paid for, in terms of execution-time cycles, by an associated timing penalty. The size of a timing penalty can be constant but

may also depend on the execution state. We consider the property that a particular instruction will not cause a particular timing accident as a safety property. The occurrence of a timing accident thus violates a corresponding safety property.

▸ Use an appropriate method for the verification of safety properties to prove that for individual instructions in the program some of the potential timing accidents will never happen. Reduce the worst-case execution-time bound for an instruction, which a sound WCET analysis would have to assume, by the penalties for the excluded timing accidents.

▸ Abstract interpretation[3] is a powerful method to prove safety properties. Use it to compute certain invariants at each program point, namely an upper approximation of the set of execution states that are possible when execution reaches this program point. Derive safety properties, that certain timing accidents will not happen, from these invariants.

This method for the micro-architectural analysis was the central innovation that made our WCET analysis work and scale.

All types of persistence are identified by the occurrence of certain patterns in memory access traces, while categorizing cache analyses, that is, *must* or *may* analyses only abstract from the last state in a trace.

Our First Illusions

Christian Ferdinand had finished his very fine dissertation on cache analysis in 1997. It still represents the state of the art in cache analysis for LRU caches.[6] Since everybody else working in the area had tried to solve this problem first, and we were convinced that our solution was the best, we felt that we had essentially solved the WCET-analysis problem. Very optimistically we founded AbsInt[a] early in 1998, "we," being five former or actual Ph.D. students and me.

This optimism turned out as wrong in several aspects. Firstly, more or less nobody uses LRU caches in their processors since the logic is considered too complex. Frequently used replacement policies are PLRU, FIFO, random replacement, or even strange looking approximations of random replacement like in the Motorola Coldfire, which is flying in the Airbus A340, and which Airbus selected as a real-life processor to test our approach.[7] This processor had a four-way set-associative cache with a rather strange cache-replacement strategy using one global round-robin replacement counter keeping track of where the next replacement should take place. It meant our cache analysis could essentially only keep track of the last loaded cache line in each cache set. This strange beast of a cache triggered the concept of *timing predictability*, which turned out to be a very fruitful research area, to be discussed later.

There was a second assumption that turned out to be false. It is intuitively clear the accuracy of the cache analysis has a strong impact on the accuracy of the WCET-analysis results. This led us to believe that cache analysis was the difficult part of WCET analysis, that the rest would be easy. It turned out that caches were relatively easy to analyze, although good solutions for Non-LRU caches had yet to be found. Much later, between 2008–2011, Daniel Grund developed an efficient analysis for FIFO caches and approached a solution for the analysis problem for PLRU caches.[9–11]

Pipelines were much more difficult to analyze and also appeared in more variations and were mostly badly documented. And then Airbus informed us about the existence of peripheries and system controllers, architectural components that serious WCET researchers had never heard of. Their analyses can have a very strong influence on the accuracy of timing analyses.

Control-Flow Reconstruction

But first we needed to get programs to analyze. WCET analysis has to be done on the executable level because the compiler influences the execution time by its memory allocation and code generation. So, we needed a reconstruction of the control flow from binary programs. This was part of Henrik Theiling's Ph.D. thesis.[24] Decoding individual instructions is relatively easy, but identifying the control flow, in particular switch tables is a non-trivial task.

Pipeline Analysis

At the end of the 1990s, Stephan Thesing started to develop the framework for modeling pipeline architectures.[16,25]

a www.absint.com

He also did, as far as I know, the first modeling of a system controller for WCET analysis.[26] Doing this precisely was highly important because an imprecise, too abstract model of a systems controller can easily cost an order of magnitude more accuracy than an imprecise pipeline model.

Pipeline analysis is a highly complex part of the overall analysis because, unlike caches, most pipelines do not have compact, efficiently updatable abstract domains. Cache analysis is efficient because the sets of concrete cache states that may occur at a program point can be compactly represented by one abstract cache state, and abstract cache states can be efficiently updated when a memory access is analyzed. Essentially, pipeline analysis uses an expensive powerset domain, that is, it collects sets of pipeline states instead of computing an abstract pipeline state representing sets of pipeline states. This characteristic would, in principle, make it amenable to model checking. Stephan Wilhelm tried this around 2009 and encountered severe problems in the application of model checking to pipeline analysis.[28,29] In particular, interfacing symbolic representations of pipelines with abstract representations of caches, while preserving accuracy, is difficult. Daniel Kaestner's Astrée group made a similar experience when attempting to interface Astrée with some model checkers.[b] It appeared that model checking and abstract interpretation were communicating badly and thus seemed to replicate the behavior of their inventors.

Another excursion into the future: Hahn et al.[13] describes a strictly in-order pipeline providing for compact abstract domains. Strictly in-order pipelines avoid all downstream dependences between consecutive instructions, such as the one of an operand load of an earlier instruction on the instruction fetch of a consecutive instruction. In case of a contention, the operand load is always guaranteed to be executed first. The loss in (average-case) performance compared to a traditional in-order pipeline was measured to be between 6% and 7%,[12] but the analysis efficiency increased by an order of magnitude.

Pipeline analysis is a highly complex part of the overall analysis because, unlike caches, most pipelines do not have compact, efficiently updatable abstract domains.

b https://www.absint.com/astree/index.htm

The Breakthrough

The European project *Daedalus, Validation of critical software by static analysis and abstract testing*, which ran from 2000 to 2002, associated us with an extremely valuable partner, Airbus. Patrick Cousot had organized an industry seminar on abstract interpretation. One of the participants was an Airbus engineer, Famantanantsoa Randimbivololona, in charge of identifying new, usable and helpful methods and tools for software development at Airbus. Randim had listed the most severe of Airbus' problems, and Cousot composed a consortium for a European project targeted at solving these problems. Saarbrücken was listed with *WCET Analysis*. My admiration of the problem-awareness of the software developers at Airbus grew during my first visit to the Airbus headquarters and development labs in Toulouse, France. Everybody greeted me, expressing their concern that they had no viable solution for the verification of their real-time requirements and their hope that we would provide a solution.

This is a good point to describe previous funding. In *DFG Collaborative Research Center* 124, DFG (our National Science Foundation) had funded the development of the foundations of WCET analysis, including Florian Martin's Program Analyzer Generator (PAG). When this DFG Collaborative Research Center, which has run 15 years (from 1983 to 1998) approached its end, DFG had just initiated a new type of grant, a *Transfer Center*, meant to support transfer of results of successful Research Centers to practice. This was a perfect fit for our situation. We applied and were granted solid funding for further development. At about this time, Airbus was searching for a solution for their WCET-analysis problem. Cousot formed the consortium, and the EU Commission granted us the Daedalus project. In hindsight, this sequence of funded projects appears like a miracle, each at exactly the right time!

Back to our contacts with Airbus and their search for some one to solve the WCET problem: They knew their previously used measurement-based

method, also used in certification, did not work any longer for the execution platform selected for the Airbus A380, namely the Motorola MPC755.

The Airbus people provided us with benchmark software, a set of 12 denatured tasks, each consisting of several million instructions, as they were flying them in the A340. The platform there was the Motorola Coldfire processor, mentioned earlier. The tool we developed until 2001 was able to analyze the benchmark provided by Airbus in decent time and with quite precise results. The upper bounds our tool computed made the Airbus people quite happy because they were apporximately in the middle between the worst observed execution times and the upper bound determined by Airbus with a measurement-based method using safety margins. More precisely, our analysis results were overestimating the worst observed execution times by roughly 15%. This breakthrough was reported in Ferdinand et al.[7] Something surprising, at least for me, happened when I described our approach and reported our results at EMSOFT 2001. Highly appreciated colleagues like Hermann Kopetz and Gérard Berry were storming the stage and congratulated me as if I had just won an Oscar. Indeed, this paper received the EMSOFT Test-of-Time Award 2019.

Some words about our long-time cooperation partner, Airbus, in Toulouse. We have experienced the group in charge of developing safety-critical software as problem-aware, highly competent, and extremely cooperative. They wanted to have this problem solved and they trusted us to solve it, and they kept us focused on the real problems, and thus prevented us from solving simplified, self-posed problems, a tendency academic researchers often are inclined to follow.

As a result of our successful development, Airbus offered our tools to the certification authorities for the certification of several Airbus plane generations, starting with the Airbus A380. The European Union Aviation Safety Agency (EASA) has accepted the AbsInt WCET analysis tool as validated tool for several time-critical subsystems of these plane types. We were less successful with Airbus' competitor, who partly certifies their planes themselves, as it recently turned out, and with the certification authority in charge, who doesn't seem to require the use of a sound verification technology for real-time requirements.

Predictability

When modeling the Motorola Coldfire cache we noticed that only one-fourth of its cache capacity could be predicted. This quickly led us to consider the problem of *timing-predictability* of architectures.[14] In his Ph.D. thesis, Jan Reineke developed the first formally founded notion of predictability, namely that of cache predictability.[20,22] The concept behind this notion is that a cache architecture, more precisely its cache-replacement policy is more predictable than another one if it recovers from uncertainty about cache contents faster, that is, needs fewer memory accesses to remove uncertainty from the abstract cache. Among all the considered cache-replacement strategies LRU fares provably best.

Reineke also compared how *sensitive* caches are to changes to the initial cache state. He could show that all non-LRU cache replacement strategies he considered were quite sensitive to such changes. This means the difference in the cache-miss rate is only bounded by the length of the memory-access sequence. Thus, missing an initial cache state when measuring execution time may mean to miss a memory-access sequence with a high cache-miss rate.

In Wilhelm et al,[27] we collected our wisdom concerning timing predictability of several types of architectural components. It heavily influenced the design of the Kalray MPPA.[5]

One could at this point remark that a future automatically driven car will employ a GPU executing learned-pattern recognition, which is controlled by an 8-core-ARM architecture whose design contradicts under almost all aspects this collected wisdom of ours. It employs random-replacement caches, a cache coherence protocol, a shared bus, and all DRAM memory.

Another remark is in place here. Our efforts to push the predictability issue had limited effect. In retrospect, it looks like we came up too early with our complaints and the ideas to remedy the corresponding problems. A few years later hardware security-attacks like Meltdown and Spectre showed that architectural features with low predictability were also the basis for these security attacks. A combination of timing unpredictability and vulnerability to security attacks might have discredited the architectural components more effectively.

AbsInt

WCET analysis for single-core architectures had been essentially solved by a sequence of Ph.D. theses in my group. The only import was the Implicit Path Enumeration Technique (IPET) of Li and Malik.[17] Li and Malik had managed to model the timing behavior of programs and the entire architecture as an integer linear program, which was far from efficiently solvable. The IPET technique was adopted to our setting by Henrik Theiling in his Ph.D. thesis.[24]

It may be valuable information for many readers to estimate the necessary effort to develop a sound formal method for an industrially relevant non-trivial problem more or less from scratch. At the core of our work leading to the first usable tool, as described in Ferdinand et al.,[7] were three Ph.D. theses, those of Christian Ferdinand, Stephan Thesing, and Henrik Theiling. Their joint development effort would amount to approximately 11 person years, ignoring the effort required to write up the theses, publish results, and satisfy project requirements. Another three years went into the implementation of static cache analysis for some non-LRU caches and into the value analysis of the timing-analysis tool. The latter was based on existing theory developed in Cousot and Cousot[3] but had to be adapted to the analysis of binary executables and to all the peculiarities of a machine semantics. Altogether the effort invested in the first usable tool would add up to roughly 14 person years. However, one should not underestimate the accelerating effect that PAG[19] had on the implementation of and experimentation with several abstract interpretations within the timing-analysis tool. This development effort was followed by work to improve efficiency of the analyses and accuracy of the results, by research into the predictability of architectural features and, based on the

results, into ways to exploit the configurability of processor architectures.

We had founded AbsInt to industrialize our WCET technology. Well, we solved the problem, we had instantiations for some processor architectures, basically for those that Airbus and their suppliers needed. However, we had to learn that hardly any two potential customers employed the same architecture configuration. The decision for a new platform was taken without considering whether a WCET-analysis existed for this platform. Instantiating our technology for a new, complex platform took a lot of effort, and platforms were not getting simpler! In consequence, such an instantiation was very expensive, which did not raise the motivation of potential customers to buy our WCET tools or order the development of a new instance for their platform. In addition, there existed some competitors, who marketed their measurement-based, unsound timing analysis and often forgot to mention the unsoundness of their tool. When companies that developed or integrated hard real-time systems were obliged to show "that they did something about this nasty problem" this unsound, inexpensive solution was sometimes preferred to show "that we do something" (and didn't pay too much for it). So, industrializing and marketing a sound WCET technology, that inherently needed to be expensive, was no promising way to get rich.

However, our development of a sound method that actually solved a real problem of real industry was considered a major success story for the often disputed formal-methods domain. AbsInt became the favorite partner for the industrialization of academic prototypes. First, Patrick Cousot and his team offered their prototype of Astrée, a static analysis for run-time errors, which in cooperation with some of the developers has been largely extended by AbsInt. Then, Xavier Leroy offered the result of his much-acclaimed research project, CompCert, the first verified optimizing C compiler. Both Astrée and CompCert are now AbsInt products.

Conclusion

My former Ph.D. students and I have solved a relevant, non-trivial problem, namely how to determine reliable and precise upper bounds on execution times of programs. We were not the only ones to attempt this. Why were we more successful than other groups? Essentially the answer is, we had a firm background in formal methods, particularly in abstract interpretation, and abstraction of the execution platform played the decisive role in our approach. Without the right abstraction of the architecture, the search space is just too large. However, there is more to it! WCET analysis consists of many phases. A practically usable WCET-analysis method requires strong solutions to all the subproblems and their adequate interaction. Otherwise, either the effort is too high, or the accuracy is too low. The people at AbsInt did an excellent engineering job to come up with WCET-analysis tools and later also other tools that were usable on an industrial scale. C

References

1. Alt, M., Ferdinand, C., Martin, F., and Wilhelm, R. Cache behavior prediction by abstract interpretation. R. Cousot and D.A. Schmidt, Eds. In *Proceedings of Static Analysis, 3rd Intern. Symp.* (Aachen, Germany, Sept. 24–26, 1996). LNCS 1145, 52–66; https://doi.org/10.1007/3-540-61739-6_33
2. Altmeyer, S., Maiza, C., and Reineke, J. Resilience analysis: Tightening the CRPD bound for set-associative caches. J. Lee and B.R. Childers, Eds. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conf. Languages, Compilers, and Tools for Embedded Systems*, (Stockholm, Sweden, Apr. 13–15, 2010), 153–162; doi:10.1145/1755888.1755911.
3. Cousot, P. and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. R.M. Graham, M.A. Harrison, and R. Sethi, Eds. In *Proceedings of the 4th ACM Symp. on Principles of Programming Languages*, (Los Angeles, CA, USA, Jan. 1977), A8–252; doi:10.1145/512950.512973.
4. Cullmann, C. Cache persistence analysis: Theory and practice. *ACM Trans. Embedded Comput. Syst. 12*, 1 (2013), 40:1–40:25; doi:10.1145/2435227.2435236.
5. de Dinechin, B.D., van Amstel, D., Poulhiès, M., and Lager, G. Time-critical computing on a single-chip massively parallel processor. G.P. Fettweis and W.N., Eds. *Design, Automation & Test in Europe Conf. & Exhibition* (Dresden, Germany, Mar. 24–28, 2014), 1–6: doi: 10.7873/DATE.2014.110.
6. Ferdinand, C. *Cache Behavior Prediction for Real-Time Systems.* Pirrot, 1997; http://d-nb.info/953983706.
7. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., and Wilhelm, R. Reliable and precise WCET determination for a real-life processor. *EMSOFT LNCS, 2211* (2001), 469–485.
8. Ferdinand, C. and Wilhelm, R. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems 17*, 2–3 (1999), 131–181.
9. Grund, D. Static Cache Analysis for Real-Time Systems: LRU, FIFO, PLRU. Ph.D. thesis, Saarland University, 2011.
10. Grund, D. and Reineke, J. Abstract interpretation of FIFO replacement. J. Palsberg and Z. Su, Eds. In *Proceedings of the 16th Intern. Symp. Statis Analysis* (Los Angeles, CA, USA, Aug. 9–11, 2009). LNCS 5673; doi:10.1007/978-3-642-03237-0_10.
11. Grund, D. and Reineke, J. Toward precise PLRU cache analysis. B. Lisper, Ed. In *Proceedings of the 10th Intern. Workshop on Worst-Case Execution Time Analysis*, (Brussels, Belgium, July 8, 2010). *OASICS 15* (2010) 23–35. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany; doi:10.4230/OASIcs.WCET.2010.23.
12. Hahn, S. On Static Execution-Time Analysis - Compositionality, Pipeline Abstraction, and Predictable Hardware. Ph.D. thesis, Saarland University, 2019.
13. Hahn, S., Reineke, J., and Wilhelm, R. Toward compact abstractions for processor pipelines. R. Meyer, A. Platzer, and H. Wehrheim, Eds. In *Proceedings of Correct System Design—Symp. Honor of Ernst-Rüdiger Olderog* (Oldenburg, Germany, Sept. 8–9, 2015). *LNCS 9360*, 205–220; doi:10.1007/978-3-319-23506-6_14.
14. Heckmann, R., Langenbach, M., Thesing, S., and Wilhelm, R. The influence of processor architecture on the design and the results of WCET tools. In *IEEE Proceedings on Real-Time Systems 91*, 7 (2003), 1038–1054.
15. Huynh, B.K., Ju, L., and Roychoudhury, A. Scope-aware data cache analysis for WCET estimation. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symp,* (Chicago, IL, USA, Apr. 11–14, 2011) 203–212; doi:10.1109/RTAS.2011.27.
16. Langenbach, M., Thesing, S., and Heckmann, R. Pipeline modeling for timing analysis. M.V. Hermenegildo and G. Puebla, Eds. In *Proceedings of the 9th Intern. Symp. on Static Analysis* (Madrid, Spain, Sept. 17–20, 2002), LNCS 2477, 294–309; doi:10.1007/3-540-45789-5_22.
17. Li, Y.S. and Malik, S. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conf.* (June 1995), 456–461.
18. Lv, M., Guan, N., Reineke, J., Wilhelm, R., and Yi, W. A survey on static cache analysis for real-time systems. *LITES 3*, 1 (2016), 05:1–05:48; doi:10.4230/LITES-v003-i001-a005.
19. Martin, F. Generating Program Analyzers. Ph.D. thesis. Saarland University, Saarbrücken, Germany, 1999; http://scidok.sulb.uni-saarland.de/volltexte/2004/203/index.html.
20. Reineke, J. Caches in WCET Analysis: Predictability – Competitiveness – Sensitivity. Ph.D. thesis. Saarland University, 2009; https://bit.ly/3enUrXr
21. Reineke, J. The semantic foundations and a landscape of cache-persistence analyses. *LITES 5*, 1 (2018), 03:1–03:52; doi:10.4230/LITES-v005-i001-a003.
22. Reineke, J., Grund, D., Berg, C., and Wilhelm, R. Timing predictability of cache replacement policies. *Real-Time Systems 37*, 2 (2007), 99–122.
23. Shaw, A.C. Deterministic timing schema for parallel programs. V.K. Prasanna Kumar, Ed. In *Proceedings of the 5th Intern. Parallel Processing Symp.* (Anaheim, CA, USA, Apr. 30–May 2, 1991), 56–63. IEEE Computer Society; doi:10.1109/IPPS.1991.153757.
24. Theiling, H. Control Flow Graphs for Real-Time Systems Analysis: Reconstruction from Binary Executables and Usage in ILP-based Path Analysis. Ph.D. thesis, Saarland University, Saarbrücken, Germany, 2003; http://scidok.sulb.uni-saarland.de/volltexte/2004/297/index.html.
25. Thesing, S. Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models. Ph.D. thesis, Saarland University, 2004.
26. Thesing, S. Modeling a system controller for timing analysis. S.L. Min and W. Yi, Eds. In *Proceedings of the 6th ACM & IEEE Intern. Conference on Embedded Software* (Seoul, Korea, Oct. 22–25, 2006), 292–300; doi:10.1145/1176887.1176929.
27. Wilhelm, R., Grund, D., Reineke, J., Schlickling, M., Pister, M., and Ferdinand, C. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Trans. on CAD of Integrated Circuits and Systems 28*, 7 (2009); doi:10.1109/TCAD.2009.2013287.
28. Wilhelm, S. Symbolic Representations in WCET Analysis. Ph.D. thesis, Saarland University, 2012; http://scidok.sulb.uni-saarland.de/volltexte/2012/4914/.
29. Wilhelm, S. and Wachter, B. Symbolic state traversal for WCET analysis. S. Chakraborty and N. Halbwachs, Eds. In *Proceedings of the 9th ACM & IEEE Intern. Conf. Embedded Software.* (Grenoble, France, Oct. 12–16, 2009), 137–146; doi:10.1145/1629335.1629354.

Reinhard Wilhelm (wilhelm@cs.uni-saarland.de) is Professor Emeritus at Saarland University in Saarbrücken, Germany.