Pipeline Modeling for Timing Analysis*

Marc Langenbach[1], Stephan Thesing[2], and Reinhold Heckmann[3]

[1] Saarland University,
thesing@cs.uni-sb.de
[2] Saarland University,
mlangen@cs.uni-sb.de
[3] AbsInt Angewandte Informatik GmbH,
heckmann@AbsInt.com

Abstract. In *hard* real-time systems, the worst-case execution times of programs must be known. Obtaining *safe* upper bounds for these times by measuring actual executions is rarely possible, since the worst case input is normally not known. We apply static program analysis methods to determine an upper bound for the WCET. While this approach is not new, we believe to be the first to have developed a tool that implements these techniques for *all* the features of a real-life, non-trivial processor, the Motorola ColdFire 5307. Our tool is, to the best of our knowledge, the first one that can determine a *safe* and rather precise WCET bound for a processor that has caches and pipelines and performs branch prediction and instruction prefetching.
Our approach to use a pipeline model in the analysis of the processor behavior opens up new perspectives towards a generative analysis approach and can prove helpful in investigating other processor properties. The emphasis of this paper is on the modeling of the pipeline behavior as input to the derivation of a pipeline analysis.

1 Introduction

Real-time systems are computer systems that interact closely with the physical environment. In such a system correctness depends not only on the logical results, but also on the time at which the results are produced. Failure of a safety-critical (also called *hard*) real-time system can lead to severe damage. Such real-time systems require timing validation by a schedulability analysis. All existing techniques for schedulability analysis require as input the worst-case execution time (WCET) of each task of the system.

In modern microprocessor architectures caches and pipelines are key features for improving performance. Caches are used to bridge the gap between processor speed and the access time of main memory. Pipelines enable acceleration by overlapping executions of several instructions. Caches and pipelines can considerably improve performance but often at the expense of reduced predictability:

* This work was partly supported by the RTD project IST-1999-20527 DAEDALUS of the European FP5 program.

The state of the cache and pipeline depends on the execution history. Therefore, classical approaches [22, 24] to WCET prediction are not directly applicable or lead to results overestimating the real execution time by orders of magnitude. At the Ada Deutschland conference, Alfred Roßkopf of EADS presented results showing that a PowerPC 604 running at 300MHz with caches disabled (and thus more predictable) delivers a similar performance on some benchmarks as a Motorola 68020 running at 20MHz, while the PowerPC outperforms the 68020 by a factor of 20 with caching enabled. For applications like fly-by-wire which require a high computing performance, disabling caches is not an option.

In our approach the prediction of WCETs is divided into two phases which are tackled with different methods: *abstract execution modeling* and *program path analysis*. The times to execute program paths are determined using abstract interpretation [4] based on a formal model of the target system. The other phase, program path analysis, is responsible for determining a worst-case execution path. The problem is formulated as an integer linear program (ILP), cf. [27], and solved by *lp_solve*. A more detailed description of the workflow is given in Section 3.

Though we are not the first to present methods for WCET determination for processors with caches and/or pipelines, our tool, we believe, is the first one that models a non-trivial real-life processor, the Motorola ColdFire 5307, taking into account all of its features in combination, like instruction prefetching, branch prediction, caches, the pipeline, and the memory bus. Other tools known to the authors (such as [16], [20], [3]) implement some of the features listed above, but make assumptions that lead to unusuable results in the context of hard real-time systems.

We applied our tool to a real-life benchmark provided by AirBus France in the DAEDALUS project[4], containing realistically sized code modules.

The modeling and implementation of our tool showed that some assumptions or simplifications made by other WCET methods are not valid for the ColdFire, e. g. that a cache-miss always leads to longer execution times than a cache-hit. Indeed, there are cases where a cache-hit results in a higher overall execution time of the program although it is locally faster, cf, section 4.3.

In this paper, we focus on the pipeline model that was used to implement the static pipeline behavior analysis. In using a model to capture the pipeline behavior, we can open up the perspective of generating the pipeline analysis from the abstract model.

In Section 2, we give an overview of work related to this article. Section 3 describes the components of our tool. In Section 4, we present the ColdFire pipeline and our formal model. Section 5 contains some practical results and conclusions on our work.

[4] The DAEDALUS project aims at introducing static program analysis methods into the airplane software validation process. Partners include AirBus France, AbsInt, PolySpace and various academic partners, e.g. ENS, Saarland University, etc.

2 Related Work

2.1 WCET determination

Based on Shaw's timing scheme [24] which focuses on high-level language constructs and does not consider cache or pipeline effects, Lim et al. present in [16, 17] and Hur et al. in [13] an analysis technique accounting for modern processor features. Cache effects are modeled via bookkeeping of first and last references to blocks and reservation tables are used to handle pipeline effects. As the target machine—a MIPS 3000—implements only a simple pipeline, reservation tables whose resources are registers and pipeline stages are sufficient. Results were only reported for toy sample programs. More sophisticated processors featuring out-of-order execution, superscalarity, or set-associative caches are not considered.

Healy et al. [9, 10] present another approach on predicting WCETs in the presence of caches and simple pipelines. In a first step of the analysis a static cache simulator classifies instructions as cache hits or misses. This information is used by a pipeline path analysis that computes the execution time for a sequence of instructions. Loops are handled in a bottom-up manner. Only the simple pipeline of a MicroSPARC is considered and in [9] only direct-mapped caches and simple pipelines are taken into account which can be described by resource usage patterns of instructions. For their experimental results the authors only consider a small direct-mapped cache with small test programs.

Li et al. suggest a solution using integer linear programming [15]. Both cache and pipeline behavior prediction are formulated as one linear program. The i960KB is investigated which has a fairly simple pipeline. So only structural hazards need to be modeled keeping the complexity of the integer linear program moderate. Branch prediction and/or instruction prefetching are not considered at all. Obtaining the ILP modeling for a more complex processor will be difficult. Using their approach for superscalar pipelines does not seem very promising considering the analysis times reported in the article. Nonetheless, the description of the worst-case path through the program via ILP is an elegant method and can be efficient if the size is kept small as is the case in our tool.

Lundqvist and Stenström present an integrated approach to obtain WCET bounds through simulation of the pipeline in [18, 19]. They extend a pipeline simulator to handle unknown values in inputs. With this approach we share conceptual similarities in that we perform a cycle-wise evolution of a pipeline (model). Different from our approach their method is an integrated one, where value analysis for register/memory contents and execution time computation are parts of the same simulation. If the simulation cannot determine a branch condition exactly due to dependencies on unknown (input) values, they have to simulate both branches. Their method does not guarantee termination of the analysis but has the advantage of determining loop bounds and/or recursion bounds for free[5]. However, we feel that their analysis is very costly due to the huge amount of data that has to be kept for each branch they follow. In contrast,

[5] If these do not depend in a non-trivial way on unknown input values.

our method does not keep information like register or memory contents in the pipeline analysis phase. A value analysis can be executed before the cache and pipeline analysis instead. In [19] experiments with a PowerPC-like architecture are made for small example programs. They use an extended PSIM simulator with simple reservation tables for instructions. In all, it is not clear how well their method scales up to programs of realistic sizes.

Narasimhan and Nilsen present a retargetable execution time analyzer for RISC processors in [20]. The target architecture is modeled using an extended MARIL language (see [2]). The generated analyzer takes an assembly program and a path represented by a sequence of labels and computes the time needed to execute that path. Due to the use of MARIL the range of targetable processors is significantly limited. Analyzing assembly programs complicates the integration of instruction and data cache analysis. This leads to a large gap between predicted and measured execution time, especially for larger inputs [20].

In [3] Colin and Puaut present a method to analyze dynamic branch prediction by obtaining classifications for branches and computing an upper bound on the number of mispredictions. The method can only be applied if the input programs adhere to rather strict syntactic constraints, questioning the applicability to real-life examples.

In [5] Ferdinand presents a method for static cache and pipeline analysis based on abstract interpretation. Using this methodology, Schneider et al. developed a pipeline behavior prediction [23] for the SuperSPARC processor. Both analyses are components of a worst case execution time prediction. However, their approach uses a simple reservation table scheme in the pipeline analysis which cannot be extended to more complex architectures. Also, they do not consider prefetching and/or branch prediction and assume the sequence of instruction accesses to be known statically. Ferdinand's cache analysis is integrated into our pipeline analysis, adapted to the ColdFire's cache update semantics.

None of the work mentioned above takes into account the combination of branch prediction, instruction prefetching, speculative execution, or effects caused by, e. g., data accesses colliding with wrap-around cache line fills on the external processor bus. Unfortunately, these features are not orthogonal, but influence each other. Therefore, they *must* be considered together to obtain tight WCET predictions.

Also, modern computers feature separated core and bus clocks, causing new wait effects for external bus accesses, which have not been investigated so far. Most models simplify the memory architecture of the systems they consider; in a real system, the memory space is made up of quite a few regions with different characteristics concerning access timing, access mode[6], and so on. Our tool makes it easy to specify these parameters and the pipeline analysis takes all this into account when examining memory accesses.

[6] E. g., speculative accesses with the PowerPC can be allowed on some regions but disallowed on others.

2.2 Pipeline Modeling

Apart from retargetable timing analysis, pipeline descriptions have been used in code generation and simulation. Early work [2] considered only registers and pipeline stages by the use of reservation tables. Improvements have led to mixed-level languages such as Expression [8]. From a structural specification of hardware resources, pipeline mechanism, and data transfer paths, reservation tables are generated automatically [7], but this approach is not able to model dynamic behavior like out-of-order execution.

Dynamic hardware features are handled in hardware description languages that are used for retargetable simulators. The language LISA [28, 21] uses a refined form of reservation tables called L-charts that can model dynamic scheduling to some extent. The language RADL [25] offers a flexible signal mechanism that seems more promising in the context of modern hardware features. Signals are emitted if a boolean expression on machine state components holds; they are used to influence scheduling behavior. In contrast to our work which also deals with signals, RADL describes a model very close to the silicon level, where, e. g., latches between pipeline stages are defined implicitly and may be accessed bitwise.

3 The ColdFire WCET Tool

The operation of our WCET tool can be roughly divided into two subsequent phases: *execution modeling* and *program path analysis*. In the first phase we use an analyzer to determine the time needed to execute a program path. In the second stage, a worst-case execution path is identified.

Both phases work on a representation of the control flow graph (CFG). This intermediate representation called CRL is generated from an executable. Analyzing an executable has several advantages. After linking, addresses and sizes of instructions are known which is essential for the cache and pipeline analysis. There are no undefined call targets[7] as their definitions have been added during linking. Another big advantage of analyzing executables is that the toolchain for program development must not be changed to introduce new components or modify existing ones[8]. In the airplane industry, e.g., the requirements imposed by the certification authorities for development tools are very strict. Here, a modified toolchain has to be certified again for development, which may be a very time-intensive and costly procedure.

The component that extracts the CFG is called *exec2crl* [26] and can rather easily be adapted to different executable formats/compilers and architectures.

The execution modeling itself is composed of several analyses. First, a value analysis collects information about the contents of registers and accessed memory cells. This information helps to determine the effects of indirect memory accesses

[7] Function pointers are handled by the value analysis.

[8] Some of the approaches mentioned in Section 2 rely on a modified compiler to extract additional information.

on the cache. It is also used to recognize infeasible paths, which improves the results of the following analyses.

A combined cache and pipeline analysis is started next, statically determining all possible combinations of pipeline states and cache states that might occur during program execution. This analysis is presented in more detail in the next section.

While cache analysis could have been performed separately from and before pipeline analysis, this would have required an additional analysis to determine an approximation to the instructions that can be accessed by the program due to the presence of instruction prefetching and branch prediction in the ColdFire, which is dependent on the pipeline execution. But since this would lead to a decrease of precision in the cache analysis and because of other analysis drawbacks caused by the design of the ColdFire cache itself, we integrated these analyses to gain maximum precision.

Value analysis and combined cache and pipeline analysis are based on abstract interpretation and implemented as data-flow analyses using the program analyzer generator **PAG** [1].

To determine the WCET of the input program, the path analysis takes the results of the preceding analyses along with some optional user provided information on loop counts and execution constraints and formulates the problem of finding a longest execution path as an integer linear program (ILP). Solving the ILP using, e. g., `lp_solve` or CPLEX yields the desired upper bound for the WCET. The results can then be visualized with the graph layout tool **aiSee** [12] and interactively explored. This gives the opportunity to see the WCET contributions of all functions and even all instructions in the CFG. Another option is to visualize the evolution of the pipeline cycle by cycle so that one can reproduce the results on any level of abstraction.

4 Pipeline Analysis

The purpose of the combined cache and pipeline analysis is to give an upper bound of the WCET for each *basic block* in the CFG. We do this by statically simulating the execution of the instructions in the CFG.

Since our method is based on abstract interpretation, we start with a concrete semantics of the instruction execution and then derive an abstraction that is implemented as a data-flow analysis in the CFG of the program.

The concrete semantics uses a *model* of the pipeline, in which only the effects that have an impact on the execution timing are present. For instance we are only interested in how long a multiplication instruction takes, but not what its arguments or results are[9].

The nodes of our CFG are individual machine instructions. Since the execution of some instructions (branches, etc.) depends on their successor in the CFG,

[9] On the ColdFire 5307 a multiplication always takes the same time. This is not true on, e. g., the PowerPC 755, where the time for a multiplication depends on the operands.

we associate the transfer functions of the analysis with the edges between nodes. The analyzer moves along an edge e between two nodes (and thus instructions) v and v' when execution of v has completed. This statement may seem trivial, but since the execution of machine instructions is distributed among several pipeline stages and thus several instructions are being executed at the same time, the meaning of the CFG must be clearly defined. As another alternative, one can pass along an edge between v and v' as soon as the execution of v *begins*. However, we choose the first alternative since it is simpler in the presence of branch prediction. Hence, the transfer function of an edge must simulate the effects on the pipeline (model) until the instruction at the source node of the edge has completed execution.

The system state in the concrete semantics is made up of the cache contents and the pipeline state. When abstracting, we use separate abstractions for the cache contents and the pipeline state.

The abstract cache state follows the semantics presented in [5], adopted to the cache replacement strategy used by the ColdFire 5307. There, an abstract cache state can represent memory blocks that are guaranteed to be in the cache (*must* information) or cache blocks that may be in the cache (*may* information). From this, classifications of memory accesses as guaranteed hits or misses (or unknown) can be derived. The abstraction that can be found for the ColdFire cache, which is a 8KB, 4-way associative unified instruction/data cache with a 'pseudo round-robin' replacement [14] suffers badly from some worst-case features of the hardware. In essence, we use an abstract cache model equivalent to the *must* information of a 2KB direct mapped cache, cf. [6, 11]. This means that we can never classify memory accesses as guaranteed cache misses. Also, the (virtual) size reduction of the cache reduces the amount of guaranteed cache hits, which has the potential to lead to a very pessimistic analysis.

In order to introduce the pipeline model, we give an overview of the ColdFire 5307 in the next section, followed by the model and the abstraction.

4.1 The ColdFire MCF 5307 Pipeline

The ColdFire 5307 is a successor to the well known M68K architecture, inheriting most of its instructions. Some restrictions on instruction formats and sizes have been made to enable a fast implementation. The MCF 5307 features a set of integrated peripherals, like DMA engines and serial ports. It is a very popular microcontroller used in many applications, e.g. the AirBus planes.

The pipeline of the ColdFire MCF 5307 consists of a *fetch pipeline* where instructions are fetched from memory (or the cache), and an *execution pipeline* where instructions are executed (cf. Figure 1). Fetch and execution pipeline are decoupled by a FIFO instruction buffer that can hold at most 8 instructions.

The MCF 5307 has a bus hierarchy to access memory. The pipeline is connected to the K-Bus, a fast pipelined bus that connects the cache and an internal 4KB SRAM area to the pipeline. Accesses to this bus are performed by the IC1/IC2 and the AGEX and DSOC stages. On the next level, the M-Bus the internal peripherals are connected. This bus runs at the external bus frequency,

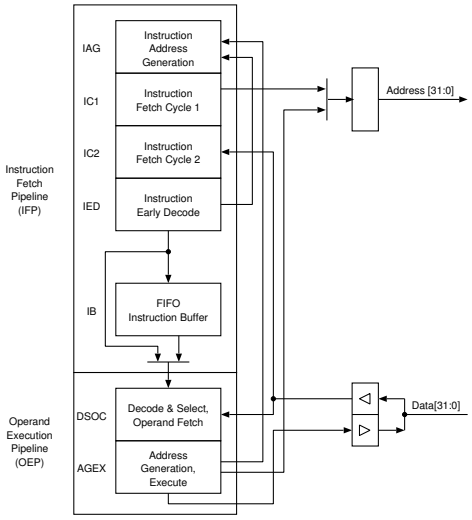

Fig. 1. The Coldfire 5307 Pipeline

while the K-Bus is clocked with the faster internal core clock. The M-Bus, connects to the external bus, which accesses off-chip peripherals and memory.

The ColdFire cache is a non-blocking wrap-around line-fill cache, which means that on a cache miss the requested longword is fetched first from external memory and delivered to the pipeline, followed by the remaining three words from that line.

The fetch pipeline performs *branch prediction* in the IED stage, redirecting fetching long before the branch reaches the execution stages. The fetch pipeline must be stalled if the instruction buffer is full, or if the execution pipeline needs the bus for a memory access. All these stalls cause the pipeline to wait for one cycle, then the stall condition is checked again.

The fetch pipeline is also stalled if the memory block to be fetched is not in the cache (cache miss). In this case, the pipeline must wait until the memory block is loaded into the cache and forwarded into the pipeline. The instructions that are already in the later stages of the fetch pipeline are forwarded to the instruction buffer.

The execution pipeline finishes the decoding of instructions, evaluates their operands, and executes the instructions.

We assume that each kind of operation follows a fixed schedule, telling how many cycles it needs and in which cycles memory is accessed[10]. The execution time varies between 2 cycles and several dozen cycles. Pipelining admits a max-

[10] In fact, there are some instructions like `MOVEM` whose execution schedule depends on the value of an argument given as immediate constant. These instructions can be handled by special means.

imum overlap of 1 cycle between consecutive instructions: the last cycle of each instruction may overlap with the first of the next one. In this first cycle, nothing important happens (no memory access, and no control-flow alteration). Thus, cache and pipeline cannot be affected by two different instructions in the same cycle.

The execution of an instruction is delayed if memory accesses lead to cache misses. Misaligned accesses lead to small time penalties of 1–3 cycles. Store operations are delayed if the distance to the previous store operation is less than 2 cycles. (This does not hold if the previous store operation was issued by a `MOVEM` instruction.) The start of the next instruction is delayed if the instruction buffer is empty.

4.2 A Formal Pipeline Model

On a detailed level, the processor pipeline can be described as a big state machines with clock-cycle wise transitions from pipeline state to pipeline state. A pipeline state contains all timing relevant information of the processor and the number of transitions from the beginning of the execution of an instruction until its end gives the execution time of that instruction. The transitions between states and the states themselve can be very complicated, and writing them down results in a complicated construct, difficult to handle and to implement.

In our formal model introduced here, a concrete pipeline state consists of several *units* with inner *states* that communicate with one another and the memory via *signals*, and evolve cycle-wise according to their inner state and the signals received. Thus, the big flat pipeline state is divided into (disjoint) portions that are encapsulated into units.

The decomposition into units accounts for reduced complexity and easier validation of the model. Units often map directly to pipeline stages, but also may represent more than one stage or introduce virtual pipeline stages that are not present in hardware but facilitate the design of the pipeline model (cf. the store stall timer).

Signals may be *instantaneous*, meaning that they are received in the same cycle as they are sent, or *delayed*, meaning that they are received one cycle after they have been sent. Signals may carry data, e.g. a fetch address. Note that these signals are only part of the formal pipeline model. They may or may not correspond to real hardware signals.

By seperating the complex state into smaller (disjoint) units, complexity is reduced and implementation is made easier. The instantaneous signals between units are now used to transport information between the corresponding portions of the flatted out state. The state transitions are coded in the evolution rules local to each unit.

Figure 2 shows the formal pipeline model for the ColdFire MCF 5307. The following units exist: IAG (instruction address generation), IC1 (instruction fetch cycle 1), IC2 (instruction fetch cycle 2), IED (instruction early decode), IB (instruction buffer), EX (execution unit), SST (store stall timer). In addition, there is a *bus unit* modeling the busses that connect the CPU, the static RAM, the

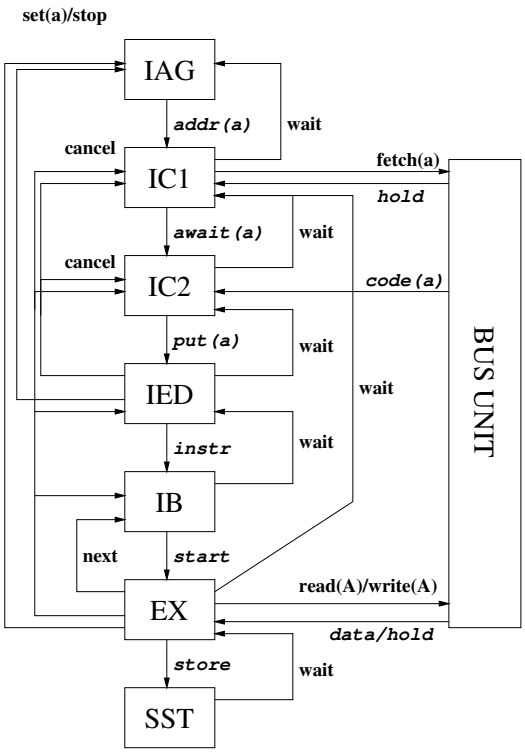

Fig. 2. Map of the formal pipeline model

cache, and the main memory. The signals between these units are shown as arrows. Most units directly correspond to a stage in the real pipeline, but the SST unit is used to model the fact that two stores must be seperated by at least two clock cycles by implementing a (virtual) counter. Furthermore, the two stages of the execution pipeline are modeled by a single stage, EX, because instructions can only overlap by one cycle.

The inner states and emitted signals of the units evolve in each cycle. The complexity of this state update varies from unit to unit. It can be as simple as a small table, mapping pending signals and inner state to a new state and signals to be emitted, e.g. for the IAG unit. It can be much more complicated, if multiple dependencies have to be considered, e.g. the instruction reconstruction and branch prediction in the IED stage. In this case, the evolution is formulated in pseudo code.

Full details on the model can be found in [11].

4.3 Pipeline States

Concrete Pipeline States are formed by combining the inner states of IAG, IC1, IC2, IED, IB, EX, SST, and bus unit plus additional entries for pending signals into one big state. This big state evolves from one cycle to the next. Practically, the evolution of the big pipeline state can be implemented by updating the functional units one by one in an order that respects the dependencies introduced by input signals and the generation of these signals.

Update Function for Pipeline States. For pipeline modeling, one needs a function that describes the evolution of the concrete pipeline state while traveling along an edge (v, v') of the control-flow graph. This function can be obtained by iterating the cycle-wise update function of the previous paragraph.

A concrete pipeline state at v has an empty execution unit EX. It is updated until an instruction is sent from IB to EX. Updating of the concrete pipeline state continues using the knowledge that the successor instruction is v' until EX has become empty again. The number of cycles needed from the beginning until this point can be taken as the time needed for the transition from v to v' for this concrete pipeline state.

Abstract Pipeline States. The abstraction we use for pipeline states is just a *set* of concrete pipeline states, since the amount of possible pipeline states for one instruction tends to be tolerable. Hence, our abstraction computes an upper bound to the collecting semantics.

When considering the abstract update of abstract pipeline states, i.e. the data flow analysis with the transfer functions at the edges of the CFG, it can no longer be assumed that an update is *deterministically*, as has been assumed in the concrete pipeline update. Consider for instance indirect load and store operations, where the address that is read or written is not known in the analysis. Accessing an unknown memory address leads to a cache update with unknown address, destroying the contents of the cache state. The time until a read request to an unknown address is served may vary wildly depending on whether a cache hit or cache miss results and on the memory regions present in the system (cacheable, uncacheable regions, peripherals, etc.). To accommodate all cases, the concrete pipeline state is split into two or more successor states, which must be interpreted independently in the sequel. The number of successor states is determined by the number of memory regions in the system that can be the target of the load/store.

The abstract update for an abstract pipeline state is thus the application of the concrete update on each concrete pipeline state in the abstract state, extended with the possibility of multiple successor states in case of uncertainties.

One may wonder why we do not discard the cache hit successor in the event of an uncertain cache access and only follow the cache miss case. This is because we found out that a cache miss is *not always* the worst case alternative. Since the MCF 5307 has a unified cache and the fetch and execute pipelines

are independent, the following can happen: a data access that is a cache hit is served directly from the cache. At the same time, the fetch pipeline fetches another instruction block from main memory, performing branch prediction and replacing two lines of *data* in the cache. These may be reused later on and cause two misses. If the data access was a cache miss, the instruction fetch pipeline may not have fetched those two lines, because the execution pipeline may have resolved a misprediction before those lines were fetched. On certain occasions one can argue that certain states will not produce a higher execution time than others. But because of the parallel workings of the pipelines and the interactions caused by deferring bus accesses due to core and bus clocks running at different speeds (requiring waits for clock alignment), it is not easy to see (and prove!) that some states are worse than others. But we can always remain on the safe side by considering all pipeline states.

Integrated Cache and Pipeline Analysis. When combining the abstract cache and pipeline state to one big abstract state for the analysis, one has two choices: there can be one abstract cache state per abstract pipeline state, i.e. one abstract cache state representing the cache contents for all concrete pipeline states in the abstract pipeline state, or there can be one abstract cache state per concrete pipeline state.

The first choice saves memory during the analysis but loses precision, since the different concrete pipeline states may cause different memory accesses and thus cache contents, which have to be merged into the one abstract state, losing information.

The second choice is more precise but requires more memory during the analysis. In our implementation, we selected the second option.

The update function belonging to an edge (v, v') of the control-flow graph updates each concrete pipeline state separately. When the bus unit is updated, the pipeline state may split into several successor states with different cache states.

The initial state at the beginning of the analysis is a set of empty pipeline states plus an empty cache state[11]. There must be multiple concrete pipeline states in the initial states, since the adjustment of internal to external clock of the processor is not known in the beginning and every possibility (aligned, one cycle apart, etc) has to be considered. Thus prefetching must start from scratch, but pending bus requests are ignored. To obtain correct results, they must be taken into account by adding a fixed penalty to the calculated worst-case execution time.

We should note that our analysis framework allows us to not only perform traditional intraprocedural data flow analysis but to perform interprocedural and context sensitive analyses. E.g. we can differentiate the iterations of loops by introducing contexts representing the first, second, etc. iteration of the loop.

[11] Which represents a cache with unknown content.

To increase precision, we virtually unroll the loops[12] and analyze each iteration on its own.

5 Results and Conclusions

We implemented the presented pipeline analysis in our WCET tool for the ColdFire MCF 5307. The implementation uses the program analyzer generator PAG [1]. Non-determinism in the analysis only occurs during updates of the bus unit, if memory accesses cannot precisely be classified as cache hits or if the access address is not precisely known.

Our tool was tested with a specific hard real-time benchmark which has been developed and provided by AirBus. This benchmark is designed in a way to resemble "real" avionics software. It basically consists of the so-called cyclic executable that reacts to external events by triggering one of 12 independent tasks. Each single task has to satisfy timing constraints.

Analysis times for all of the tasks in the benchmark are below 30 minutes on a 1.0GHz Athlon running RedHat Linux with 1.5GB memory. The benchmark itself is under a non-dislosure agreement with AirBus. Characteristics of the benchmark, analysis results and a comparision against WCET results obtained by other methods will be presented at the conference. However, our method compares very well to the ones used by AirBus and accepted by the airworthiness authorities.

Applying our method to toy programs as used by other groups, the predicted WCET differs from measured execution times by a factor of up to two. Compared to the results of other groups that can only handle rather simple architectures, this factor might seem high. But those results were obtained on "analysis-friendly" hardware, which is rarely used in real-life systems. Our method has the advantage that it scales up to real-life sized programs.

Our WCET prediction tool for the Motorola ColdFire MCF5307 processor has been installed in AirBus Toulouse plant. The initial assessment AirBus software verification specialists carried out was positive. AirBus is currently in the process to decide to use the WCET technology for the timing validation of avionics software (now evolving and future aircraft programs).

During the evaluation of the analysis results, we came across several interesting timing effects, e.g. in one loop the first three loop iterations all had widely varying execution times before the execution time stayed the same for the remaining iterations. By following the cycle-wise pipeline evolution, we discovered that instruction prefetching and instruction/data cache interference were responsible for this effect.

Our approach of separating the pipeline components into smaller units with inner state and the notion of communication via instantaneous and delayed signals simplified the implementation of a non-trivial pipeline.

[12] Annotations about the number of loop iterations have to be given anyhow for the path analysis, so we can use them here, too.

For the first time, a WCET tool models instruction prefetching and branch prediction together with memory access effects in the bus unit and write buffers while still considering all details and features of an off-the-shelf processor core, the MCF 5307, that are of importance for the WCET determination.

Despite the precise and complex modeling we were able to analyze a real-life benchmark with good results.

Our technique of using a pipeline model as the basis for pipeline behavior prediction also works for even more complicated architectures. We are currently finishing the pipeline analysis tool for the PowerPC 755, a processor that features out-of-order execution, speculation and superscalarity. Also, our technique opens the perspective to a generative approach, where analyses are generated from the specification of a model.

References

1. Martin Alt and Florian Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *Proceedings of SAS'95, Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
2. David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The Marion System for Retargetable Instruction Scheduling. In Brent Hailpern, editor, *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 229–240, Toronto, ON, Canada, June 1991. ACM Press.
3. A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Real-Time Systems, Special issue on worst-case execution time analysis*, 18(2):249–274, 2000.
4. Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York.
5. Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Universität des Saarlandes, 1997.
6. Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of EMSOFT 2001, First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, 2001.
7. Peter Grun, Ashok Halambi, Nikil Dutt, and Alex Nicolau. RTGEN: An Algorithm for Automatic Generation of Reservation Tables from Architectural Descriptions. In *Proceedings on the 12th International Symposium on Systems Synthesis*, 1999.
8. Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *DATE*, 1999.
9. Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1):53–70, January 1999.
10. Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.

11. Reinhold Heckmann and Stephan Thesing. Cache and Pipeline Analysis for the ColdFire 5307. Technical report, Universität des Saarlandes, 2001.
12. http://www.aisee.com. *aiSee Home Page*.
13. Yerang Hur, Young Hyun Bae, Sung-Soo Lim, Sung-Kwan Kim, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Minsuk Lee, Heonshik Shin, and Chong Sang Kim. Worst Case Timing Analysis of RISC Processors: R3000/R3100 Case Study. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 308–319, December 1995.
14. Motorola Inc. *MCF5307 ColdFire Integrated Microprocessor User's Manual*. Motorola Inc., August 2000. MCF5307UM/D, Rev. 2.0.
15. Y.-T. S. Li, S. Malik, and A.Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. *IEEE Real-Time Systems Symposium*, January 1997.
16. Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, and Chong Sang Kim. An Accurate Worst Case Timing Analysis Technique for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7), July 1995.
17. Sung-Soo Lim, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, and Chong Sang Kim. Issues of Advanced Architectural Features in the Design of a Timing Tool. In *Proceedings of the 11th IEEE Workshop on Real-time Operating Systems and Software*, 1994.
18. Thomas Lundqvist and Per Stenström. Integrating Path and Timing Analysis Using Instruction-Level Simulation Techniques. In Frank Mueller and Azer Bestavros, editors, *Proceedings of the ACM SIGPLAN Workshop Languages, Compilers and Tools for Embedded Systems (LCTES)*, volume 1474 of *Lecture Notes in Computer Science*, pages 1–15, 1998.
19. Thomas Lundqvist and Per Stenström. An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution. *Real-Time Systems Journal*, 17(2/3):183–207, November 1999.
20. K. Narasimhan and K. Nilsen. Portable Execution Time Analysis for RISC Processors. 1994.
21. S. Pees, V. Zivojnovic, A. Hoffmann, and H. Meyr. Retargetable Timed Instruction Set Simulation of Pipelined Processor Architectures. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, September 1998.
22. P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 2(1):159–176, 1989.
23. Jörn Schneider and Christian Ferdinand. Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, volume 34, pages 35–44, May 1999.
24. Alan C. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.
25. Chuck Siska. A Processor Description Language Supporting Retargetable Multi-Pipeline DSP Program Development Tools. In *Proceedings of the 11th International Symposium on System Synthesis*, 1998.
26. Henrik Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing Systems and Applications*, Cheju Island, South Korea, 2000.

27. Henrik Theiling and Christian Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain, 1998.
28. V. Zivojnovic, S. Pees, Ch. Schläger, Markus Willems, Rainer Schoenen, and H. Meyr. LISA – Machine Description Language and Generic Machine Model. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, October 1996.