Reliable and Precise WCET Determination for a Real-Life Processor

Christian Ferdinand[1], Reinhold Heckmann[1],
Marc Langenbach[2], Florian Martin[2], Michael Schmidt[2],
Henrik Theiling[2], Stephan Thesing[2], and Reinhard Wilhelm[2]

[1] AbsInt Angewandte Informatik GmbH, Saarbrücken***
[2] Fachrichtung Informatik, Universität des Saarlandes, Saarbrücken†

Abstract. The USES-group at the Universität des Saarlandes follows an approach to compute reliable run-time guarantees which is both well-based on theoretical foundations and practical from a software engineering and an efficiency point of view. Several aspects are essential to the USES approach: the resulting system is modular by structuring the task into a sequence of subtasks, which are tackled with appropriate methods. Generic and generative methods are used whenever possible. These principles lead to an understandable, maintainable, efficient, and provably correct system. This paper gives an overview of the methods used in the USES approach to WCET determination. A fully functional prototype system for the Motorola ColdFire MCF 5307 processor is presented, the implications of processor design on the predictability of behavior described, and experiences with analyzing applications running on this processor reported.

1 Introduction

Real-time systems are subject to stringent timing constraints which are dictated by the surrounding physical environment. Failure of a safety-critical real-time system can lead to considerable damage and even loss of lives. Therefore, a schedulability analysis has to be performed in order to guarantee that all timing constraints will be met ("timing validation"). All existing techniques for schedulability analysis require the worst-case execution time (WCET) of each task in the system to be known prior to its execution. Since this is not computable in general, estimates of the WCET have to be calculated. These estimates have to be safe, i. e., they must never underestimate the real execution time. Furthermore, they should be tight, i. e., the overestimate should be as small as possible.

For processors with fixed execution times for each instruction there are established methods to compute sharp WCET bounds [PK95,PS91]. However, in

*** This work is supported by the RTD project IST-1999-20527 DAEDALUS of the European FP5 programme.

† Work of the USES group (**U**niversity of the **S**aarland **E**mbedded **S**ystems group) is partially supported by TFB 14 of the Deutsche Forschungsgemeinschaft and by the RTD project IST-1999-20527 DAEDALUS of the European FP5 programme.

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 the executions of different instructions. The consequence is that the execution behavior of the instructions cannot be analysed separately since this depends on the execution history. Therefore, the classical approaches to worst-case execution time prediction are not directly applicable or lead to results exceeding the real execution time by orders of magnitude. This may influence the degree of success of timing validations or may lead to a waste of hardware resources. For many mass products, e. g., in the automotiv or telecommunications industries this waste may have a tremendous effect on system costs.

2 The `USES` approach to WCET Computation

2.1 Rationale and Essentials

Several aspects are essential to the `USES` approach:

Modularity: As far as possible for a given processor architecture, the task of determining the WCET is structured into a sequence of subtasks that can be solved in isolation. Interference between processor components may prevent this modularity.

Adequacy of methods: Each sub-task is tackled by appropriate methods. This guarantees efficiency of computation and precision of results.

Genericity: Generic and generative methods are used whenever possible. This allows a large number of different processor architectures and the evolution of processor families to be properly dealt with.

Separation into phases. The determination of the WCET of a program is composed of several different tasks: computation of address ranges for instructions accessing memory by a *value analysis*, classification of memory references as cache misses or hits, called *cache analysis*, predicting the behavior of the program on the processor pipeline, called *pipeline analysis*, and the determination of the worst-case execution path of the program, called *path analysis*. Many of these tasks are quite complex for modern microprocessors and DSPs. Combining them into a single analysis adds complexity.

An arrangement into a sequence of phases would limit complexity. The sequence of the value, cache, pipeline and path analyses was chosen in the first experiments using the SPARC processor [Fer97]. The results of the value analysis were used by the cache analysis to predict the behavior of the (data) cache. The results of the cache analysis were fed into the pipeline analysis allowing the prediction of pipeline stalls due to cache misses. The combined results of the cache and pipeline analyses were used to compute the execution times of program paths. The separation of WCET determination into several phases has the additional effect that different methods tailored to the subtasks can be used. In

our case, the value analysis, the cache analysis, and the pipeline analysis were done by *abstract interpretation*, a semantics based method for static program analysis. Path analysis is done by integer linear programming. The precision of the results and the efficiency of the WCET computation were convincing.

This serialization of subtasks works if there are no cyclic dependencies of processor components on one another which lead to cyclic dependencies of the phases on one another. However, some architectures create dependencies between these tasks which do not allow a straightforward serialization. This is another instance of a *phase ordering problem* encountered frequently in compiler design [WM95].

There are also cases when serialization may lead to a loss of precision which is partly caused by the use of static program analysis, but is unavoidable with any static approach. Cache analysis assumes that all program paths are executable. Cache analysis may collect cache information from non-executable paths thereby corrupting precision, albeit not correctness. Path analysis could model the control flow in such a fine-grained way so as to exclude some of these paths. Hence, there is a slight chance of losing precision by separating cache analysis from path analysis.

Architectural features may ruin the unidirectional dependence between cache analysis and pipeline analysis: for example Motorola ColdFire has two pipelines (a *fetch* and an *execute pipeline*) coupled by an instruction buffer. The order of memory references and, hence, the effects on the cache depend on the pipeline behavior, which in turn depends on the number of prefetched instructions in the buffer.

Generic and generative methods. Several problems can be solved by generic or generative methods. A *generic* method is realized with some formal parameters unspecified. Cache analysis for most processors can be implemented generically by abstracting away from the parameters *capacity, associativity, line size,* and *replacement strategy.* An approach is *generative* if the implementation is generated from some specification by precomputation. For example, the generic cache analysis is generated from a (generic) specification of the domain of *abstract cache states* and a specification of *abstract semantic functions.* The latter was made possible by using abstract interpretation, a well-established method offering a host of available theoretical results. The theory underlying abstract interpretation makes this generative support possible. As context-free parsers are generated using `yacc` or `bison`, analyses can be generated from appropriate specifications by generators. Much of the machinery of each individual analysis is reusable. In our case, the *Program Analysis Generator* `PAG` [Mar99,Mar98] is used.

The generative approach has the advantage of separating specification and implementation. Specifications are more easily understood than hand-coded analyses. Modifying and extending of an analysis by changing the specification is easier than changing hand-written implementations.

Frontends reading assembly or executable code are also generated from descriptions. They translate into a common intermediate language, CRL, described

in [Lan98]. Analyses and transformations of machine programs can then be performed on CRL representations.

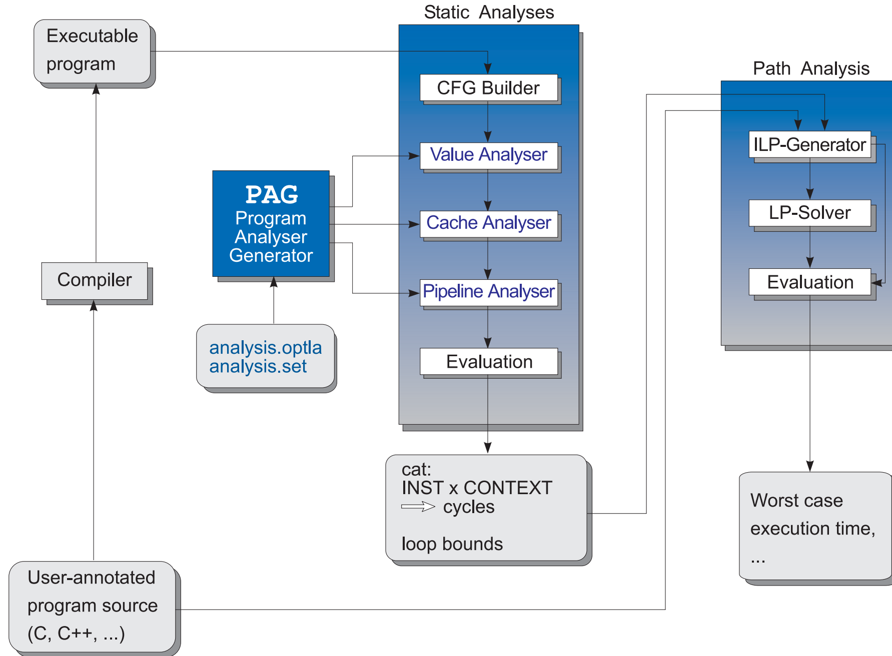

Fig. 1. The structure of the analysis

2.2 Underlying Framework

Composition of the Framework. The starting point of our analysis framework (*see* Figure 1) is a binary program and additional user-provided information about numbers of loop iterations, upper bounds for recursion, etc. Plans have been made to derive more information from the program source if available so that the user has to provide only a minimum of annotations.

In the first step a parser reads the compiler output and reconstructs the control flow [The00,The01]. This requires some knowledge about the underlying hardware, e. g., which instructions represent branches or calls. The reconstructed control flow is annotated with the information needed by subsequent analyses and then translated into `CRL` (Control Flow Representation Language)[1]. This annotated control flow graph serves as the input for microarchitecture analyses:

[1] `CRL` is a human-readable intermediate format designed to simplify analyses and optimizations at the executable/assembly level.

The value analysis (based on [Sic97]) determines ranges for values in registers and by this it can resolve indirect accesses to memory. The cache analysis classifies the accesses to main memory, i. e., whether or not the needed data resides in the cache. The pipeline analysis models the pipeline behavior to determine execution times for a sequential flow of instructions. The result is an execution time for each instruction in each distinguished execution context.

Following from the results of the microarchitecture analyses, the path analysis determines a safe estimate of the WCET. First an ILP generator models the program's control-flow using an integer linear program and a mapping from variable names to basic blocks which will be needed later on. The resulting integer linear program is solved by `lp_solve` [2]. The value of the objective function in the solution is the predicted worst-case execution time for the input program. The ILP solution is then evaluated using the variable mapping resulting in execution and traversal counts for every basic block and edge. The path analysis is generic in such a way that it abstracts from the underlying target machine.

3 Program Analysis to Predict Run-time Behavior

Program analysis is a widely-used technique to determine the runtime properties of a given program without actually executing it. Such information is used, for example, in optimizing compilers [ASU86,WM95] to detect the applicability of efficiency-improving program transformations. A program analyzer takes a program as input and attempts to determine properties of interest. It computes an approximation to an often undecidable or very hard to compute program property.

There is a well-developed theory of program analysis, *abstract interpretation* [CC77,NNH99]. This theory states criteria for correctness and termination of a program analysis. A program analysis is considered an abstraction of the standard semantics of the programming language.

The standard (operational) semantics of a language is given by a *concrete domain* of data and a set of functions describing how the statements of the language transform concrete data. The *abstract semantics* then consists of a (simpler) abstract domain and a set of abstract semantic functions, so called transfer functions, for the program statements computing over the abstract domain.

The designer of a program analysis must define the *abstract domain*. It is obtained from the concrete domain by abstracting from all aspects up to those which are the subject of the analysis to be designed.

Both domains are usually complete partially ordered lattices of values. The partial order in the abstract domain corresponds to precision, i.e. quality of information. By agreement, elements higher up in the order are considered to contain less information[3], i.e. to be less precise.

---

[2] `lp_solve` solves LP, ILP, and MILP problems. The latest version is to be found on ftp://ftp.ics.ele.tue.nl/pub/lp_solve.

[3] Things would work equally well the other way round due to the duality principle of lattice theory.

The partial order determines the *least upper bound* operation, $\sqcup$, on the lattice, which is used to combine information stemming from different sources, e.g. from several possible control flows into one program point.

The designer must also define the *transfer functions*. They describe how the statements transform abstract data. They must be monotonic in order to guarantee correctness and termination.

3.1 Value Analysis

The *value analysis* computes for each processor register an interval of possible values as approximations to the values occurring during runtime. To do this, abstract versions of all processor instructions have to be modeled which are based on interval values as operands. This includes not only simple arithmetic operations like *add* or *mul*, but also complex addressing modes like *register indirect with scaled index* to approximate the effective addresses of memory references.

The $\sqcup$ operator for merging two abstract register or memory cell values is a simple union of intervals:

$$\sqcup : D_{abs} \times D_{abs} \longrightarrow D_{abs}, [l_1, u_1] \sqcup [l_2, u_2] := [min(l_1, l_2), max(u_1, u_2)]$$

Since registers and memory cells have a finite precision, the detection of (possible) overflows requires special attention to compute a *correct* approximation. For example, the *add* instruction is implemented as follows:

$$add : D_{abs} \times D_{abs} \longrightarrow D_{abs},$$
$$[l_1, u_1] + [l_2, u_2] := \begin{cases} [l_1 + l_2, u_1 + u_2] & \text{if no overflow is possible,} \\ \text{unknown} & \text{otherwise} \end{cases}$$

In some cases the value analysis enables the detection of infeasible paths, e.g., for a conditional branch instruction, where the approximated values indicate, that a branch condition always (or never) holds. The information about infeasible paths is forwarded to the cache and pipeline analyses to improve analysis quality by reducing the number of combine operations.

3.2 Cache Analysis

Two analyses have been designed and implemented in different variations corresponding to different cache architectures. A fully associative cache with an LRU replacement strategy is assumend in the following explanations. More complete descriptions that explicitly describe direct mapped and A-way set associative caches are to be found in [Fer97,FMW97,FMWA98]. Section 4 describes a cache architecture with quite a different replacement strategy wich exerts a bad influence on prediction quality.

The *must analysis* determines a set of memory blocks that are in the cache at a given program point whenever execution reaches this point. The *may analysis*

determines all memory blocks that may be in the cache at a given program point. The complement of the information derived by this analysis is used to determine the absence of a memory block in the cache.

These analyses are used to compute a categorization for each memory reference describing its cache behavior. The categories are described in Table 1.

Category	Abb.	Meaning
always hit	ah	The memory reference will always result in a cache hit.
always miss	am	The memory reference will always result in a cache miss.
not classified	nc	The memory reference could neither be classified as ah nor am.

Table 1. Categorizations of memory references and memory blocks.

Must Analysis. The must analysis determines a set of memory blocks that are in the cache at a given program point whenever execution reaches this point. "Good" information is the knowledge that a memory block is in this set. The bigger the set, the better the exploitable information. As we will see, additional information will even tell how long a memory block will remain in the cache at minimum. This is linked to the "age" of a memory block. Therefore, the partial order on the *must*–domain is as follows. Above some abstract cache state in the domain, i.e., less precise, are states where memory blocks in this state are either missing or are older than here. Therefore, applying the $\sqcup$ operator to two abstract cache states will produce a state containing only those memory blocks contained in both and will give them the maximum of their ages in the operand states (see Figure 2). Thus the positions of the memory blocks in the abstract cache state are the upper bounds of the *ages* of the memory blocks in the concrete caches occurring in the collecting cache semantics.

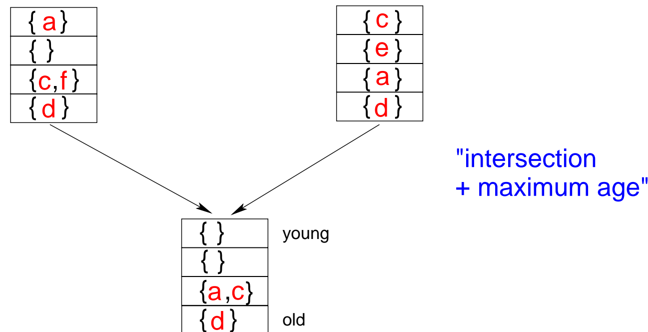

Fig. 2. Upper Bound in the Must Analysis

May Analysis. In order to determine whether a memory block will never be in the cache, the complementary information is computed, i.e., the sets of memory blocks that *may* be in the cache. “Good” information is that a memory block is not in this set, because this memory block can be classified as definitely not being in the cache whenever execution reaches the given program point. Thus, the smaller the sets are, the better. Additionally, the older blocks will reach the desirable situation of being removed from the cache faster than the younger ones. Therefore, the partial order in this domain is as follows: above some abstract cache state in the domain, i.e., less precise, are those states which contain additional memory blocks or where memory blocks in these state are younger than in the given state. Therefore, the $\sqcup$ operator applied to two abstract cache states will produce a state containing those memory blocks contained in at least one of the operand states and will give them the minimum of their ages (see Figure 3).

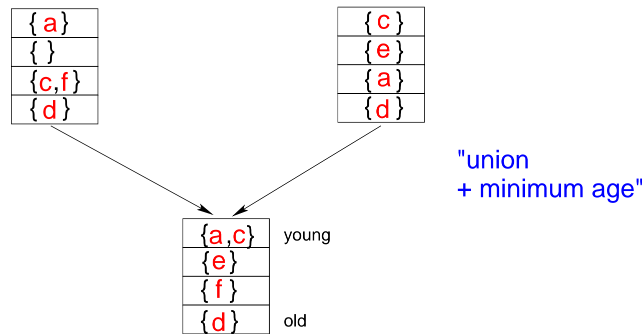

Fig. 3. Upper Bound in the May Analysis

Thus the positions of the memory blocks in the abstract cache state are the lower bounds of the *ages* of the memory blocks in the concrete caches occurring in the collecting cache semantics.

Analysis of Loops and Recursive Procedures. Loops and recursive procedures are of special interest, since programs spend most of their runtime there. Treating them naïvely when analyzing programs for their cache and pipeline behavior will result in a high loss of precision.

Frequently, the following observation can be made: the first execution of the loop body usually loads the cache, and subsequent executions find most of their referenced memory blocks in the cache. Hence, the first iteration of the loop often encounters cache contents quite different from those of later iterations. This has to be taken into account when analyzing the behavior of a loop on the cache. A naïve analysis would combine the abstract cache states from the entry to the loop and from the return from the loop body, thereby losing most of the

contents. Therefore, it is useful to distinguish the first iteration of loops from the others.

In the USES approach, a method has been designed and implemented in the program analyzer generator PAG, which virtually unrolls loops once, the so-called VIVU approach [MAWF98]. Memory references are now considered in different execution contexts, essentially nestings of first and non-first iterations of loops. Experiments have shown VIVU approach to have enormously increased precision [Fer97].

The program analysis techniques thus analyze instruction/context pairs of an input program. A context represents the execution stack, i.e., the function calls and loops along the corresponding path in the control-flow graph to the instruction. It is represented as a sequence of first and recursive function calls ($call_f_f, call_f_r$) and first and other executions of loops ($loop_l_f, loop_l_o$) for the functions f and (virtually) transformed loops l of a program. $INST$ is the set of all instructions $inst$ in a program. $CONTEXT$ is the set of all execution contexts $context$ of a program. IC is the set of all instruction/context pairs ic.

$$\begin{aligned} CONTEXT &= \{call_f_f, call_f_r, loop_l_f, loop_l_o\}^* \\ IC &= INST \times CONTEXT \\ cat &: IC \rightarrow \{\mathsf{ah}, \mathsf{am}, \mathsf{nc}\} \end{aligned}$$

3.3 Pipeline Analysis

[SF99] describes the foundations of pipeline analysis and an experiment done for the superscalar pipeline of the SuperSPARC I. Again abstract interpretation was used. The way from a concrete pipeline to an abstract pipeline consists of several steps. Most complex pipelines are badly documented by the manufacturers. To be of the safe side, the initial model of the pipeline already reflects pessimistic assumptions about undocumented features. A further pessimizing step may be necessary to simplify the representation of pipeline states. The abstract state is finally a superset of this type of pessimisms of “concrete” pipeline states which could actually occur at this program point.

The abstract pipeline update function reflects what happens when the pipeline is advanced by one step. It takes into account the current set of pipeline states, in particular the resource occupancies, the contents of the prefetch queue, the grouping of instructions, and the classification of memory references as cache hits or misses.

At control-flow merging points, the two abstract pipeline states are combined by set union. This leads to supersets of concrete pipeline states being computed. Section 4 will describe a very complex pipeline model that is the basis for the ColdFire analyzer.

4 Analyzing the ColdFire 5307

The ColdFire family of microcontrollers is the successor of the well known M68k architecture of Motorola. The ColdFire 5307 is an implementation of the Version

3 ColdFire architecture, see [Col97]. It contains an on-chip 4K SRAM, a unified 8K data/instruction cache, and a number of integrated peripherals.

It implements a subset of the M68k opcodes, restricting opcodes to two, four or six bytes, thereby simplifying the decoding hardware. The CPU core and the external memory bus can be clocked with different speeds (e.g. 20 MHz bus clock and 60MHz internal core clock).

In the DAEDALUS project, we developed and implemented a complete WCET analysis for the MCF 5307, consisting of a value analysis, an integrated cache and pipeline analysis and a path analysis. In addition to the raw information about the WCET, several parts can be visualized by the **aiSee** tool [aiS] to view detailed information delivered by the analysis.

4.1 The ColdFire Cache

The MCF 5307 has a unified data/instruction cache of 8K. The cache is organized into 128 sets with 4 lines of 16 bytes in each set. Each line contains an additional tag, containing the upper address bits of the block stored in that line and status bits denoting whether that line contains a block or whether the data in that line has been modified.

Caching behavior can be configured for up to three memory areas. An area can be configured as uncacheable or cacheable with write-back or write-through caching.

Data and instruction accesses are mapped to one of the 128 sets by taking bits $4 \ldots 10$ of the address as a 7-bit index. All four lines in the set are then searched for the desired data on a read access by comparing the tags with the upper address bits of the access. If a read misses, i.e., if the desired data is not in the cache, then the data is loaded from main memory and placed into the cache. The requested word in the cacheline is read first, so that the desired data can be delivered quickly back to the CPU core[4]. If a new line has to be loaded on a cache miss, the MCF 5307 selects the line receiving the new data by the following algorithm:

- If a line of the set does not contain valid data then the new data is placed into that line. If several such lines exist, the one with the lowest number is taken.
- If all lines contain valid data, then a *global replacement counter* of two bits is used to give the number of the line to place the data into. The counter is afterwards incremented by one.

Accesses that hit the cache do not influence the replacement counter.

Example: Assume a program accesses the memory blocks $0, 1, 2, 3, \ldots$, and block i is put in set $i \bmod 128$. Such a scenario corresponds to a linear program without memory access (all data are in registers or in the non-cacheable SRAM

[4] The remaining three words of the line are loaded from main memory in the background.

area). Assume further the program starts with an empty cache. Then blocks 0–127 are put into the first line of each set, blocks 128–255 into the second line, 256–383 into the third line, and 384–511 into the fourth line. The resulting cache state is as follows:

Line 0	0	1	2	3	4	5	···	127
Line 1	128	129	130	131	132	133	···	255
Line 2	256	257	258	259	260	261	···	383
Line 3	384	385	386	387	388	389	···	511

where the columns represent sets. The next memory block 512 is put into set 0. The counter has not been used so far, and still has value of 0, hence block 512 is put into line 0 and replaces block 0. The counter is now set to 1, and so, block 513 is put into line 1 of set 1, replacing block 129. Continuing like this until block 639, the resulting cache state is

Line 0	**512**	1	2	3	**516**	5	···	127
Line 1	128	**513**	130	131	132	**517**	···	255
Line 2	256	257	**514**	259	260	261	···	383
Line 3	384	385	386	**515**	388	389	···	**639**

where the recently added blocks are shown in boldface. Block 640 then replaces 512, 641 replaces 513, etc.

This example shows, that some blocks (like 1 and 128) may stay in the cache forever although they are never referenced again, while other blocks (like 512 and 513) are removed from the cache when their set is referenced for the next time. Although these remarks in their full strength only hold in this especial example, they show that in general, one must take into account that some blocks may survive many cache updates, while others are thrown out immediately.

For our static cache analysis this "pseudo round robin" replacement strategy has serious consequences. Since we cannot define abstract cache states as unions of concrete cache states for space reasons, we have to merge several concrete states into one abstract state, which represents all of them. One might be tempted to abstract the replacement counter in the following manner: if a control-flow join with two incoming states with different replacement counters is encountered during analysis, both counters have to be taken into the resulting state, e.g. counters of '2' and '3' may exist in the incoming states, so that the resulting state would have the set $\{2,3\}$ to denote both possibilities. Unfourtunately, this set of possible counter values never decreases, i.e. we can never reduce the number of possible counter values again. So after three control-flow joins the set may have all four possible elements and will stay so for the rest of the analysis. Apart from control flow joins, not classifiable cache accesses cause the same phenomen: if an access cannot be precisely classified as a cache hit or a cache miss, then we have to consider both possibilities. On a miss, the counter

would increase by one; on a hit it would stay the same as before. So if the counter was '2' before, this results in the set $\{2, 3\}$ afterwards. After another three non-classifiable accesses, this may lead to total loss of counter information. Yet if we do not know anything about the counter, we cannot say precisely how long a cacheline may survive in the cache. As the above example shows, a cacheline may be thrown out again by the next access to the set of that line.

This leads to the consequence that all that can be said about the ColdFire cache is that a cacheline will survive until the next access to the set it is in. This is essentially the behavior of a direct mapped cache with 128 sets (and lines). On the other hand, nothing can be said about which lines may be in the cache. This is because it can never be ensured that a line that may be in the cache has definitely been removed and is guaranteed to no longer be in the cache.

Following these observations, we implemented the cache analysis for the MCF 5307 as a must analysis for a 2K direct mapped cache. Naturally, this reduces the precision of our cache behavior prediction, since we were looking at a cache four times smaller than the actual hardware. On the other hand, the size of the analysis data became much smaller and we were able to integrate the cache and pipeline analysis without the danger of excessive memory consumption.

4.2 The ColdFire Pipeline

The MCF 5307 has two coupled pipelines, cf. Figure 4: a *fetch pipeline* fetches instructions from memory, partly decodes them, performs branch prediction and places the instructions into an instruction buffer, consisting of a FIFO with eight entries (complete instructions). The *execution pipeline* consists of two stages that fetch complete instructions from the instruction buffer, decode them, followed by executing them.

The instruction fetch pipeline performs branch prediction by scanning the fetched instructions at the IED stage and redirecting the fetching if a backward branch or an unconditional branch is detected. Therefore, memory access behavior is heavily dependent on pipeline behavior, since a mispredicted (and thus fetched) branch may cause different memory accesses depending on the time that the branch is actually performed at the execution pipeline's AGEX stage. Luckily, we can precisely model this behavior and its consequences by using a unified pipeline and cache analysis.

Our abstract pipeline state consists of a set of concrete pipeline states, giving a safe approximation of the set of all possible pipeline states that may occur at any given program point. From this we can compute an upper bound of the number of cycles the instruction at that points needs to execute in the worst case.

We now have to model the effects of performing one cycle of actual pipeline execution. The state and interactions of the different pipeline stages have to be considered. In our model the different stages of the pipeline have an inner state and communicate with one another through two types of *signals*: immediate signals take effect in the same cycle, e.g. model the cancellation of certain stages by others. Delayed signals take effect only in the *next* cycle and are used among

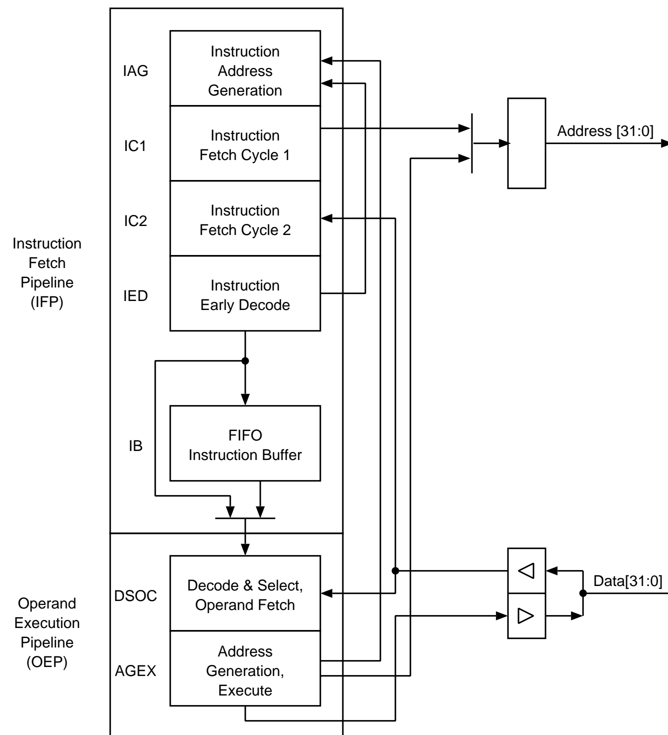

Fig. 4. The MCF 5307 Pipeline

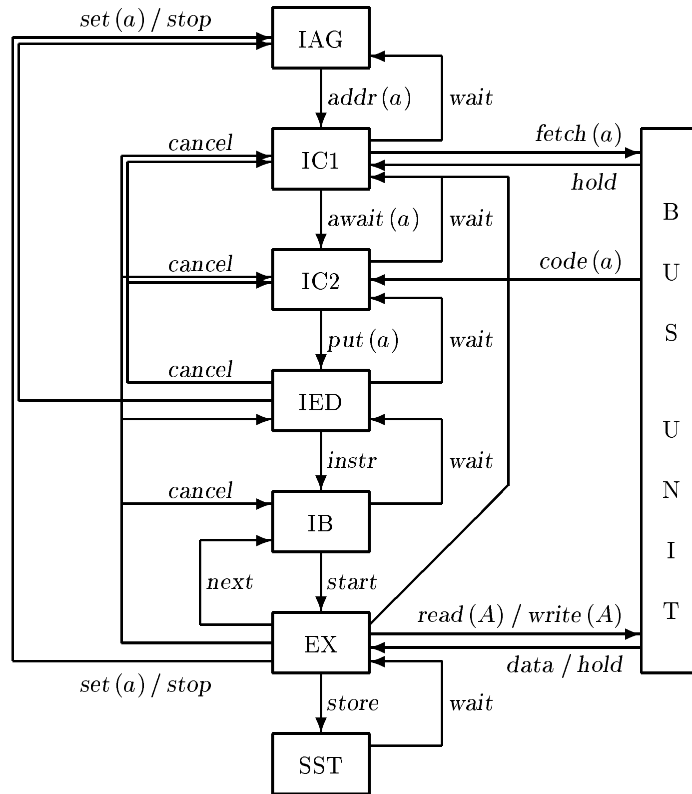

Fig. 5. Map of formal pipeline model

others to tell the IED stage that an instruction word has been fetched by the IC2 stage.

The model is depicted in Figure 5. Note that we modeled the execution pipeline in a way different to the actual hardware. Since ColdFire instructions can overlap by 1 cycle at most and the DSOC stage only performs decoding and register fetching for the first cycle, we don't have to model both execution stages. We added another stage, SST, to model a stall that occurs in the pipeline if two write operations follow immediately (the second is stalled for up to two cycles).

Not shown in detail in Figure 5 is the bus unit that is responsible for modeling the fetching and writing of data. It is rather complex since many stall conditions and interactions on the pipelined ColdFire bus have to be taken into account. This stage also incorporates the cache analysis, updating the cache state according to the accesses made.

The evolution of the pipeline is performed at cycle granularity, i.e. one cycle of execution is modeled. Each stage evolves in that cycle according to certain update rules, taking into account the immediate and delayed signals. The internal state of the stage is updated, and signals may be generated. The update is

performed in the following order: SST, EX, IB, IED, IC2, IC1, bus unit, and IAG. This way, all the dependencies of immediate signals are obeyed.

The implementation very closely follows the formal model, with state and updates of the different stages.

The output of the pipeline analysis is the number of cycles a basic block takes to execute, for each context. These results are then fed into the path analysis to obtain the WCET for the whole program.

4.3 Experiments

The prototype system for the Motorola ColdFire MCF 5307 processor 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 an infinite loop that reacts onto external events by triggering one of 12 independent tasks. Each single task has to satisfy restrictive timing constraints.

Since the tasks are independent from each other, each task was extracted from the control software and analyzed separately for our experiments. The original binary in *.elf* format is about 100kB in size, with the extracted tasks being nearly equal in size.

Value Analysis: Table 2 shows a quality statistics for the value analysis of each task. Memory references are distinguished as read and write accesses. The table shows that the major portion of memory accesses can be resolved *exactly*. References are classified as *good* if the interval for the start address of the access can be restricted to an area of 16 or less cachelines, which is a range of 256 bytes for the MCF 5307. Nothing can be said about the start address of references classified as *unknown*[5]. The analysis time for a single task is a maximum of 61 seconds with a memory usage of about 25MB (computed on a 1.2GHz Athlon system running RedHat Linux).

Cache and Pipeline Analysis: The combined Cache and Pipeline analysis of the EADS benchmark was analyzed in two memory configurations: the *original* configuration partitions the memory in cacheable and non-cacheable areas as well as different memory types (DRAM and On-Chip-SRAM). The *all-cacheable* configuration assumes a simple cacheable DRAM scheme for the whole address space.

Table 3 shows the analysis times for all tasks (computed on a 1.0GHz Athlon running RedHat Linux). The analyzer consumed at most 110MB of memory for every task. The output of the Cache and Pipeline analysis contains the number of cycles for each basic block the number to execute (for each *context*).

[5] Note that the percentage values do not refer to pure instructions, but to instruction/context pairs. This increases the weight of instructions in deeply nested loops or recursive functions.

Task	read accesses [%]			write accesses [%]			unreachable [%]	time [seconds]
	exact	good	unknown	exact	good	unknown		
1	93.32	3.69	2.99	94.51	4.11	1.38	9.1	61
2	93.42	4.01	2.57	93.39	4.62	1.99	8.5	24
3	92.58	4.20	3.22	93.25	4.59	2.16	9.3	31
4	90.24	5.61	4.15	91.71	6.35	1.94	13.5	21
5	93.79	3.33	2.88	94.61	3.83	1.56	7.0	59
6	93.23	4.21	2.56	93.02	4.82	2.16	10.2	26
7	92.06	4.63	3.31	93.13	5.01	1.86	11.0	35
8	90.87	5.03	4.10	91.97	5.88	2.15	11.4	18
9	93.97	3.22	2.81	94.74	3.72	1.54	6.8	56
10	92.40	4.72	2.88	92.79	5.32	1.89	11.0	26
11	92.16	4.47	3.37	92.98	4.90	2.12	9.4	31
12	90.51	5.28	4.21	91.57	6.16	2.27	11.5	24

Table 2. Results of the value analysis.

5 Conclusion

We have given an overview to the methods for cache, pipeline, and path analysis used in the `USES` approach. We have presented the design of a fully functional prototype to analyze the timing behavior of the Motorola ColdFire 5307 processor and reported our experiences with this tool.

The WCET system 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 (coming soon and future aircraft programs).

References

[aiS] http://www.aisee.com. *aiSee Home Page.*

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[Col97] Motorola. *Coldfire Microprocessor Family Programmer's Reference Manual*, 1997.

[Fer97] Christian Ferdinand. Cache Behavior Prediction for Real-Time Systems. PhD Thesis, Universität des Saarlandes, 1997.

[FMW97] Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1997.

Task	memory configuration	
	original [m:s]	all-cacheable [m:s]
1	5:25	8:10
2	4:32	6:47
3	6:03	8:26
4	7:37	10:32
5	4:55	7:32
6	4:34	6:56
7	4:49	8:14
8	7:35	10:31
9	4:54	7:25
10	4:32	6:54
11	5:48	8:09
12	7:35	10:26

Table 3. Analysis times of the Cache and Pipeline analysis

[FMWA98] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache Behavior Prediction by Abstract Interpretation. *Science of Computer Programming, Elsevier*, 1998.

[Lan98] Marc Langenbach. CRL – A Uniform Representation for Control Flow. Technical report, Universität d. Saarlandes, 1998.

[Mar98] Florian Martin. PAG—an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1), 1998.

[Mar99] Florian Martin. *Generation of Program Analyzers.* PhD thesis, Universität d. Saarlandes, 1999.

[MAWF98] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In Kai Koskimies, editor, *Proceedings of the 7th International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[PK95] P. Puschner and C. Koza. Computing Maximum Task Execution Times with Linear Programming Techniques. Technical Report, Technische Universität Wien, Institut für Technische Informatik, Vienna, 1995.

[PS91] Chang Yun Park and Alan C. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *IEEE Computer*, 24(5), 1991.

[SF99] Jörn Schneider and Christian Ferdinand. Pipeline Behavior Prediction for Superscalar Processors. Technical Report A/02/99, Universität des Saarlandes, 1999.

[Sic97] Martin Sicks. Adreßbestimmung zur Vorhersage des Verhaltens von Daten-Caches. Diploma Thesis, Universität d. Saarlandes, 1997.

[The00] Henrik Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing Systems and Applications*, Cheju-do, South Korea, December 2000.

[The01] Henrik Theiling. Generating Decision Trees for Decoding Binaries. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tools for Embedded Systems*, Snowbird, Utah, USA, June 2001.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. International Computer Science Series. Addison–Wesley, 1995. Second Printing.