Real-Time Systems, 5, 1?? (1999)

Fast and Precise WCET Prediction by Separated Cache and Path Analyses

HENRIK THEILING theiling@cs.uni-sb.de
Department of Computer Science, University of the Saarland, Germany

CHRISTIAN FERDINAND ferdinand@absint.de
AbsInt Angewandte Informatik GmbH, Saarbrücken, Germany

REINHARD WILHELM wilhelm@cs.uni-sb.de
Department of Computer Science, University of the Saarland, Germany

Received

Abstract. Precise run-time prediction suffers from a complexity problem when doing an integrated analysis. This problem is characterised by the conflict between an optimal solution and the complexity of the computation of the solution.

The analysis of modern hardware consists of two parts: a) the analysis of the microarchitecture's behaviour (caches, pipelines) and b) the search for the longest program path. Because an integrated analysis has a significant computational complexity, we chose to separate these two steps. By this, an ordering problem arises, because the steps depend on each other.

In this paper we show how the microarchitecture analysis can be separated from the path analysis in order to make the overall analysis fast. Practical experiments will show that this separation, however, does not make the analysis more pessimistic than existing approaches. Furthermore, we show that the approach can be used to analyse executables created by a standard optimising compiler.

Keywords: Run-time analysis, hard real-time systems, abstract interpretation, integer linear programming (ILP), phase ordering problem.

1. Introduction

Caches are used to improve the access times of fast microprocessors to relatively slow main memories. By providing faster access to recently referenced regions of memory, they can reduce the number of cycles a processor is waiting for data. Caching is used for more or less all general purpose processors and, with increasing application sizes, it becomes increasingly relevant and used for high performance microcontrollers and DSPs.

Programs with hard real-time constraints have to be subjected to a schedulability analysis. It has to be determined whether all timing constraints can be satisfied. The degree of success for such a timing validation (*see* [26]) depends on tight WCET estimations. For example, for hardware with caches, the typical worst-case assumption is that all accesses miss the cache. This is an overly pessimistic assumption which leads to a waste of hardware resources.

For this reason the cache behaviour must be taken into account when finding the WCET. Earlier work (*see* [2, 8–10]) describes how a cache analysis can efficiently be implemented using *abstract interpretation (AI)*.

Modern high performance real-time systems not only make use of caches, but also of pipelined processors. With the same reasoning as for caches, their pipeline behaviour must be predicted before run-time in order to obtain tight WCET estimations.

To find the worst-case execution path of the program under examination, integer linear programming techniques have become popular, because they yield very tight results and they are capable of handling the problem of infeasible paths very well.

The results of the cache and pipeline analyses must be combined with the results of a worst-case program path analysis. Existing approaches either neglect the analysis of modern hardware, or use integrated methods which make the analysis very complex and, therefore, infeasible for large programs.

In our approach we use the fine grained results of the cache and pipeline analysis by AI for the generation of an integer linear problem (ILP) to find the worst-case program path. In this way, each technique is used to solve the problem it can handle best.

There is a dependency problem between the two steps of the analysis, however. The cache and pipeline analyses—the microarchitecture analysis—perform better when knowing about infeasible paths which can then be ignored. On the other hand the program path analysis only performs well if the results of the microarchitecture analysis are known.

For the practical experiments, we used executables produced by a standard optimising compiler in order to provide WCET bounds for executables that meet the goal of exploiting system resources best.

2. Program Analysis by Abstract Interpretation

Program analysis is a widely used technique to determine runtime properties of a given program without actually executing it. Such information is used for example in optimising compilers (*see* [1, 30]) to detect the applicability of program optimisations. A program analyser 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* (*see* [5, 20]). This theory states criteria for correctness and termination of a program analysis. A program analysis is considered an abstraction of a standard semantics of the programming language. Abstract interpretation amounts to performing a program's computations using *value descriptions* or *abstract values* in place of concrete values.

One reason for using abstract values instead of concrete ones is computability: to ensure that analysis results are obtained in finite time. Another is to obtain results that describe the result of computations on a set of possible (e. g., all) inputs.

The behaviour of a program (including its cache and pipeline behaviour) is given by the semantics of the program. A standard (operational) semantics of a language is given by a *domain* of data and a set of functions describing how the statements of the language transform data. To predict the run-time behaviour of a program, we approximate its 'collecting semantics (In [5], the term 'static semantics' is used for this.)'. The collecting semantics gives the set of all program (cache and pipeline, resp.) states for a given program point. Such information combined with a path analysis can be used to derive WCET bounds of programs. This approach has successfully been applied to predict the cache behaviour of

programs (*see* [4, 8, 27]) and can naturally be adapted to predict the pipeline behaviour (*see* [8, 22, 23]).
The approach works as follows:

- In a first step, the *concrete semantics* of programs is defined. The concrete semantics is a simplified (auxiliary) semantics that describes only the interesting aspects of computation but ignores other details of execution, thereby abstracting away from all aspects except for those that are subject to the analysis to be designed. E. g., for the cache analysis, the concrete cache semantics only determines the resulting cache state for a given path in the program, but ignores for example register values. In this way each real state is represented by a concrete state.

- In the next step, an *abstract semantics* that 'collects' all possibly occurring concrete states for each program point is defined. An abstract semantics consists of an abstract domain and a set of abstract semantic functions, so called transfer functions, for the program statements computing over the abstract domain. They describe how the statements transform abstract data. They must be monotonic to guarantee termination. An element of the abstract domain represents sets of elements of the concrete domain. The subset relation on the sets of concrete states determines the complete partial order of the abstract domain. The partial order on the abstract domain corresponds to precision, i. e., quality of information.

  To combine abstract values, a *join* operation is needed. In our case this is the *least upper bound* operation, $\sqcup$, on the abstract domain, which also defines the partial order on the abstract domain. This operation is used to combine information stemming from different sources, e. g. from several possible control flows into one program point.

The abstract semantics is constructed in such a way that an abstract state at a given program point 'represents' at least all concrete states that are included in the collecting semantics (*see* [8]). An abstract state at a program point may additionally represent concrete states that can not occur in any real execution at this point, due to infeasible paths. This can reduce the precision of the analysis but does not affect the correctness (*see* [8]).

The computation of the abstract semantics can be implemented with the help of the program analyser generator PAG (*see* [16]), which allows to generate a program analyser from a description of the abstract domain and of the transfer functions.

3. Cache Memories

A cache can be characterised by three major parameters:

- *capacity* is the number of bytes it may contain.

- *line size* (also called block size) is the number of contiguous bytes that are transferred from memory on a cache miss. The cache can hold at most $n = \mathit{capacity}/\mathit{line\ size}$ blocks.

- *associativity* is the number of cache locations where a particular block may reside.
  $n/\mathit{associativity}$ is the number of *sets* of a cache.

If a block can reside in any cache location, then the cache is called *fully associative*. If a block can reside in exactly one location, then it is called *direct mapped*. If a block can reside in exactly A locations, then the cache is called *A-way set associative*.

The fully associative and the direct mapped caches are special cases of the A-way set associative cache where $A = n$ and $A = 1$ resp.

In the case of an associative cache, a cache line has to be selected for replacement when the cache is full and the processor requests further data. This is done according to a replacement strategy. Common strategies are *LRU* (Least Recently Used), *FIFO* (First In First Out), and *random*.

The set where a memory block may reside in the cache is uniquely determined by the address of the memory block, i. e., the behaviour of the sets is independent of each other. The behaviour of an A-way set associative cache is completely described by the behaviour of its n/A fully associative sets (This also holds for direct mapped caches where $A = 1$.).

For the sake of space, we restrict our description to the semantics of fully associative caches with LRU replacement strategy. More complete descriptions that explicitly describe direct mapped and A-way set associative caches can be found in [2] and in [8].

4. Cache Semantics

In the following, we consider a (fully associative) cache as a set of cache lines $L = \{l_1, \ldots, l_n\}$, and the store as a set of memory blocks $S = \{s_1, \ldots, s_m\}$.
To indicate the absence of any memory block in a cache line, we introduce a new element I; $S' = S \cup \{I\}$.

Definition 1 (concrete cache state). A *(concrete) cache state* is a function $c : L \to S'$.
C_c denotes the set of all concrete cache states.

If $c(l_x) = s_y$ for a concrete cache state c, then x describes the relative age of the memory block according to the LRU replacement strategy and not the physical position in the cache hardware (*see* Figure 1).

The *update* function describes the side effect on the cache of referencing the memory. The LRU replacement strategy is modelled by putting the most recently referenced memory block in the first position l_1. If the referenced memory block s_x is in the cache already, then all memory blocks in the cache that have been more recently used than s_x increase their relative age by one, i. e., they are shifted by one position to the next cache line. If the memory block s_x is not in the cache already, then all memory blocks in the cache are shifted and the oldest, i. e., least recently used memory block is removed from the cache.

Definition 2 (cache update). A cache update function $\mathcal{U} : C_c \times S \to C_c$ describes the new cache state for a given cache state and a referenced memory block.

Updates of fully associative caches with LRU replacement strategy are modelled as in Figure 1.

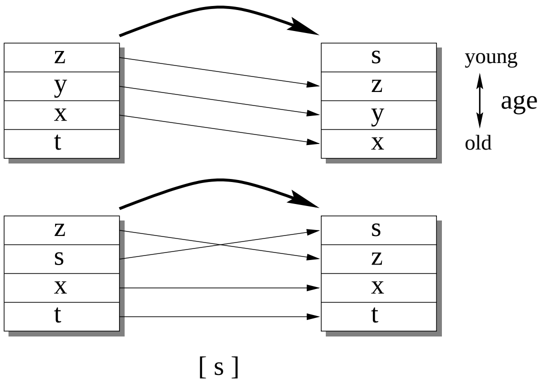


Figure 1. Update of a concrete fully associative (sub-) cache. On the left the old cache is depicted, on the right the new cache after access to s. The picture on top shows an access where s was not in the old cache, the picture on the bottom shows what happens if s was already in the cache.

Control Flow Representation We represent programs by control flow graphs consisting of nodes and typed edges. The nodes represent *basic blocks*. For each basic block, the sequence of references to memory is known (This is appropriate for instruction caches and can be too restricted for data caches and combined caches. *See* [2, 8] for weaker restrictions.), i. e., there exists a mapping from control flow nodes to sequences of memory blocks: $\mathcal{L} : V \to S^*$.

We can describe the working of a cache with the help of the update function $\mathcal{U}$. Therefore, we extend $\mathcal{U}$ to sequences of memory references: $\mathcal{U}(c, \langle s_{x_1}, \ldots, s_{x_y} \rangle) = \mathcal{U}(\ldots(\mathcal{U}(c, s_{x_1}))\ldots, s_{x_y})$.

The cache state for a path $(k_1, \ldots, k_p)$ in the control flow graph is given by applying $\mathcal{U}$ to the initial cache state c_I that maps all cache lines to I and the concatenation of all sequences of memory references along the path: $\mathcal{U}(c_I, \mathcal{L}(k_1) \circ \cdots \circ \mathcal{L}(k_p))$.

Abstract Semantics The domain for our AI consists of *abstract cache states*:

Definition 3 (abstract cache state). An *abstract cache state* $\hat{c} : L \to 2^S$ maps cache lines to sets of the memory blocks. $\hat{C}$ denotes the set of all abstract cache states.

We will present three analyses. The **must analysis** determines a set of memory blocks that are in the cache at a given program point under all circumstances. The **may analysis** determines all memory blocks that may be in the cache at a given program point. The latter analysis is used to guarantee the absence of a memory block in the cache. The **persistence analysis** determines memory blocks that will never be removed from the cache after having been loaded into it.

The analyses are used to compute a categorisation for each memory reference that describes its cache behaviour. The categories are described in Table 1.

Table 1. Categorisations of memory references.

Category	Abbrev.	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.
persistent	ps	The memory reference could neither be classified as ah nor am. But the second and all further executions of the memory reference will always result in a cache hit.
not classified	nc	The memory reference could neither be classified as ah, am, nor ps.

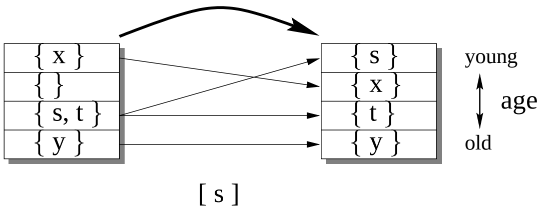


Figure 2. Update of an abstract fully associative (sub-) cache.

The abstract semantic functions describe the effects of a control flow node on an element of the abstract domain. The **abstract cache update** function $\hat{\mathcal{U}}$ for abstract cache states is an extension of the cache update function $\mathcal{U}$ to abstract cache states.

On control flow nodes with at least two (Our join functions are associative. On nodes with more than two predecessors, the join function is used iteratively.) predecessors, *join*-functions are used to combine the abstract cache states.

Definition 4 (join function). A *join function* $\hat{\mathcal{J}} : \hat{C} \times \hat{C} \mapsto \hat{C}$ combines two abstract cache states.

4.1. Must Analysis

To determine whether a memory block is definitely in the cache we use abstract cache states where the positions of the memory blocks in the abstract cache state are upper bounds of the *ages* of the memory blocks. $\hat{c}(l_x) = \{s_y, \dots, s_z\}$ means the memory blocks $s_y, \dots, s_z$ are in the cache. $s_y, \dots, s_z$ will stay in the cache at least for the next $n - x$ references to memory blocks that are not in the cache or are *older* than $s_y, \dots, s_z$, whereby s_a is older than s_b means: $\exists l_x, l_y : s_a \in \hat{c}(l_x), s_b \in \hat{c}(l_y), x > y$.
We use the abstract cache update function depicted in Figure 2.

The join function is similar to set intersection. A memory block only stays in the abstract cache if it is in both operand abstract caches states. It gets the maximal age if it has two different ages (*see* Figure 3A).

The solution of the must analysis computed by the PAG(*see* [17, 28]) generated analysers by fixpoint iteration is interpreted as follows: Let $\hat{c}$ be an abstract cache state at a control

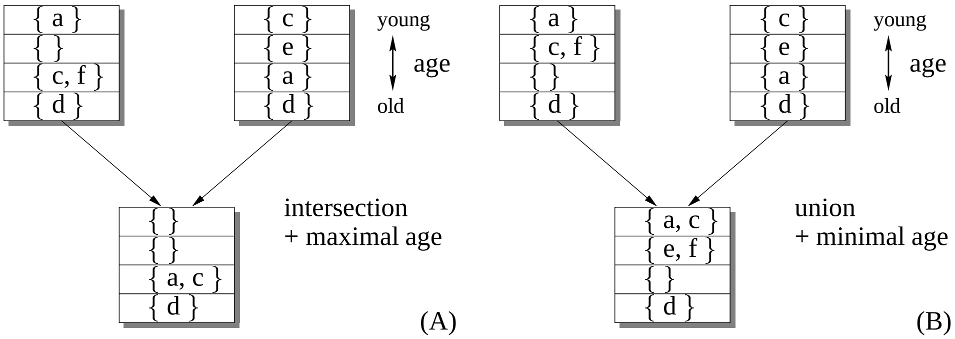


Figure 3. A) Join for the must analysis. B) Join for the may analysis.

flow node k that references a memory block s_x. If $s_x \in \hat{c}(l_y)$ for a cache line l_y then s_x is definitely in the cache. A reference to s_x is categorised as *always hit* (ah).

4.2. *May Analysis*

To determine whether a memory block s_x is never in the cache we compute the set of all memory blocks that *may* be in the cache. We use abstract cache states where the positions of the memory blocks in the abstract cache state are lower bounds of the *ages* of the memory blocks. $\hat{c}(l_x) = \{s_y, \ldots, s_z\}$ means the memory blocks $s_y, \ldots, s_z$ may be in the cache. A memory block $s_w \in \{s_y, \ldots, s_z\}$ will be removed from the cache after at most $n - x + 1$ references to memory blocks that are not in the cache or are *older or the same age* than s_w, if there are no memory references to s_w. s_a is older or same age than s_b means: $\exists l_x, l_y : s_a \in \hat{c}(l_x), s_b \in \hat{c}(l_y), x \geq y$.

We use the following join function: The join function is similar to set union. If a memory block s has two different ages in the two abstract cache states then the join function takes the minimal age (*see* Figure 3B).

The solution of the may analysis computed by the PAG generated analysers is interpreted as follows: Let $\hat{c}$ be an abstract cache state at a control flow node k that references a memory block s_x. If s_x is not in $\hat{c}(l_y)$ for an arbitrary l_y then it is definitely not in the cache. A reference to s_x is categorised as *always miss* (am).

4.3. *Persistence Analysis*

To improve the prediction we use the classification ps (*persistent*) (This improvement is described in [8]), which means that a first execution of a memory reference may result in either a hit or a miss, but all non-first executions will result in hits (This is similar to the 'first miss' classification as found in [19].). An abstract cache state $\hat{c}$ at a control flow node k that references a memory block s_x is interpreted in the following way: If $s_x \in \hat{c}(l_y)$ for $y \in \{1, ..., n\}$ then s_x would not have been replaced if it had been in the cache. If a reference to s_x cannot be categorised as ah, then it is categorised as ps. When a memory block gets

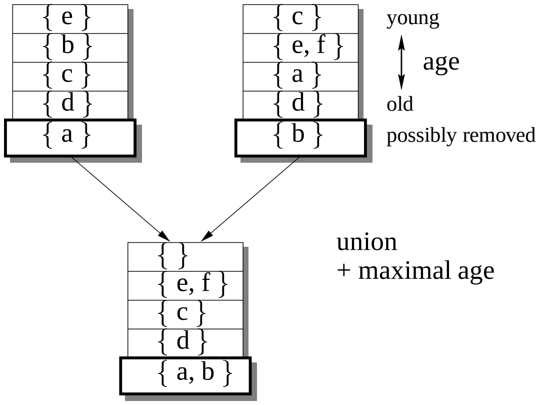


Figure 4. Join for the persistence analysis.

too old, it is moved into an additional virtual cache line $l_\top$ which holds those cache lines that could once have been removed from the cache.

The join function is similar to set union. If a memory block s has two different ages in the two abstract cache states then the join function takes the maximal age (*see* Figure 4).

The solution of the persistence analysis is interpreted as follows: Let $\hat{c}$ be an abstract cache state at a control flow node k that references a memory block s_x. If s_x is not in $\hat{c}(l_\top)$, it can definitely not be removed from the cache. A reference to s_x is categorised as ps.

4.4. Termination of the Analysis

There are only a finite number of cache lines and for each program a finite number of memory blocks. This means the domain of abstract cache states $\hat{c} : L \rightarrow 2^S$ is finite. Hence, every ascending chain is finite. Additionally, the abstract cache update functions $\hat{\mathcal{U}}$ and the join functions $\hat{\mathcal{J}}$ are monotonic. This guarantees that our analysis will terminate.

5. Analysis of Loops and Recursive Procedures

Loops and recursive procedures are of special interest, since programs spend most of their runtime there.

A loop often iterates more than once. Since the execution of the loop body usually changes the cache contents, it is useful to distinguish the first iteration from others.

For our analysis of cache behaviour we treat loops as procedures to be able to use existing methods for the interprocedural analysis (*see* Figure 5).

In the presence of (recursive) procedures, a memory reference can be executed in different execution contexts. An execution context corresponds to a path in the call graph of a program.

```
                                   proc loop$_L$();
  ⋮                                if $P$ then
while $P$ do                         BODY
      BODY           ⟹               loop$_L$();      (2)
end;                               end
  ⋮                                ⋮
                                   loop$_L$();          (1)
                                   ⋮
```

Figure 5. Loop transformation.

The interprocedural analysis methods differ in which execution contexts are distinguished for a memory reference within a procedure. Widely used is the *callstring approach* whose applicability to cache behaviour prediction is limited (*see* [8]).

To get more precise results for the cache behaviour prediction, we have developed the **VIVU approach** (*see* [18]) which has been implemented with the mapping mechanism of PAG as described in [3]. Paths through the call graph that only differ in the number of repeated passes through a cycle are not distinguished. It can be compared with a combination of *virtual inlining* of all non-recursive procedures and *virtual unrolling* of the first iterations of all recursive procedures including loops. The results of the VIVU approach can naturally be combined with the results of a path analysis to predict the WCET of a program.

6. Program Path Analysis

A problem formulated in ILP consists of two parts: the objective function, and constraints on the variables used in the objective function. Our objective function represents the number of CPU cycles in the worst case. Correspondingly, it will be maximised. Each variable in the objective function will represent the execution count of one basic block of the program and will be weighted by the execution time of that basic block. Additionally, we will use variables corresponding to the traversal counts of the edges in the control flow graph.

Looking at the control flow graph of the program under examination, integer constraints describing how often basic blocks are executed relative to each other can automatically be generated. Figure 6 shows how constraints can be generated for a simple `if`-statement.

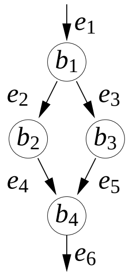
However, additional information about the program provided by the user is usually needed, as the problem of finding the worst-case program path is unsolvable in the general case. In our approach the user only needs to provide bounds for loop iterations and recursion depths.

The ILP approach for program path analysis has the advantage that users are able to describe in precise terms virtually anything they know about the program's control flow. In our system arbitrary integer constraints can be added by the user to further improve the analysis.

```
IF b1 THEN
 b2
ELSE
 b3
END
b4
```

$n_{b_2} = n_{e_2}$

$n_{b_2} = n_{e_4}$



$$n_{b_1} = n_{e_1}$$
$$n_{b_1} = n_{e_2} + n_{e_3}$$
$$n_{b_3} = n_{e_3}$$
$$n_{b_3} = n_{e_5}$$
$$n_{b_4} = n_{e_4} + n_{e_5}$$
$$n_{b_4} = n_{e_6}$$

Figure 6. Constraints that can automatically be generated from the control flow graph. $b_1 \ldots b_4$ are the basic blocks whose execution counts are $n_{b_1} \ldots n_{b_4}$. The basic principle is that the execution count of a basic block equals the sum of the traversal counts of its incoming edges and also traversal counts of its outgoing edges.

As an example, the recursive `isprime` function in one of our test programs was annotated as follows.

```
bool isprime (int i)
{
  if (i<2) return false;
  if (i==2) return true;
  if (i % 2==0) return false;
  for (int o=3; o*o<=i; o+=2) {
    if (i % o == 0)
      return false; //$ (iterations (floor (/ (1- (sqrt i)) 2)))
  }
  return true;
}
```

In our program path analysis we use the VIVU approach described in [8–10, 18] to distinguish first iterations of loops (or calls to functions) and all other iterations (or recursive calls).

The system first generates the obvious constraints automatically and then adds user supplied constraints to tighten the WCET bounds.

6.1. *Generating Constraints*

A description of how to generate constraints for program path analysis with ILP is given in [12]. In the following we will show how to use this technique together with our VIVU analysis to distinguish execution contexts, i. e., how the objective function can be generated, how to generate the constraints that describe the relation between basic blocks and thereby bounding the objective function, and how to incorporate the microarchitecture analysis by AI.

In our approach a basic block is considered in different contexts for which separate constraints are generated. A context is determined by 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 (C$[c]$ and R$[c]$) and first

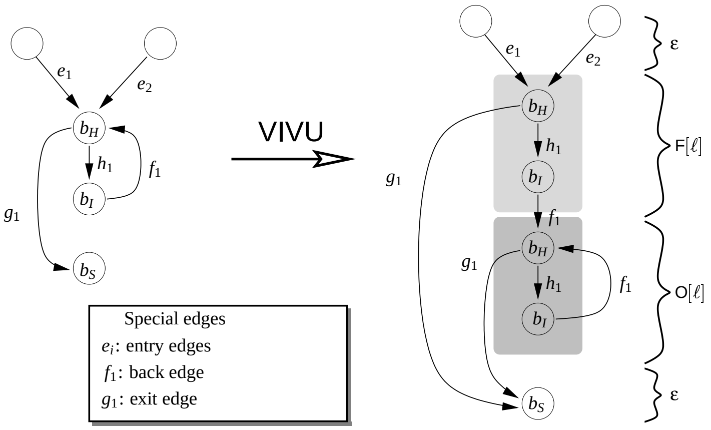


Figure 7. A simple loop (left). Virtually unrolled once on the right.

and other executions of loops ($\mathsf{F}[\ell]$ and $\mathsf{O}[\ell]$) for each basic block c calling a function and each (virtually) transformed loop ℓ of a program.

Definition 5 (Context). Let $calls(\mathcal{P})$ be the set of basic blocks of program $\mathcal{P}$ that contain a call to a function of $\mathcal{P}$. Let $loops(\mathcal{P})$ be the set of loops of $\mathcal{P}$. The set $\mathcal{T}$ of all contexts is a sequence (denoted by *) of context distinctions, thus it is defined as follows:

$$\mathcal{T} \;=\; \Big\{\underbrace{\mathsf{C}[c],\, \mathsf{R}[c],\, \mathsf{F}[\ell],\, \mathsf{O}[\ell]}_{\text{context distinctions}} \mid c \in \ calls\ (\mathcal{P}),\ \ell \in \ loops\ (\mathcal{P})\Big\}^*$$

In Figure 7 a simple loop is shown that will be used to demonstrate the generation of constraints.

Definition 6. Let n_b be the execution count of basic block b and let $\tau(b) \subseteq \mathcal{T}$ be the set of distinguished execution contexts of b. Let n_b^ϑ be the execution counts for each context $\vartheta \in \tau(b)$ and let n_e^ϑ be the traversal count for an edge e in a context ϑ.

Generally, each simple constraint for n_b, like those in Figure 6, will be generated for each n_b^ϑ. Following this general rule, we obtain the following simple constraints for basic block b_H in Figure 7:

$$\begin{aligned} n_{b_H}^{\mathsf{F}[\ell]} &= n_{h_1}^{\mathsf{F}[\ell]} + n_{g_1}^{\mathsf{F}[\ell]} \\ n_{b_H}^{\mathsf{O}[\ell]} &= n_{h_1}^{\mathsf{O}[\ell]} + n_{g_1}^{\mathsf{O}[\ell]} \end{aligned}$$

At basic block b_S we generate the following constraint for its incoming edge:

$$n_{b_S}^{\varepsilon} = n_{g_1}^{\varepsilon}$$

An edge has the same contexts as the corresponding basic block. Therefore, g_1 is considered in two different sets of contexts depending on the basic block it belongs to when the constraint is generated (*see* (7)).

6.2. *Objective Function*

The cache analysis presented earlier in this paper calculated a categorisation (ah, am, nc or ps) for each basic block in each VIVU context. This categorisation together with the results of the pipeline analysis define a worst-case execution time for each basic block in all of its contexts. These execution times are the basis for the objective function of the generated ILP to weight the basic blocks on the paths in the program.

Let t_b^ϑ be the execution time of basic block b in any of its contexts $\vartheta \in \tau(b)$.

The objective is to maximise the execution time given in CPU cycles of the whole program. Thus, for a given worst-case execution path $(b_1, \vartheta_1), \ldots, (b_k, \vartheta_k)$ and their corresponding execution times $t_{b_i}^{\vartheta_i}, i = 1, \ldots, k$ this time is

$$fmax = \sum_{i=1}^{k} t_{b_i}^{\vartheta_i}. \tag{1}$$

To make this function suitable for an objective function in an ILP, a technique called implicit path enumeration (*see* [11]) is used. The execution counts will be used to represent the above formula. These execution counts are related to the worst-case execution path in the following way:

$$\forall b, \vartheta \in \tau(b): \quad n_b^\vartheta = \left|\left\{ i \in \{1, \ldots, k\} \mid (b_i, \vartheta_i) = (b, \vartheta) \right\}\right| \tag{2}$$

Then (1) can be written as

$$fmax = \sum_{b} \sum_{\vartheta \in \tau(b)} t_b^\vartheta \cdot n_b^\vartheta. \tag{3}$$

The next two sections deal with constraints to describe the control flow between blocks with different contexts.

6.2.1. *Loops*

For loops, special constraints are generated. Let b_H be the first basic block executed in a loop ℓ, called the *head* of the loop.

Let e_i be the edges that enter the loop ℓ. For each context ϑ of an arbitrary e_i there must be two contexts of b_H which have the same prefix but either $\mathsf{F}[\ell]$ or $\mathsf{O}[\ell]$ appended:

$$\begin{aligned} &\vartheta \circ \mathsf{F}[\ell] \\ &\vartheta \circ \mathsf{O}[\ell] \end{aligned} \tag{4}$$

The constraints generated for control flow entering ℓ are distinguished by the first and all other iterations. The first iteration of ℓ is naturally entered from outside ℓ. So the constraint for each ϑ looks like this:

$$n_{b_H}^{\vartheta \circ \mathsf{F}[\ell]} = \sum_{e_i} n_{e_i}^{\vartheta} \tag{5}$$

In our example we obtain:

$$n_{b_H}^{\mathsf{F}[\ell]} = n_{e_1}^{\varepsilon} + n_{e_2}^{\varepsilon}$$

Let f_i be the back edges of ℓ, i. e., those edges that jump back to the beginning of the loop from inside the loop. All non-first iterations of loop ℓ are entered from inside the loop via these back edges either from the first or from other iterations. This is described by the following constraint:

$$n_{b_H}^{\vartheta\circ\mathsf{O}[\ell]} = \sum_{f_i} \left(n_{f_i}^{\vartheta\circ\mathsf{F}[\ell]} + n_{f_i}^{\vartheta\circ\mathsf{O}[\ell]} \right) \tag{6}$$

In the example in Figure 7 there is only one back edge, correspondingly, we generate the following constraint:

$$n_{b_H}^{\mathsf{O}[\ell]} = n_{f_1}^{\mathsf{F}[\ell]} + n_{f_1}^{\mathsf{O}[\ell]}$$

We also need special constraints for exiting a loop, so we introduce a constraint for each exit edge g_i of loop ℓ and for each context ϑ:

$$n_{g_i}^{\vartheta} = n_{g_i}^{\vartheta\circ\mathsf{F}[\ell]} + n_{g_i}^{\vartheta\circ\mathsf{O}[\ell]} \tag{7}$$

The context ϑ outside our example loop is the empty context, so we obtain:

$$n_{g_1}^{\varepsilon} = n_{g_1}^{\mathsf{F}[\ell]} + n_{g_1}^{\mathsf{O}[\ell]}$$

This constraint links together the two views on the exit edges from inside and outside the loop ℓ.

6.2.2. Loop Bounds First, we have to define what it means when a loop is executed k times. We used an executable produced by an optimising standard compiler as input for our analysis so this issue is not as easy as it may appear at first sight. In fact we have to identify some *inner* basic block, i. e., one that does not belong to the loop's exit condition check which is usually executed once more than the whole loop. There are various types of loops, those that test the condition at the beginning, e. g. `while() { ... }` in the C programming language, those that test the condition at the end of the loop, e. g. `do { ... } while()` loops in C, or even in the middle of the loop, e. g. if the loop contains a statement like `if ( ... ) break;`.

There are two factors that complicate the analysis. Firstly, C programmers usually like to use `break` and `return` inside loops and, secondly, the compiler sometimes optimises quite fancily. The first problem can be solved by user provided additional constraints.

There are still some open problems with optimising compilers, as can be seen from the experimental results. The compiler optimisation problem could perhaps be solved by the help of the compiler itself. In our case the compiler very often inferred that the first iteration is always executed and, therefore, either unrolled it once or moved the condition test from the beginning to the end of the loop.

Because the problem of finding an inner basic block of a loop couldn't be solved in all the cases, and because we have to guarantee that the analysis doesn't calculate time bounds that are below the WCET, we had to leave the problem of defining which basic block is an inner basic block of the loop with the user. We will therefore assume that b_I is the first basic block of the body of the loop. Then b_I is executed as many times as the loop is iterated.

Let e_i be the entry edges to ℓ. Let k be the maximal number of iterations of ℓ. Then we use the following constraint for each context ϑ of e_i to bound the number of iterations of ℓ:

$$n_{b_I}^{\vartheta\circ\mathsf{F}[\ell]} + n_{b_I}^{\vartheta\circ\mathsf{O}[\ell]} \leq k \cdot \sum_{e_i} n_{e_i}^{\vartheta} \tag{8}$$

Assuming our example loop will be executed 10 times at the most, we generate the following constraint:

$$n_{b_I}^{\mathsf{F}[\ell]} + n_{b_I}^{\mathsf{O}[\ell]} \leq 10 \cdot \left(n_{e_1}^{\varepsilon} + n_{e_2}^{\varepsilon}\right)$$

6.2.3. Functions We now describe the generation of the constraints necessary to link together functions.

Without loss of generality we assume that the first basic block of a function is executed as many times as the function is called. This is not the case if the first basic block of the function is also the first basic block of a loop in that function. But in that case we can simply introduce an empty basic block not belonging to the loop at the beginning of such a function.

We now generate the constraints for the fact that each function's first basic block is executed as many times as the function is called. We distinguish each context according to the VIVU analysis.

Let b_C be an arbitrary basic block from which the first basic block b_φ of a function φ is called. For each context $\vartheta \in \tau(b_C)$ there is at least one context $\vartheta \circ \mathsf{C}[b_C]$ of b_φ. If φ is called recursively, there is also a context $\vartheta \circ \mathsf{R}[b_C]$. For each context ϑ of b_C we generate one constraint:

$$n_{b_C}^{\vartheta} = n_{b_\varphi}^{\vartheta\circ\mathsf{C}[b_C]} \tag{9}$$

For recursive calls we need a recursion bound b which must be supplied by the user. In many cases, b could be found by a data flow analysis. If the compiler optimises by inlining functions we meet similar problems as for loop unrolling. In order to overcome these problems, we had to switch off function inlining.

For each $\vartheta \in \tau(b_C)$ we generate the following constraint which bounds the number of calls to φ:

$$n_{b_\varphi}^{\vartheta\circ\mathsf{C}[b_C]} + n_{b_\varphi}^{\vartheta\circ\mathsf{R}[b_C]} \leq b \cdot n_{b_C}^{\vartheta} \tag{10}$$

6.2.4. Handling the Categorisation ps To handle ps correctly we have to change the objective function and add additional constraints, because a basic block in a context ϑ has two different execution times, one for the first execution that might be a miss, and one for all other references which will be hits.

We define $\hat{n}_b^\vartheta$ to count all but the first execution of b in context ϑ. The relation between n_b^ϑ and $\hat{n}_b^\vartheta$ is:

$$n_b^\vartheta = \hat{n}_b^\vartheta = 0 \quad \vee \quad \left(n_b^\vartheta > 0 \quad \wedge \quad \hat{n}_b^\vartheta = n_b^\vartheta - 1\right). \tag{11}$$

Since this relation contains an $\vee$ operator, the formulation in ILP is not straightforward. However, $\hat{n}_b^\vartheta$ will have a negative coefficient in the objective function and the above constraints will be the only ones for $\hat{n}_b^\vartheta$. Therefore, we can formulate (11) as follows:

$$\hat{n}_b^\vartheta \;\geq\; n_b^\vartheta - 1 \tag{12}$$

Next we introduce $\hat{t}_b^\vartheta$ to be the time that b executes faster in all non-first executions compared to the first execution in context ϑ.

The new times must be incorporated into the objective function. Its definition changes as follows.

Let k_i be the number of distinguished contexts for basic block i and m the number of basic blocks in the program. Then the objective function is

$$fmax = \sum_b \sum_{\vartheta \in \tau(b)} \left(t_b^\vartheta \cdot n_b^\vartheta - \hat{t}_b^\vartheta \cdot n_b^\vartheta\right). \tag{13}$$

6.2.5. Tightening the Bounds The previous sections introduced the basic constraints our technique uses. However, there are some possibilities of further tightening the bounds by additional constraints. The implemented groups of constraints are:

- Whenever lower bounds for loop iterations and recursion depths are known constraints will be introduced for them.
- When it is known that a loop is executed at least once, the infeasible path traversing the loop exit of the first iteration is excluded by an additional constraint. This is easily possible, because we use the VIVU-approach which virtually unrolls the first iteration, thereby introducing an additional edge for the loop exit in the first iteration.
- For mutually recursive functions additional constraints are introduced which are similar to the back edge constraints (6) for loops. These are slightly more complex because recursive function calls might be deeply nested and therefore many different contexts are possible.

6.2.6. Reducing the Problem Size Two simple optimisations concerning the problem size of the ILP have been implemented in our approach.

- Equations of the form $A = B$ are removed and A is substituted by B in the whole ILP.
- In the same way, a pair of constraints $A = X$, $B = X$ is replaced by $B = X$ and A is replaced by B in the whole ILP.

These simple optimisations lead to a quite significant reduction of the size of the generated ILPs. Our tool issues an additional variable → block execution/edge traversal count mapping to enable subsequent analysis steps to evaluate the ILP solution easily.

7. Interpretation of the Relaxed Problem

It is well-known (*see* [24]) that the problem of solving ILPs is $\mathcal{NP}$-complete. Fortunately, the ILPs our approach creates are similar to network flow problems, which usually can be solved fast.

Still, the threat of user provided constraints that destroy the integer nature of the linear problem remains. It is quite unpredictable how arbitrary user constraints interact with the automatically created constraints. In any case, it is safe and only a source of overestimation to solve the relaxed problem, i. e., leaving out the demands for integrality of the execution count variables, as any solution to the ILP is also a solution to the relaxed problem. This way, solving the relaxed problem is suitable whenever there is no need to find execution counts of basic block, but only the worst-case execution time.

During our experiments we found that all of the test problems only needed one step in the branch-and-bound algorithm of lp_solve (written by Michel Berkelaar and freely available at ftp://ftp.es.ele.tue.nl/pub/lp_solve) and could, therefore, be solved as fast as the relaxed problems.

8. Implementation

The cache analysis techniques are implemented with the help of the program analyser generator PAG. The cache analyser gets the executable and a specification of the instruction cache as input and produces a categorisation of the instruction/context pairs in the input program.

The program path analysis consists of about 10 000 lines of C++ code. It includes a C++ and C parser to read the user annotations and an interpreter for a Lisp-like language that provides an easy way of specifying loop and recursion bounds as well as additional constraints.

The frontend of our analyser reads a Sun SPARC executable in `a.out` format. The Sun SPARC is a RISC architecture with a uniform instruction size of four bytes. Our implementation is based on the Executable Editing Library (EEL) of the Wisconsin Architectural Research Tool Set (WARTS) (*see* http://www.cs.wisc.edu/~larus/warts.html).

The profiles used for comparison with our program path analysis were produced with the help of qpt2 (Quick program Profiler and Tracer) that is part of the WARTS distribution.

To solve the integer constraints generated by the program path analyser we used the freely available constraint solver lp_solve.

For our experiments we used parts of the program suites of Frank Müller, some of the programs of Yau-Tsun Steven Li, and some additional programs (*see* Table 2). The programs were compiled using the GNU C compiler version 2.7.2 under SunOS 4.1.4 with either `-O0` or `-O2`.

9. Experimental Results

To test the accuracy of our system we first predicted the hits and misses for each instruction in the test programs. We assumed an instruction cache with a size of 1k byte, a level of associativity of 4, and a line size of 16 bytes.

Table 2. List of the test programs we used. The number of lines of the source code and the number of bytes of the compiled code when compiling with `-O2` are given.

Name	Description	#lines	#bytes
fac	recursive calculation of factorials	16	96
prime	test of several numbers to be prime*	34	228
sort	Bubble sort implementation*	123	236
matmul	5×5 matrix multiplication	46	240
circle	circle drawing routine	200	1240
jfdctint	JPEG forward discrete cos transform	392	1476
stats	two array sums, mean, variance, standard derivation, and linear correlation	186	1512
ndes	data encryption*	287	1944

* with worst-case input

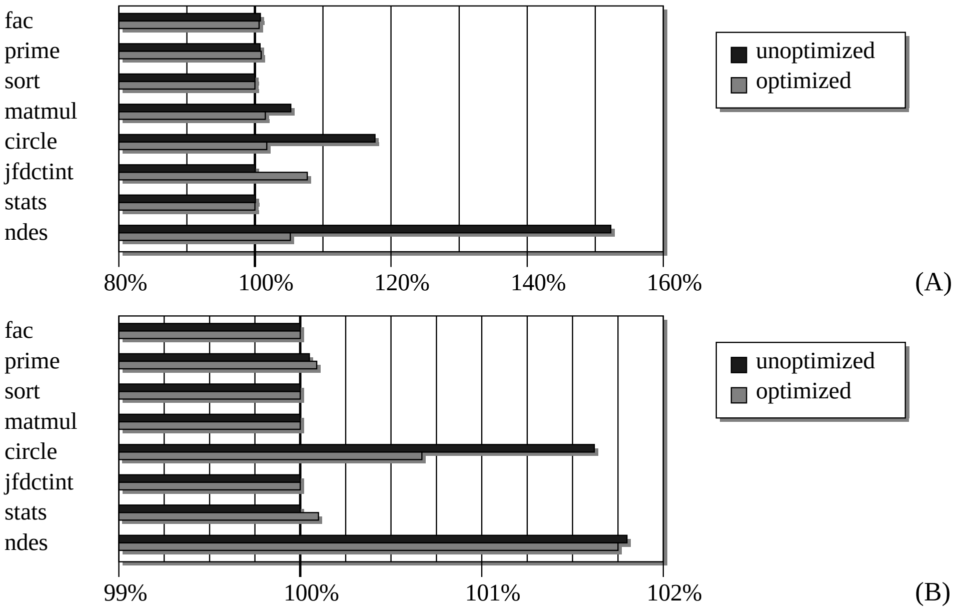


Figure 8. Prediction accuracy of the WCET of several test programs. A) Combined I-cache analysis and program path analysis. B) Program path analysis alone (assuming uniform instruction execution time).

For each instruction in each context the cache analyser predicted whether the instruction execution will always result in a cache hit (ah), or always result in a cache miss (am), whether it will be persistent (ps) or not classified (nc). For the experiments we assumes an idealised virtual hardware that executes all instructions that result in an instruction cache hit in one cycle and all instructions that result in an instruction cache miss in 10 cycles.

For each basic block b in each context $\vartheta \in \tau(b)$ we defined t_b^ϑ to be the sum of the instruction execution times of the idealised hardware and $\hat{t}_b^\vartheta$ the sum of the speed-up in all non-first executions. With this information the program was then first traced with worst-case input (as far as applicable) and then analysed by generating the set of constraints and the objective function described in Section 6. The resulting set of constraints was then solved using the ILP solver.

The results of the objective function which represents the worst-case execution time are compared to the measured cache behaviour (the number of hits and misses) obtained by simulation. Figure 8 presents the overestimation of the analysis. The overestimation is caused by unknown cache behaviour (classifications `nc` and `ps`) on the one hand and by misprediction of worst-case paths in `if`-statements and loops (mainly for the loop condition checks) on the other hand. For technical reasons, calls to library functions are not taken into account by the simulation and are, therefore, also ignored by the objective function in the practical experiments.

Figure 8 shows that the prediction does not always become worse when using compiler optimisation. This is quite surprising as the generated set of constraints is definitely more pessimistic when the compiler optimises the code due to unrolled loops and the like which the analyser is not aware of. However, it seems that because the code shrinks with optimisation, the cache analysis becomes better, so that the more pessimistic set of constraints does not lead to worse analyses. Comparing the execution counts of the trace with our prediction, without taking into account the cache behaviour, our prediction is only slightly worse. For `ndes`, the cache prediction yielded many `nc` classifications due to the bigger executable at compiler optimisation level `-O0`. This is the reason for the overestimation. For other test programs, like `jfdctint`, the cache prediction for the optimised executable was slightly worse than for the non-optimised one. The path analysis alone predicts the worst-case exactly (100%). An interface to the compiler that provides information about the compiler optimisations would help to eliminate this overestimation.

For programs without conditionally executed code apart from the loops like `matmul`, `stats`, etc. the overestimation of our analysis is very small. The path analysis alone predicts the worst case perfectly. Except for loop bounds, no additional user constraints were necessary. For `sort` and `ndes` we had to add additional constraints to reach good results. E. g. the triangular loop structure in `sort` looked like this:

```
a
FOR i:= 1 TO N-1 DO
    FOR j:= N DOWN TO i+1 DO
        b
```

This can be described by the following constraint:

$$n_b = \frac{(\mathtt{N}-1)\cdot\mathtt{N}}{2} n_a \tag{14}$$

The complete analysis, cache prediction plus path analysis, of any of the test programs takes only a few seconds (less than five in all cases) on an AMD K6-2 with 300 MHz. Table 3 shows an overview of the test results with a compiler optimisation level of `-O2`.

10. Related Work

The work of Arnold, Müller, Whalley, and Harmon has been one of the starting points of our work. [19] describes a data flow analysis for the prediction of instruction cache behaviour of programs for direct mapped caches.

Table 3. Results of the program path analysis with optimisation level `-O2`.

	problem size		CPU cycles		accuracy
Name	variables	constraints	predicted	traced	[%]
fac	61	53	25756	25600	100.61
prime	274	238	62507	61940	100.92
sort	110	104	2503726	2503646	100.00
matmul	123	103	1908	1879	101.54
circle	53	49	1945	1912	101.73
jfdctint	49	46	4587	4259	107.70
stats	148	135	150592	150578	100.01
ndes	625	540	54148	51478	105.19

In [11–13] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe describe how integer linear programming can be used to solve the problem of finding worst-case program paths and how this can be extended to cache memories and pipelines (See http://www.ee.princeton.edu/˜yauli/cinderella-3.0/). Their work inspired us to use integer constraints for finding worst-case program paths because of the users being able to formulate nearly everything they know about the program. In our approach we can additionally take advantage of the VIVU analysis. This enables the user to provide constraints separately for different contexts. In the approach of Li, Malik and Wolfe, the cache behaviour prediction with integer constraints leads to quite large sets of constraints when the level of associativity is increased, and leads to prohibitively high analysis times (*see* [13]). Our approach is virtually independent of the level of associativity. Everything between 1 and 8 has been tested without significant changes of analysis times.

By using the results of the cache analysis based on AI (which performs very well for independent of the associativity level) the dramatic increase in complexity can be avoided. The sets of constraints are smaller than those reported in [13] and the estimated WCETs are equally tight. Furthermore, there are no automatically generated long-distance constraints in our approach, which can be introduced by cache conflict constraints.

Widely based on [13], Greger Ottoson and Mikael Sjödin [21] have tried to develop a framework to estimate WCETs for architectures with pipelines, instruction and data caches. Unlike the approach of Li, Malik, and Wolfe, they are not restricted to linear constraints but use logical constraints (formulas) to model microarchitectural properties. Nevertheless, they experience the same problems. In an experiment to predict the cache behaviour of a very small program they report analysis times of several hours.

In [14], Lim et al. describe a general framework for the computation of WCETs of programs in the presence of pipelines and cache memories. Two kinds of pipeline and cache state information are associated with every program construct for which timing equations can be formulated. One describes the pipeline and cache state when the program construct is finished. The other can be combined with the state information from the previous construct to refine the WCET computation for that program construct. The set of timing equations must be explicitly solved. An approximation to the solution for the set of timing equations has been proposed. The usage of an input and output state provides a way for a modularisation for the timing analysis. The method is extended to multi-issue machines in [15]. However, the analyses in this paper only take into account the multi-issue pipeline and the results are, therefore, incomparable to ours.

In [7], Ermedahl and Gustafsson present a method of computing annotations about loops and paths automatically. This technique uses abstract interpretation. It could be used to replace and check some of the user annotations. The work of Sicks (*see* [25]) about determining value ranges of registers does similar computations and will be incorporated into our tool.

In [29], Tice and Graham present a method of retrieving better information about compiler optimisations. The source program is passed through the compiler and optimisations that are interesting for the user in order to understand the behaviour of the executable are represented by source code modifications. Our approach could profit from this idea by letting the user annotate this modified source code instead of the original source code. By this the user immediately sees whether, e. g., the compiler has moved the loop condition test to a different code position and this would enable them to supply our tool with this information. The disadvantage is that the user has to annotate the source code each time the program is compiled.

In [6], Engblom et al. present a method of making the compiler produce an additional description of the optimisations performed. A description language for optimisation transformations is defined which is used as an exchange format between the compiler and the analysis tool. The paper does not present precision results so the method cannot be compared easily to ours. The approach, however, is likely to have the potential of improving our source-target code mapping as well.

11. Conclusion and Future Work

We have shown how it is possible to separate the microarchitecture analysis from the program path analysis by using the results from a microarchitecture analysis by AI for the calculation of worst-case program paths using integer linear programming. This phase ordering performs very well with regard to WCET estimations. Our implementation can be used even for optimising compilers and uses the VIVU analysis to distinguish different execution contexts. By using WCETs for basic blocks as input it is not limited to cache analyses but can use results from any microarchitecture simulation.

We solved the phase ordering problem that occurred by preceding the path analysis with the microarchitecture analysis without noticeable loss of precision. The separation into phases makes it possible to integrate additional analyses into our tool without the threat of inducing unpredictable computational problems by interaction of the analysis steps.

There is still room for further research. We have developed a pipeline analysis that will be integrated into the existing analysers.

Another area of research is the development of an appropriate interface to the compiler. Several ways of retrieving more information about compiler optimisations exist and will be further examined.

Furthermore, we are developing a value analysis to automatically retrieve bounds for loops and recursions, thus minimising the amount of necessary user annotations. In our experiments many loops had a fixed start and end value for the iteration counter. Again an interface to the compiler would help, since it must obviously use such information when rearranging the code.

12. Acknowledgements

We would like to thank Mark D. Hill, James R. Larus, Alvin R. Lebeck, Madhusudhan Talluri, and David A. Wood for making available the Wisconsin architectural research tool set (WARTS), Thomas Ramrath for the implementation of the PAG frontend for SPARC executables, Yau-Tsun Steven Li and Frank Müller for providing their benchmark programs, Martin Alt for an early implementation of the cache analysis, and Joanne Capstick for her proof-reading.

References

1. Aho, A., R. Sethi, and J. Ullman: 1986, *Compilers: Principles, Techniques, and Tools*. Addison Wesley.
2. Alt, M., C. Ferdinand, F. Martin, and R. Wilhelm: 1996, 'Cache Behavior Prediction by Abstract Interpretation'. In: *Proceedings of SAS'96, Static Analysis Symposium*, Vol. 1145 of *Lecture Notes in Computer Science*. pp. 52–66.
3. Alt, M. and F. Martin: 1995, 'Generation of Efficient Interprocedural Analyzers with PAG'. In: *Proceedings of SAS'95, Static Analysis Symposium*, Vol. 983 of *Lecture Notes in Computer Science*. pp. 33–50.
4. Christian Ferdinand, R. W.: 1998, 'On Predicting Data Cache Behavior for Real-Time Systems.'. In: *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*. Montreal, Canada.
5. Cousot, P. and R. Cousot: 1977, '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*. pp. 238–252.
6. Engblom, J., A. Ermedahl, and P. Altenbernd: 1998, 'Facilitating Worst-Case Execution Time Analysis For Optimized Code'. In: *The 10th Euromicro Workshop on Real-Time Systems*. Berlin, Germany.
7. Ermedahl, A. and J. Gustafsson: 1997, 'Deriving Annotations for Tight Calculation of Execution Time'. In: *In Proceedings of Euro-Par '97, LNCS 1300*. Passau, Germany, pp. 1298–1307.
8. Ferdinand, C.: 1997, 'Cache Behavior Prediction for Real-Time Systems'. PhD Thesis, Universität des Saarlandes.
9. Ferdinand, C., F. Martin, and R. Wilhelm: 1997, 'Applying Compiler Techniques to Cache Behavior Prediction'. In: *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*. pp. 37–46.
10. Ferdinand, C., F. Martin, R. Wilhelm, and M. Alt: 1998, 'Cache Behavior Prediction by Abstract Interpretation'. *Science of Computer Programming, Elsevier*.
11. Li, Y.-T. S. and S. Malik: 1995, 'Performance Analysis of Embedded Software Using Implicit Path Enumeration'. In: *Proceedings of the 32nd ACM/IEEE Design Automation Conference*. pp. 456–461.
12. Li, Y.-T. S., S. Malik, and A. Wolfe: 1995, 'Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software'. In: *Proceedings of the 16th IEEE Real-Time Systems Symposium*. pp. 298–307.
13. Li, Y.-T. S., S. Malik, and A. Wolfe: 1996, 'Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches'. In: *Proceedings of the 17th IEEE Real-Time Systems Symposium*.
14. Lim, S.-S., Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C. S. Kim: 1995, 'An Accurate Worst Case Timing Analysis for RISC Processors'. *IEEE Transactions on Software Engineering* **21(7)**, 593–604.
15. Lim, S.-S., J. H. Han, J. Kim, and S. L. Min: 1998, 'A Worst Case Timing Analysis Technique for Multi-Issue Machines'. In: *Proceedings of the 19th IEEE Real-Time Systems Symposium*. Madrid, Spain, pp. 334–345.
16. Martin, F.: 1998, 'PAG—an efficient program analyzer generator'. *International Journal on Software Tools for Technology Transfer* **2**(1).
17. Martin, F.: 1999, 'Generation of Program Analyzers'. Ph.D. thesis, Universität des Saarlandes. to appear.

18. Martin, F., M. Alt, R. Wilhelm, and C. Ferdinand: 1998, 'Analysis of Loops'. In: K. Koskimies (ed.): *Proceedings of the 7th International Conference on Compiler Construction*, Vol. 1383 of *Lecture Notes in Computer Science*.
19. Mueller, F., D. B. Whalley, and M. Harmon: 1994, 'Predicting Instruction Cache Behavior'. In: *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*.
20. Nielson, F., H. R. Nielson, and C. Hankin: 1999, *Principles of Program Analysis.* Springer-Verlag.
21. Ottoson, G. and M. Sjödin: 1997, 'Worst-Case Execution Time Analysis for Modern Hardware Architectures'. In: *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*. pp. 47–55.
22. Schneider, J.: 1998, 'Statische Pipeline-Analyse für Echtzeitsysteme'. Diploma Thesis, Universität des Saarlandes.
23. Schneider, J. and C. Ferdinand: 1999, 'Pipeline Behaviour Prediction for Superscalar Processors by Abstract Interpretation'. In: *In Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*.
24. Schrijver, A.: 1996, *Theory of Linear and Integer Programming*. John Wiley & Sons Ltd..
25. Sicks, M.: 1997, 'Adreßbestimmung zur Vorhersage des Verhaltens von Daten-Caches'. Diploma Thesis, Universität des Saarlandes.
26. Stankovic, J. A.: 1996, 'Real-Time and Embedded Systems'. ACM 50th Anniversary Report on Real-Time Computing Research. http://www-ccs.cs.umass.edu/sdcr/rt.ps.
27. Theiling, H. and C. Ferdinand: 1998, 'Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis'. In: *Proceedings of the 19th IEEE Real-Time Systems Symposium*. Madrid, Spain, pp. 144–153.
28. Thesing, S., F. Martin, and M. Alt: 1997, 'PAG Manual'. Fachbereich 14, Universität des Saarlandes.
29. Tice, C. and S. L. Graham: 1998, 'OPTVIEW: A New Approach for Examining Optimized Code'. In: *Proceedings of the 1998 Workshop on Program Analysis for Software Tools and Engineering, Montreal*.
30. Wilhelm, R. and D. Maurer: 1995, *Compiler Design*, International Computer Science Series. Addison–Wesley. Second Printing.