# Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis

Henrik Theiling and Christian Ferdinand
Universität des Saarlandes / Fachbereich Informatik
Postfach 15 11 50 / D-66041 Saarbrücken / Germany
{theiling,ferdi}@cs.uni-sb.de

## Abstract

*Abstract interpretation (AI) and integer linear programming (ILP) are two techniques which were used independently from each other for worst case execution time (WCET) approximation.*

*With AI one can compute interesting properties of programs. It can be implemented efficiently and yields provably correct results. Previous work has shown that it is suitable for cache behaviour prediction of memory references of a program.*

*By using ILP the structure of a program and the program path can be described easily and in a very natural way. A set of constraints describes the overall structure of the program and solving the constraints yields very precise results. However, when modelling microarchitectural components like caches or pipelines the complexity of the solving process can increase dramatically.*

*Our approach uses AI to model the microarchitecture's behaviour and ILP for finding worst case program paths using the results of the AI. This combines the advantages of both approaches.*

## 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 [13] 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 [1, 6, 4, 5] describes how a cache analysis can efficiently be implemented using AI.

The results of the cache analysis must be combined with the results of a worst case program path analysis. In our approach we use the fine grained results of the cache analysis by AI for the generation of integer constraints to find the worst case program path. In this way, each technique is used to solve the problem it can handle best.

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 AI

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 [14] to enable code improving transformations.

A program analyser takes a program as input and computes some interesting properties. Most of these properties are undecidable. Hence, both correctness and completeness of the computed information are not achievable together. Program analysis makes no compromise on the correctness side; optimising transformations rely on the computed information. It cannot thus guarantee completeness. The quality of the computed information, usually called its *precision*, should be as good as possible.

There is a well developed theory of static program analysis called *abstract interpretation* [3]. With this theory, correctness of a program analysis can easily be derived. According to this theory a program analysis is determined by an *abstract semantics*.

Usually the meaning of a language is given as functions for the statements of the language computing over a concrete domain. A domain is a complete partially ordered set of values. For such a semantics, an abstract version consists of a new simpler abstract domain and simpler abstract functions which define the abstract meaning for every program statement.

The program analyser generator `PAG` [2] offers the possibility of generating a program analyser from a description of the abstract domain and of the abstract semantic functions in two high level languages, one for the domains and the other for the semantic functions. Domains can be constructed inductively starting from simple domains using operators like constructing power sets and function domains. The semantic functions are described in a functional language which combines high expressiveness with efficient implementation. Additionally the user has to supply a join function combining two domain values into one. This function is applied whenever a point in the program has two (or more) possible execution predecessors.

## 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[2].

[2]This holds also 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 [1] and [4].

## 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.

### Control flow representation

We represent programs by control flow graphs consisting of nodes and typed edges. The nodes represent *basic blocks*[3]. For each basic block, the sequence of references to memory is known[4], i. e., there exists a mapping from control flow nodes to sequences of memory blocks: $\mathcal{L} : V \to S^*$.

[3]A basic block is a sequence (of fragments) of instructions in which control flow enters at the beginning and leaves at the end without halt or possibility of branching except at the end. For our cache analysis, it is most convenient to have one memory reference per control flow node. Therefore, our nodes may represent the different fragments of machine instructions that access memory. For non-precisely determined addresses of data references, one can use a set of possibly referenced memory blocks.

[4]This is appropriate for instruction caches and can be too restricted for data caches and combined caches. See [1, 4] for weaker restrictions.

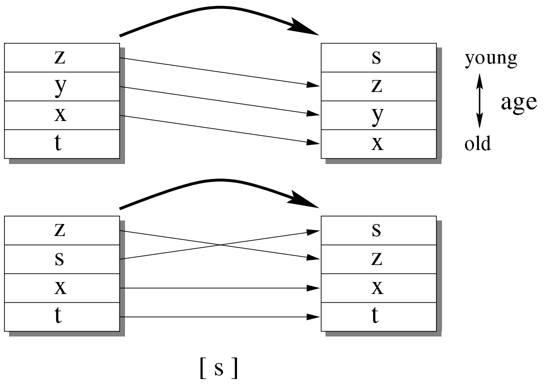



**Figure 1. Update of a concrete fully associative (sub-) cache.**

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: $\mathcal{U}(c_I, \mathcal{L}(k_1) \circ \ldots \circ \mathcal{L}(k_p))$.

### 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.

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[5] predecessors, *join*-functions are used to combine the abstract cache states.

[5] Our join functions are associative. On nodes with more than two predecessors, the join function is used iteratively.

| 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. |
| persistent | pers | 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 pers. |

**Table 1. Categorisations of memory references.**

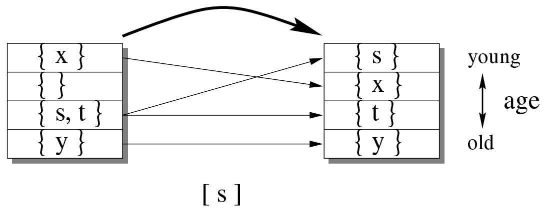



**Figure 2. Update of an abstract fully associative (sub-) cache.**

**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. $\hat{c}(l_x) = \{s_y, \ldots, s_z\}$ means the memory blocks $s_y, \ldots, s_z$ are in the cache. $s_y, \ldots, s_z$ 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 $s_y, \ldots, s_z$, 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 oldest age if it has two different ages (see Figure 3).

The solution of the must analysis computed by the PAG generated analysers by fixpoint iteration 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 $s_x \in \hat{c}(l_y)$ 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).

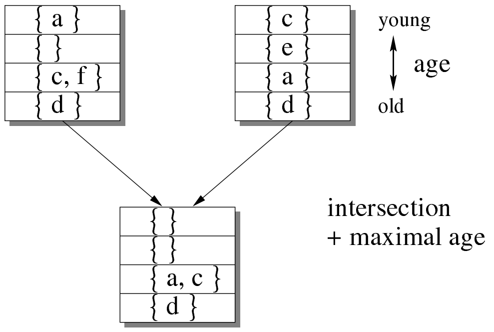



**Figure 3. Join for the must analysis.**

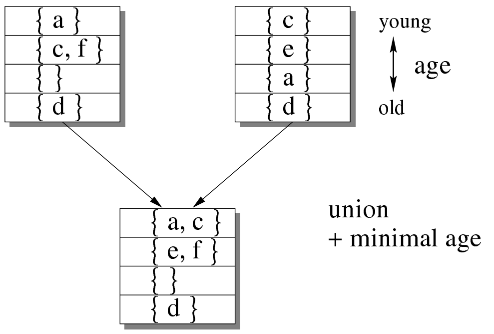



**Figure 4. Join for the may analysis.**

## 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: $\exists l_x, l_y : s_a \in \hat{c}(l_x), s_b \in \hat{c}(l_y), x \supseteq y$.

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 youngest age (see Figure 4).

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).

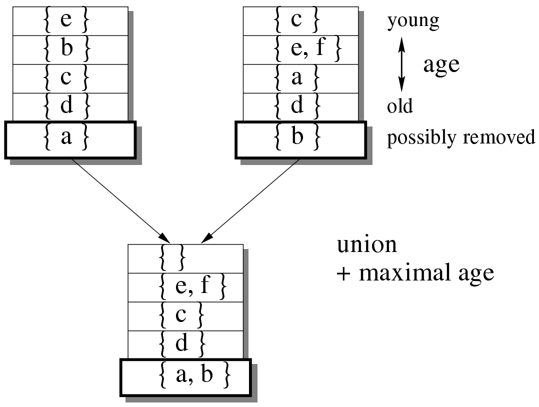



**Figure 5. Join for the persistence analysis.**

## 4.3. Persistence analysis

To improve the prediction we use the classification pers (*persistent*)[6], 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[7]. 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 pers. When a memory block gets 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 youngest age (see Figure 5).

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 pers.

## 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 $\hat{c} : L \to 2^S$ 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.

[6]This improvement is described in [4]

[7]This is similar to the 'first miss' classification as found in [11].

# 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 6).

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

**Figure 6. Loop transformation.**

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.

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 [4].

To get more precise results for the cache behaviour prediction, we have developed the **VIVU approach** which has been implemented with the mapping mechanism of PAG as described in [2]. 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.

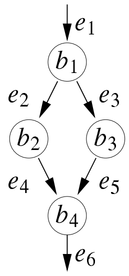
# 6. Program path analysis

A problem formulated in ILP consists of two parts: the cost function, and constraints on the variables used in the cost function. Our cost function represents the number of CPU cycles in the worst case. Correspondingly, it will be maximised. Each variable in the cost 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.

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

$n_{b_2} = \tilde{n}_{e_2}$

$n_{b_2} = \tilde{n}_{e_4}$




$n_{b_1} = \tilde{n}_{e_1}$

$n_{b_1} = \tilde{n}_{e_2} + \tilde{n}_{e_3}$

$n_{b_3} = \tilde{n}_{e_3}$

$n_{b_3} = \tilde{n}_{e_5}$

$n_{b_4} = \tilde{n}_{e_4} + \tilde{n}_{e_5}$

$n_{b_4} = \tilde{n}_{e_6}$

**Figure 7. Constraints that can automatically be generated from the control flow graph. $\mathbf{b_1 \ldots b_4}$ are the basic blocks whose execution counts are $\mathbf{n_{b_1} \ldots n_{b_4}}$. 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 of its outgoing edges.**

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 7 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. Loop and recursion bounds cannot always be inferred automatically and must therefore be provided by the user.

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. In our system arbitrary integer constraints can be added by the user to improve the analysis.

In our program path analysis we use the VIVU approach described in [4, 5, 6] 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 is given in [8]. In the following we will show how to use this together with the VIVU analysis to distinguish execution contexts, so how the cost function can be generated, how to generate the constraints that describe the relation between basic blocks and thereby bounding the cost function, and how to incorporate the cache 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 ($\mathsf{C}[c]$ and $\mathsf{R}[c]$) and first 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 $\ell$ 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 defined as follows:

$$\begin{aligned}\mathcal{T} \;=\; & \{\mathsf{C}[c], \mathsf{R}[c], \mathsf{F}[\ell], \mathsf{O}[\ell] \mid \\ & \quad c \in \ calls\ (\mathcal{P}),\ \ell \in \ loops\ (\mathcal{P})\}^*\end{aligned}$$

In Figure 8 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 $n_b^\vartheta$ be the execution counts for each context $\vartheta \in \tau(b)$ and let $\tilde{n}_e^\vartheta$ be the traversal count for an edge $e$ in a context $\vartheta$.

Generally, each simple constraint for $n_b$, like those in Figure 7, will be generated for each $n_b^\vartheta$. Following this general rule, we obtain the following simple constraints for basic block $b_H$ in Figure 8:

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

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

$$n_{b_S}^\varepsilon = \tilde{n}_{g_1}^\varepsilon$$

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.

## 6.2. Cost function

The cost function computes how many CPU cycles the program will take to execute in the worst case. A basic block can take different times to execute for each of its contexts so we define the cost function as follows.

The cache analysis presented earlier in this paper calculated a categorisation (ah, am, nc or pers) for each basic block in each VIVU context. This categorisation defines a worst case execution time for each basic block in all of its contexts. These execution times are the basis for the cost function of the generated ILP to weigh the basic blocks on the paths in the program.

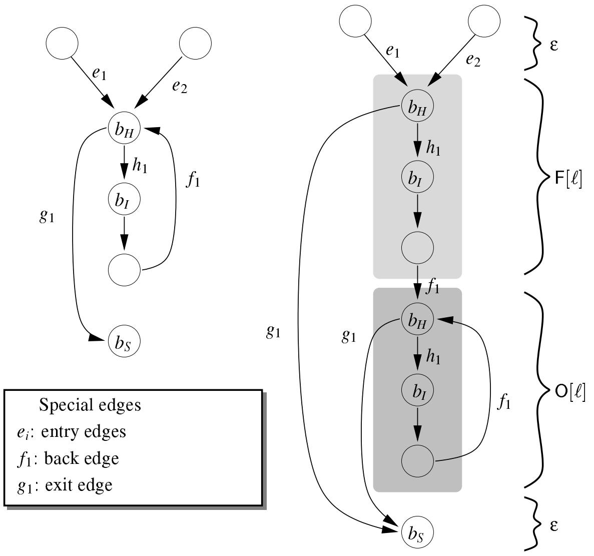



**Figure 8. A simple loop (left). Virtually unrolled once on the right.**

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

Then the cost function is

$$fmax = \sum_b \sum_{\vartheta \in \tau(b)} t_b^\vartheta \cdot n_b^\vartheta. \tag{1}$$

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 $\ell$, called the *head* of the loop.

Let $e_i$ be the edges that enter the loop $\ell$. For each context $\vartheta$ 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:

$$\begin{aligned} &\vartheta \circ \mathsf{F}[\ell] \\ &\vartheta \circ \mathsf{O}[\ell] \end{aligned} \tag{2}$$

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

$$n_{b_H}^{\vartheta \circ \mathsf{F}[\ell]} = \sum_{e_i} \tilde{n}_{e_i}^\vartheta \tag{3}$$

In our example we obtain:

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

Let $f_i$ be the back edges of $\ell$, i. e., those edges that jump back to the beginning of the loop from inside the loop. All non-first iterations of loop $\ell$ 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:

$$n_{b_H}^{\vartheta\circ\mathsf{O}[\ell]} = \sum_{f_i} \left(\tilde{n}_{f_i}^{\vartheta\circ\mathsf{F}[\ell]} + \tilde{n}_{f_i}^{\vartheta\circ\mathsf{O}[\ell]}\right) \tag{4}$$

In the example in Figure 8 there is only one back edge, correspondingly, we generate the following constraint:

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

We also need special constraints for exiting a loop, so we introduce a constraint for each exit edge $g_i$ of loop $\ell$ and for each context $\vartheta$:

$$\tilde{n}_{g_i}^{\vartheta} = \tilde{n}_{g_i}^{\vartheta\circ\mathsf{F}[\ell]} + \tilde{n}_{g_i}^{\vartheta\circ\mathsf{O}[\ell]} \tag{5}$$

The context $\vartheta$ outside our example loop is the empty context, so we obtain:

$$\tilde{n}_{g_1}^{\varepsilon} = \tilde{n}_{g_1}^{\mathsf{F}[\ell]} + \tilde{n}_{g_1}^{\mathsf{O}[\ell]}$$

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

#### 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 $\ell$. Let $k$ be the maximal number of iterations of $\ell$. Then we use the following constraint for each context $\vartheta$ of $e_i$ to bound the number of iterations of $\ell$:

$$n_{b_I}^{\vartheta\circ\mathsf{F}[\ell]} + n_{b_I}^{\vartheta\circ\mathsf{O}[\ell]} \leqq k \cdot \sum_{e_i} \tilde{n}_{e_i}^{\vartheta} \tag{6}$$

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

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

#### 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 $b_\varphi$ of a function $\varphi$ is called. For each context $\vartheta \in \tau(b_C)$ there is at least one context $\vartheta \circ \mathsf{C}[b_C]$ of $b_\varphi$. If $\varphi$ is called recursively, there is also a context $\vartheta \circ \mathsf{R}[b_C]$. For each context $\vartheta$ of $b_C$ we generate one constraint:

$$n_{b_C}^{\vartheta} = n_{b_\varphi}^{\vartheta\circ\mathsf{C}[b_C]} \tag{7}$$

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 $\varphi$:

$$n_{b_\varphi}^{\vartheta\circ\mathsf{C}[b_C]} + n_{b_\varphi}^{\vartheta\circ\mathsf{R}[b_C]} \leqq b \cdot n_{b_C}^{\vartheta} \tag{8}$$

#### 6.2.4. Handling the categorisation pers

To handle pers correctly we have to change the cost function and add additional constraints, because a basic block in a context $\vartheta$ 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 $\vartheta$. The relation between $n_b^\vartheta$ and $\hat{n}_b^\vartheta$ is:

$$n_b^\vartheta = \hat{n}_b^\vartheta = 0 \quad \vee \quad \left(n_b^\vartheta > 0 \quad \wedge \quad \hat{n}_b^\vartheta = n_b^\vartheta - 1\right). \qquad (9)$$

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 cost function and the above constraints will be the only ones for $\hat{n}_b^\vartheta$. Therefore, we can formulate (9) as follows:

$$\hat{n}_b^\vartheta \quad \geqq \quad n_b^\vartheta - 1 \qquad (10)$$

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 $\vartheta$.

The new times must be incorporated into the cost 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 cost function is

$$fmax = \sum_b \sum_{\vartheta \in \tau(b)} \left(t_b^\vartheta \cdot n_b^\vartheta - \hat{t}_b^\vartheta \cdot n_b^\vartheta\right). \qquad (11)$$

## 7. 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 9 500 lines of C++ code. It includes a 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)[8].

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 `lp_solve`[9].

[8]The WARTS home page containing EEL and qpt2 is reachable via `http://www.cs.wisc.edu/~larus/warts.html`

[9]lp_solve was written by Michel Berkelaar and is freely available at `ftp://ftp.es.ele.tue.nl/pub/lp_solve`

| 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 | $5 \times 5$ 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

**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.**

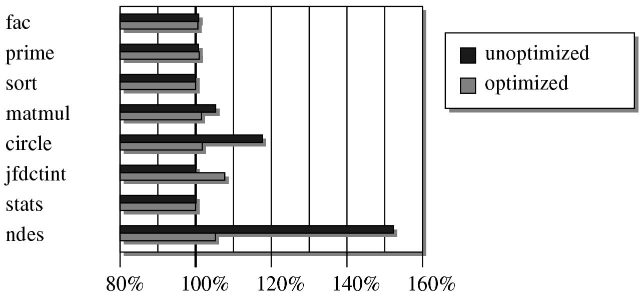



**Figure 9. Prediction accuracy of the WCET of several test programs. The WCET is normalised to 100%. The programs were tested with `gcc` using both no optimisation and `-O2`.**

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`.

## 8. 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.

For each instruction in each context the cache analyser predicted whether the instruction was always going to be in the cache (ah), or never be in the cache (am), whether it will be persistent (pers) or not classified (nc). To make this information usable for the program path analysis we assume 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. Accordingly, for instructions classified as pers the first execution was assumed to take 10 cycles, all further ones only one.

For each basic block $b$ in each context $\vartheta \in \tau(b)$ we defined $t_b^\vartheta$ 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 cost function described in section 6. The resulting set of constraints was then solved using the ILP solver.

The results of the cost 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 9 presents the overestimation of the analysis. The overestimation is caused by unknown cache behaviour (classifications nc and pers) 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 therefore also ignored by the cost function in the practical experiments.

Figure 9 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 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 more pessimistic program path analysis lead to an overestimation for executables compiled with optimisation. 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 small. 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:

$$n_b \quad = \quad \frac{(\mathtt{N}-1)\cdot\mathtt{N}}{2}\, n_a \qquad (12)$$

The analysis of any of the test programs takes only a few seconds on a SUN UltraSPARC with 168 MHz, although `ndes` compiled without optimisation takes around 80s to be solved (with 705 constraints). Table 3 shows an overview of the test results with a compiler optimisation level of `-O2`.

| Name | problem size | | CPU cycles | | accuracy |
|---|---|---|---|---|---|
| | 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 |

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

## 9. Related work

The work of Arnold, Müller, Whalley, and Harmon has been one of the starting points of our work. [11] describes a data flow analysis for the prediction of instruction cache behaviour of programs for direct mapped caches.

In [7, 8, 9] 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[10]. 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. This was the reason for us to use abstract interpretation for this kind of problem which was shown (in [4]) to compute the results quickly for any level of associativity. The sets of constraints are smaller than those reported in [9] and the estimated WCETs are equally tight. The threat of user provided constraints that destroy the structure of the ILP leading to very high solving times still remains although it did not occur with our test programmes.

Widely based on [9], Greger Ottoson and Mikael Sjödin [12] 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 experi-

[10]See `http://www.ee.princeton.edu/~yauli/cinderella-3.0/`

ment to predict the cache behaviour of a very small program they report analysis times of several hours.

In [10], 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. Unlike our method that is based on well explored theories and tools for AI, 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.

## 10. Conclusion and future work

We have shown how it is possible to use the results from a cache analysis by AI (presented in [1, 4, 5, 6]) for the calculation of worst case program paths using integer linear programming. 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.

There is still room for further research. We are currently developing 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. The compiler should produce a description of the applied optimisations, which can be used in our analyser.

Furthermore, we are developing a data flow analysis to be able to predict bounds for simple loops and recursions. 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.

## 11. 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] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *Proceedings of SAS'96, Static Analysis Symposium*, LNCS 1145, pages 52–66. Springer, Sept. 1996.

[2] M. Alt and F. Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *Proceedings of SAS'95, Static Analysis Symposium*, LNCS 983, pages 33–50. Springer, Sept. 1995.

[3] P. Cousot and R. 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*, pages 238–252, Jan. 1977.

[4] C. Ferdinand. Cache Behavior Prediction for Real-Time Systems. Dissertation, Universität des Saarlandes, Sept. 1997.

[5] C. Ferdinand, F. Martin, and R. 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*, pages 37–46, June 1997.

[6] C. Ferdinand, F. Martin, and R. Wilhelm. Cache Behavior Prediction by Abstract Interpretation. *Science of Computer Programming*, 1998. Selected for SAS'96 special issue.

[7] Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.

[8] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 298–307, Dec. 1995.

[9] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 1996.

[10] S.-S. Lim, 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. An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[11] F. Mueller, D. B. Whalley, and M. Harmon. Predicting Instruction Cache Behavior. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1994.

[12] G. Ottoson and M. Sjödin. 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*, pages 47–55, June 1997.

[13] J. A. Stankovic. *Real-Time and Embedded Systems*. ACM 50th Anniversary Report on Real-Time Computing Research, 1996. http://www-ccs.cs.umass.edu/sdcr/rt.ps.

[14] R. Wilhelm and D. Maurer. *Compiler Design*. International Computer Science Series. Addison–Wesley, 1995. Second Printing.