Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation

Jörn Schneider*

Universität des Saarlandes
Fachbereich Informatik
Postfach 15 11 50
D-66041 Saarbrücken, Germany
Phone: +49 681 302 3434
Fax: +49 681 302 3065
js@cs.uni-sb.de
http://www.cs.uni-sb.de/~js

Christian Ferdinand

AbsInt Angewandte Informatik GmbH
Universität des Saarlandes
Starterzentrum – Gebäude 45
D-66123 Saarbrücken, Germany
Phone: +49 681 8318317
Fax: +49 681 8318320
ferdinand@AbsInt.de
http://www.AbsInt.de

Abstract

For real time systems not only the logical function is important but also the timing behavior, e. g. hard real time systems must react inside their deadlines. To guarantee this it is necessary to know upper bounds for the worst case execution times (WCETs). The accuracy of the prediction of WCETs depends strongly on the ability to model the features of the target processor.

Cache memories, pipelines and parallel functional units are architectural components which are responsible for the speed gain of modern processors. It is not trivial to determine their influence when predicting the worst case execution time of programs.

This paper describes a method to predict the behavior of pipelined superscalar processors and reports initial results of a prototypical implementation for the SuperSPARC I processor.

1 Introduction

The correctness of a hard real-time system depends not only on the logical functionality, the programs must also satisfy the temporal requirements dictated by the physical environment. To analyze the schedulability of a given set of tasks, upper bounds for the worst case execution time (WCET) of the tasks are required. In general it is not possible to determine these upper bounds by measurements or simulation, because all possible combinations of input values must be considered. We use static program analysis to compute the WCET of programs. Our method is based on abstract interpretation.

To obtain sharp bounds for the WCET it is necessary to consider the features of the target processors. Modern processors gain speed through: The use of cache memories and pipelines, and the parallel execution of instructions.

In this paper the prediction of the behavior of pipelines combined with parallel execution on a superscalar processor is presented. Initial results of a prototypical implementation for the SuperSPARC I processor [18] are presented.

*Partly supported by DFG (German Research Foundation), Transferbereich 14

2 Superscalar Pipelines

2.1 Principle of Pipelines

The idea of pipelining is to overlap the execution of instructions. This is accomplished by dividing the execution of instructions into several steps (pipeline stages). The ideal case of full overlapped instructions can not always be reached. Pipeline bubbles can occur in real pipelines, when the pipeline stalls. Situations, which can lead to pipeline stalls are called *pipeline hazards.* There are three types of hazards. Structural hazards are caused by resource conflicts, data hazards are caused by data dependences, and control hazards are caused by control flow changes (e.g. branches).

2.2 Superscalar Execution

A measure for the throughput of a processor is the CPI value (Cycles Per Instruction).

With the concept of pipelining it is possible to reach at best a CPI value of 1. To increase the throughput,

i. e. decrease the CPI value further it is necessary to introduce parallel execution. The parallelization can be done statically or dynamically. The static approach is used in e. g. VLIW (Very Large Instruction Word) processors [8], where the compiler is responsible for selecting the instructions to execute simultaneously. In superscalar processors the dynamic approach is used, i. e. the processor selects the instructions to be processed concurrently during run time.

Stage	Tasks
F0	• Fetch of up to 4 instr. in case of cache hit, or up to 8 instr. from memory
F1	• Filling of the prefetch queue
D0	• Selection of instructions from prefetch queue to form a group of concurrent executable instructions
D1	• Assignment of functional units and register ports to instructions • Computation of branch destination addr. • Reading of address registers for memory accesses
D2	• Reading of operand registers • Computation of virtual addresses for memory references
E0	• Execution of computations in Arithmetic Logic Unit 1 (ALU1) • Start of data cache accesses • Transfer of floating point instructions to the FPU
E1	• Computations, that depend on instruction in same group are executed in the cascaded ALU2 • Completion of data cache accesses
WB	• Write back of integer results

Table 1: Stages of the integer pipeline.

2.3 Example: SuperSPARC I

The SuperSPARC I is a superscalar RISC processor, compatible with the SPARC version 8 architecture [19]. It is capable of grouping up to three instructions to execute them concurrently. The grouping of instructions is a dynamic process. The decision which instructions are grouped together depends on the availability of instructions in the prefetch queue or the cache, and on data dependences, resource conflicts, and control flow changes. The data dependences and the resource conflicts can occur between the available instructions and between these and already grouped instructions.

Stage	Tasks
FD	Decode of Floating point operations
FRD	Read of FP registers
FE	Execution of FP instructions
FL	Rounding and normalization of results
FWB	Write back of floating point results

Table 2: Stages of the floating point pipeline.

The SuperSPARC I has a main or integer pipeline and a floating point pipeline. The pipeline stages of the main and floating point pipeline are displayed in Table 1 and Table 2.

The processing of a floating point instruction starts in the first part of the main pipeline and continues in the floating point pipeline. The transfer from the integer pipeline to the floating point pipeline is done in E0 (FD).

The execution time of some floating point instructions depends on their operands. If the operands or the result are subnormal values the computation takes longer (see [8]).

3 Pipeline Analysis by Abstract Interpretation

Abstract interpretation is a well developed theory of static program analysis [3]. It is semantics based, thus supporting correctness proofs of program analyses. 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 behavior of a program (including its pipeline behavior) is given by the (formal) semantics of the program. To predict the pipeline behavior of a program, its "collecting semantics[1]" approximated. The collecting semantics gives the set of all program (pipeline) 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 behavior of

[1] In [3], the term "static semantics" is used for this.

programs [4, 6, 20].

The approach works as follows: in a first step, the "concrete pipeline semantics" of programs is defined. The concrete pipeline semantics is a simplified semantics that describes only the interesting aspects of the pipeline behavior but ignores other details of execution, e. g. register values, results of computations, etc. In this way each real pipeline state is represented by a concrete pipeline state. In the next step, we define the "abstract pipeline semantics" that "collects" all occurring concrete pipeline states for each program point.

From the construction of the abstract semantics follows that all concrete pipeline states that are included in the collecting semantics for a given program point are also included in the abstract semantics [4]. An abstract pipeline state at a program point may contain concrete pipeline states that cannot occur at this point, due to infeasible paths. This can reduce the precision of the analysis but doesn't affect the correctness (see [4]).

The computation of the abstract semantics has been implemented with the help of the program analyzer generator PAG [14], which allows to generate a program analyzer from a description of the abstract domain (here, sets of concrete pipeline states) and of the abstract semantic functions.

4 Pipeline Semantics

Before the pipeline semantics is presented, which allows to analyze the behavior of superscalar pipelines, we try to motivate its design. The pipeline semantics must at least allow us to detect stalls and to model the selection of instructions for concurrent execution. Therefore, the detection of hazards must be possible and the information which instructions are available must be provided.

To detect structural hazards the resource usage of instructions has to be known. For most modern processors, memory access is too slow to cause data hazards. Dependences over cache accesses can be treated within a data cache/store buffer analysis [6]. It is assumed that data hazards occur only in case of dependences between data registers. They can be detected by modeling read and write ports of registers as resources. Control hazards can also be detected with information about resource usage. An instruction that changes the control flow must write to a special resource, e. g. the *next program counter register.*

To know which instructions are available the cache behavior and the state of the prefetch queue should be known. The cache behavior is predicted by a separate cache analysis [5]. To model the prefetch queue the pipeline semantics must allow the description of resources with their own state.

Our approach is based on the pipeline analysis framework in [4]. We consider an in-order superscalar processor that can execute a *group* of up to N instructions concurrently.

For superscalar processors it is not sufficient to build a kind of static reservation table, since the assignment of resources to pipeline stages of instructions can change dynamically during the grouping process. Additionally the state of some resources must be provided.

4.1 Concrete Pipeline Semantics

Definition 4.1 (resource association)
Let $R = \{r_1, \ldots, r_m\}$ be the set of resource types and resources of the processor. Let PS be the set of pipeline stages. A pair $(s, \{r_{j_1}, \ldots, r_{j_n}\})$ with $s \in PS$ and $r_{j_1}, \ldots, r_{j_n} \in R$ is a *resource association.* $\mathcal{R} = (PS \times 2^R)$ denotes the set of all resource associations. □

Definition 4.2 (resource association sequence)
A sequence $\bar{r} \in \bar{R} = \mathcal{R}^*$ is a *resource association sequence.* Let "." be the concatenation operator for resource association sequences. □

Definition 4.3
A *resource demand sequence* is a resource association sequence describing the statically given resource demand of an instruction (type).
A *resource allocation sequence* is a resource association sequence describing an actual assignment of resources to an instruction. It depends on the current state of the pipeline. □

Resource allocation sequences always start with the resource allocation for the current pipeline stage, i. e. resource allocations of previous pipeline stages are removed, when the instruction advances through the pipeline.

Example 4.1
Consider an instruction with the following resource demand sequence

$$(s_1, \{r_{x_1}, \cdots, r_{x_k}\}).(s_2, \{r_{y_1}, \ldots, r_{y_l}\}).$$
$$(s_3, \{\}).(s_4, \{r_{z_1}, \ldots, r_{z_m}\}).(s_4, \{r_{z_1}, \ldots, r_{z_m}\}) \cdots$$

This instruction needs the resource types or resources $\{r_{x_1}, \cdots, r_{x_k}\}$ in pipeline stage s_1, before $\{r_{y_1}, \ldots, r_{y_l}\}$ are needed in stage s_2. The instruction requires no resources in stage s_3. It stays two cycles in s_4 and needs $\{r_{z_1}, \ldots, r_{z_m}\}$ both times before it continues through the remaining stages. □

The resource demand sequence of an instruction depends only on the instruction type (e. g. ADD or DIV) and the operand types (e. g. register or immediate value).

How the resource demand of an instruction can be satisfied depends on the actual situation in the pipeline.

The initial resource allocation sequence of an instruction can differ from the resource demand sequence of the type of this instruction. Where multiple resources of a resource type are available a particular instance is chosen.

A concrete pipeline state describes the occupancy of the pipeline stages by instructions, the current and future resource allocations for these instructions and the state of some special resources, e. g. the prefetch queue.

Definition 4.4 (concrete pipeline state)
A *concrete pipeline state* p consists of the resource allocation sequences of the up to $N * |PS|$ (up to N instructions per pipeline stage) instructions, which are currently in the pipeline, $R_{\#} = ((\bar{R})^N)^{|PS|}$, and the state s_R of some resources, i. e. $p = (r_{\#}, s_R)$. P denotes the set of all possible concrete pipeline states[1]. □

The concrete pipeline state changes when a new instruction enters the pipeline. The resulting new pipeline state depends on the previous pipeline state, on the (*resource demand sequence* of the) new instruction and on the states of other processor parts (e. g. the state of the cache memory).

Definition 4.5 (update function)
Let IS be the instruction set of the processor. The (concrete) *update function* $\mathcal{U} : P \times IS \to P$ models the effect on the concrete pipeline state caused by the entrance of a new instruction into the pipeline. □

Definition 4.6 (cycles function)
The *cycles function* $\mathcal{C} : P \times IS \to \mathbb{N}_0$ computes the number of cycles needed by a new instruction to enter the pipeline, i. e. the number of cycles needed to reach the pipeline state $\mathcal{U}(p, i)$. □

Definition 4.7 (empty function)
The *pipeline empty function* $\mathcal{E} : P \to \mathbb{N}_0$ computes the number of cycles which are needed to flush the pipeline, i. e. the numbers of cycles needed to reach the empty pipeline state P_ε. □

4.2 Control Flow Representation

Programs are represented by control flow graphs consisting of nodes and typed edges. The nodes represent instructions. Each instruction is statically assigned a resource demand sequence, i. e. there exists a mapping from control flow nodes to resource demand sequences: $res_v : V \to \bar{R}$.

[1]This set is finite. The length of the resource allocation sequences is limited, because the number of pipeline stages is finite and the number of repetitions of pipeline stages is also limited. The sets of possible states of resources are finite.

The update function $\mathcal{U}$ is extended to sequences of instructions:

$$\mathcal{U}(p, \langle i_1, \ldots, i_k \rangle) = \mathcal{U}(\ldots \mathcal{U}(\mathcal{U}(p, i_1), i_2), \ldots, i_k)$$

The pipeline behavior of a path $(i_1, \ldots, i_l)$ in the control flow graph is given by applying $\mathcal{U}$ to the empty pipeline state p_ε and the concatenation of all instructions paired with the appropriate processor state information along the path:

$$\mathcal{U}(p_\varepsilon, (i_1, \ldots, i_l))$$

4.3 Abstract Semantics

There are only finitely many concrete pipeline states and their representation is usually small. Therefore, *sets* of concrete pipeline states[2] can be used as the domain for our abstract interpretation and do not need space efficient descriptions of sets of concrete pipeline states.

Definition 4.8 (abstract pipeline state)
An *abstract pipeline state* $\hat{p} \subseteq P$ is a set of concrete pipeline states. $\hat{P} = 2^P$ denotes the set of all abstract pipeline states. □

The abstract version of the concrete pipeline update function is a canonical extension of the concrete pipeline update function to sets: $\hat{\mathcal{U}}(\hat{p}, i) = \{\mathcal{U}(p, i) \mid p \in \hat{p}\}$

Definition 4.9 (pipeline join function)
A join function combines two abstract pipeline states. The join function is given by the least upper bound of the abstract domain. The *pipeline join function* $\hat{\mathcal{J}} : \hat{P} \times \hat{P} \to \hat{P}$ is set union: $\hat{\mathcal{J}}(\hat{p}_1, \hat{p}_2) = \hat{p}_1 \cup \hat{p}_2$ □

The join function is used to union the abstract pipeline states of two or more merging paths. To combine more than two values the join function is extended:

$$\hat{\mathcal{J}}(\hat{p}_1, \ldots, \hat{p}_n) = \hat{\mathcal{J}}(\hat{p}_1, \hat{\mathcal{J}}(\hat{p}_2, \cdots, \hat{\mathcal{J}}(\hat{p}_{n-1}, \hat{p}_n) \cdots))$$

4.4 Pipeline Analysis

In order to solve the pipeline analysis for a program, one can construct a system of recursive equations from its control flow graph. In the program analyzer generator PAG this is only done implicitly.

The variables in the equation system stand for abstract pipeline states for program points. For every control flow node k, representing instruction i_k there is an equation $\hat{p}_k = \hat{\mathcal{U}}(pred(k), i_k)$. If k has only one direct predecessor k', then $pred(k) = \hat{p}_{k'}$. If k has more than

[2]The domains of the abstract semantics and the collecting semantics are equal. But an abstract pipeline state may contain concrete pipeline states, that do not occur in the respective collecting semantics, due to infeasible paths.

one direct predecessors $k_1, ..., k_x$, then there is an equation $\hat{p}_{k'} = \hat{\mathcal{J}}(\hat{p}_{k_1}, \ldots, \hat{p}_{k_x})$, and $pred(k) = \hat{p}_{k'}$. PAG derives the solution of the abstract interpretation by fixpoint iteration. The iteration starts with the empty abstract pipeline state.

The solution of the abstract interpretation is understood in the following way: An abstract pipeline state $\hat{p}$ at a control flow node k reflects all concrete pipeline states that may occur whenever control reaches k.

For each concrete pipeline state in an abstract pipeline state the pipeline cycles function or the pipeline empty function is applied to determine the elapsed clock cycles. Let $\hat{p}_1, \ldots, \hat{p}_n$ be the abstract pipeline states at the immediate predecessors of k and let

$$pred = \hat{\mathcal{J}}(\hat{p}_1, \cdots, \hat{p}_n)$$

The maximal number of clock cycles needed for the instruction i at k to enter the pipeline is determined with the help of the *cycles function* as defined in Section 4.1:

$$max\{\mathcal{C}(p_x, i) \mid p_x \in pred\}$$

The maximal number of clock cycles needed to finish all pending instructions of the abstract pipeline state $\hat{p}$ is determined with the help of the *empty function* as defined in Section 4.1

$$max\{\mathcal{E}(p) \mid p \in \hat{p}\}$$

4.5 Example: SuperSPARC I

In the following an example of an update function for the SuperSPARC I is given. The update function is provided with the pipeline state p and the new instruction ni. Since the results of a preceding cache analysis are incorporated the cache state isn't included in s_R, but use an explicit parameter $cr \in CR = \{hit, miss\}$. The update function needs two more informations: the address of the new instruction which is needed to model the prefetch queue behavior, and an indicator that distinguishes normal and *hard instructions* (hard instructions can't be executed with others in one group). The two possible values of the indicator are *open* (the group of concurrently executed instructions can still be enhanced) and *closed* (the group can't be enhanced). The instruction address depends on the particular instruction, while the indicator depends on the instruction type. They are accessed via the functions *get_address*() and *get_ind*() respectively. The prefetch queue is modeled as a resource with its own state. The following notation is used: $p.s_R$ denotes the prefetch queue state and $p.R_\#[s]$ denotes the resource allocation sequences of the instructions in stage s, nPC^w denotes the write port of the next program counter register.

```
U(p, ni, cr):

newgrp_stat := get_ind(ni); // closed if ni is a
                            // hard instruction open otherwise

addr := get_address(ni);
if head(p.s_R) ≠ addr AND cr = miss then
    // ni neither at top of prefetch queue, nor in cache
    // ⇒ fetch from memory (let the
    // cache_miss_penalty elapse)
    for j := 1 to cache_miss_penalty do
        // Either insert a pipeline bubble
        // or in case of a stall elapse one cycle
        p := delay_one_cycle(p);
    od
    // Enqueue up to 8 instructions
    p.s_R := enq(p.s_R, fetch_from_memory(addr));
fi
if cr = hit AND head(p.s_R) ≠ addr
    // ni in cache but not yet in prefetch queue
    // ⇒ fetch from cache
    // Enqueue up to 4 instructions (cache hits only)
    p.s_R := enq(p.s_R, fetch_from_cache(addr));
fi
// Allocate resources for new instruction
new_res_alloc := allocate_res(res_demand(ni), p);
last_res_alloc := p.R_#[F0];
if res_confl(new_res_alloc,last_res_alloc) OR // (A)
    data_dep(new_res_alloc,last_res_alloc) // (B)
    OR newgrp_stat = closed
    OR grp_stat = closed then
        // ni can't join an existing group
        newgrp := create_group(ni);
        newgrp_created := True;
        if nPC^w in new_res_alloc then// (C)
            // ni changes the control flow
            newgrp_stat := closed;
        fi
fi
if newgrp_created then
    // Free F0 for new group
    p := advance_pipeline(p);
fi
while structural_hazard(new_res_alloc,p) OR
    data_hazard(new_res_alloc,p) OR // (D)
    control_hazard(new_res_alloc,p) do
        p := delay_one_cycle(p) ;
od
if newgrp_created then
    // Let new group enter the pipeline
    p.R_#[F0] := newgrp;
else
    // Let ni join existing group
    p := join_group(p, ni);
    if grp_full(p.R_#[F0]) then
```

```
        newgrp_stat := closed;
    fi
fi
grp_stat := newgrp_stat;
return(p);
```

The grouping process is modeled by a set of rules. These rules are based upon the resource demand sequences of available instructions and the resource allocation sequences of instructions which are already further in the pipeline.

Example 4.2
One of the rules that are applied in *res_confl*() (position **(A)**) in the update function example says that two instructions that access the data cache cannot be grouped together.

A rule applied in *data_dep*() (position **(B)**) says that instructions which access a read port of a data register in stage D1 (i. e. load or store instructions) cannot be grouped with a preceding instruction that uses the write port of this particular data register.

The rule applied at position **(C)** says that an instruction which uses the write port of the next program counter register (e. g. branches) is always the last in a group.

These rules prevent problems arising from resource conflicts, data dependences, and control flow changes respectively. □

While the application of all these rules can be triggered by the resource allocations of instructions, it is not always necessary to exhaustively search the resource allocation or resource demand sequences for a triggering resource. From the resource demand sequences of the instruction types some necessary preconditions can be precomputed.

Example 4.3
Consider the first rule of Example 4.2. Only the resource allocation sequence of the various types of load and store instructions contain the data cache. Therefore, this rule can be triggered just by looking at the instruction type. □

The stall behavior of the SuperSPARC I is also modeled by rules.

Example 4.4
One of the rules applied in *data_hazard*() (position **(D)**) in the update function example says that a pipeline bubble has to be inserted between a group wherein the write port of a data register R_x and the ALU in stage E1 is used and a group which uses the read port of R_x in stage D1. □

In some special cases the documentation of the SuperSPARC I was insufficient for our purposes. Therefore pessimism is introduced in the analyzer. This pessimism results in pessimistic concrete pipeline states. A pessimistic concrete pipeline state contains more resources than the instruction actually uses or allocates resources for more pipeline stages than are actually occupied by the instruction.

Example 4.5
Consider the following SuperSPARC I instruction sequence:

```
A  8000:  ADD R1,R2,R3
B  8004:  ADD R3,R5,R4
C  8008:  LD [R4+4],R3
D  8012:  SUB R6,R3,R7
```

The resource demand sequences for these instructions are shown in Table 3. R_x^r, R_x^w are the read and write ports of data register x. DC stands for the data cache and ALU for the resource type arithmetic logic unit.

	D1	D2	E0	E1	WB
A	-	R_1^r,R_2^r	R_1^r,R_2^r,ALU	-	R_3^w
B	-	R_3^r,R_5^r	R_3^r,R_5^r,ALU	-	R_4^w
C	R_4^r	R_4^r	DC	DC	R_3^w
D	-	R_6^r,R_3^r	R_6^r,R_3^r,ALU	-	R_7^w

Table 3: Resource demand sequences of Example 4.5.

Figure 1 shows the dynamic change of the resource allocation sequences and the prefetch queue state for this example. The prefetch queue state is modeled by a sequence of instruction addresses (shown in the box under the appropriate call to the update function). The first address is on top of the queue.

In pipeline state p_1 a new group is created, which contains just A. The prefetch queue is filled from the cache and A is dequeued. In p_2 B joins the group of A. A dynamic change of the resource allocation against the statically assigned resource demand for instruction B occurs. Since the result of B depends on the result of A, B must use the cascaded ALU (ALU_2) and not ALU_1. B is dequeued from the prefetch queue. In p_3 a pipeline bubble is inserted and a new group is started for C. The bubble is necessary, since the LD instruction depends on a result of the cacaded ALU in E1, which otherwise could not be forwarded. A and B advance four pipeline stages (two cycles). C is dequeued from the prefetch queue. In p_4 D starts a new group since it depends on C which forwards its result from E1. D is dequeued from the prefetch queue, which is empty then. □

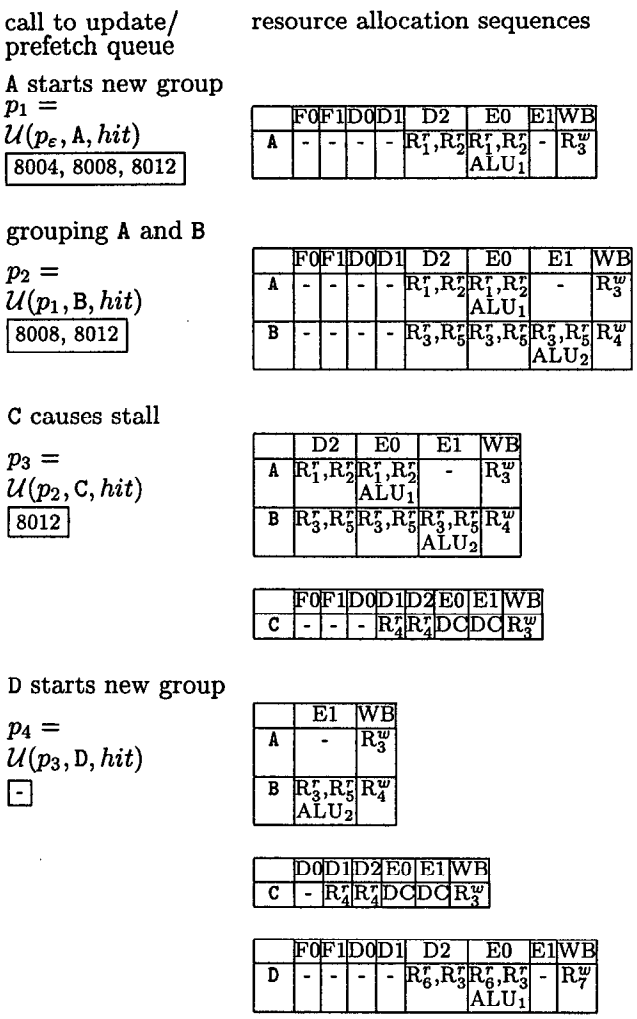

Figure 1: Application of the concrete update function on Example 4.5

5 Out of Order Execution

Processors with out of order execution are more flexible in choosing instructions for parallel execution. If the foremost instruction in row doesn't fit they will try the next one. Of course out of order pipelines too, have to check for hazards.

As stated above our semantics is suitable for superscalar pipelines with in order execution. As a matter of fact the above semantics is also suitable for pipelines with out of order execution. The decision of an out of order execution machine to choose a subset from the available instructions is based on the resource requirements of the instructions. Since our semantics is based on resources, no changes are needed to support out of order execution.

Nevertheless the concrete semantic functions, and the cycles and empty function must be adapted for a new target processor.

6 Practical Experiments

In this section the results of a first implementation of the pipeline analysis are presented. We have chosen the SuperSPARC I processor as target. The implementation has been done with the program analyzer generator PAG, i.e. the pipeline analyzer is generated from a description of the semantic functions. For the sake of space, this description isn't shown here (for further information see [18]).

The analyzer takes as input the control flow graph of a program and the results of the cache analysis [5]. We conservatively assume that memory references that are not classified as *hits* by the cache analysis are cache misses[1]. The output of the analyzer is a mapping *map* of instruction/context pairs to pairs of clock cycles. The first element of a clock cycle pair is the result of the *cycles function* applied to the abstract pipeline state as shown in Section 4.4. The second element is either the result of the *empty function* applied to the abstract pipeline state for *exit instructions*[2] or zero for all other instructions.

A context represents the execution stack, i.e. the trace of function calls and loops along the corresponding path in the control flow graph to the instruction. Let IC be the set of all instruction/context pairs.

$$map \;:\; IC \to \mathbb{N}_0 \times \mathbb{N}_0$$

In general it is impractical to regard all possible contexts (paths) of a program. The cache behavior (and thereby the pipeline behavior) often exhibits significant differences caused by initialization effects. Therefore, first iterations of each loop from further iterations and first calls to each (recursive) function from recursive calls are distinguished. Instead of distinguishing all paths, path classes are considered for which similar behavior is expected.

This is realized by a generalization of well known interprocedural analysis schemes [15]. The approach is called **VIVU** (Virtual Inlining of non-recursive functions and Virtual Unrolling of loops and recursive functions).

The frontend of the analyzer reads a Sun SPARC executable in a.out format. The implementation is based on the EEL library of the Wisconsin Architectural Research Tool Set (WARTS).

The worst case execution profile of a program determines how often each instruction/context pair is maximally encountered during the execution of a program. By combination with the results of our pipeline analysis the worst case number of clock cycles needed to execute the input program can be estimated. For our

[1] For the SuperSPARC I this assumption is safe, since cache misses don't lead to accelerations.

[2] An exit instruction is a last instruction of the program.

experiments "exact" execution profiles are used instead of deriving them via a path analysis. This allows us to assess the effectiveness of the pipeline analysis without the influence of possibly pessimistic path analyses. The profilers used to create the profiles are produced with the help of qpt2 (Quick program Profiler and Tracer) [1] that is part of the WARTS distribution. An execution profile maps instruction/context pairs to execution counts:

$$profile : IC \rightarrow \mathbb{N}_0$$

We have chosen four small programs (see Table 4) to test our implementation.

Program	Comment
lsimple	simple **for** loop
matmult	matrix multiplication
fib_r	recursive computation of the 23. Fibonacci number
pi	approximative computation of Pi

Table 4: List of test programs

6.1 Improvements by the pipeline analysis

To show the effectiveness of our pipeline analysis we compare the results of the combined instruction cache/pipeline analysis[1] with an (virtual) analysis without cache and pipeline behavior prediction, and with our cache analysis. The CPI (Cycles Per Instruction) values of the different analyses are compared.[2] For the SuperSPARC I the best CPI value that an analysis without cache and pipeline behavior prediction can reach is 13[3] (assuming no overlap of instructions and 100% instruction cache miss rate). The best CPI value that can be reached by a cache analysis alone is 4.

Table 5 displays the improvement by the pipeline analysis. In the second column the CPI value according to the combined cache/pipeline analysis is shown. The improvement factor against the best possible results of an analysis without cache/pipeline behavior prediction by the cache analysis is shown in the third row. The improvement factor of the combined instruction cache/pipeline analysis against an optimal instruction cache analysis can be found in the fourth column. The last column shows the improvement of our combined cache/pipeline analysis against the best possible results of an analysis without cache/pipeline behavior prediction.

Program	*CPI*	Cache Analysis improv. factor	Additional improv. factor by Pipeline A.	Combined improv. factor
lsimple	0,556	3,24	7,19	23,29
matmult	3,436	3,24	1,16	3,75
fib_r	1,800	3,24	2,22	7,19
pi	3,138	3,24	1,27	4,11

Table 5: Improvements by the pipeline analysis.

7 Related Work

Healy, Whalley and Harmon developed an approach [7] to predict worst case execution times in the presence of instruction caches and simple pipelines. Their analyzer predicts the WCET of a user specified program part. The results of a preceding cache analysis are used. Their target processor is a MicroSPARC I. Since the pipeline of this processor is pretty simple they can limit the used resources to registers and pipeline stages.

For each instruction type several informations must be presented to the analyzer. These are the first and the last pipeline stage from or to which forwarding is possible and the maximum number of clock cycles per pipeline stage. Each instruction is assigned the registers it uses and the result of the preceding cache analysis.

The analysis of a program path is done by repeated concatenation of instructions. Healy et al. use a bottom up algorithm to apply their approach to programs with loops. For the analysis of the innermost loop all paths through it are merged. The results of this analysis are used in the next higher loop level as if the inner loop was a single instruction. The distinction between first and other loop iterations is not done explicitly but by the cache classifications (*first miss, first hit, always miss, always hit*). For the step from an inner to an outer loop it can be necessary to apply adjustments to the result of the analysis. To avoid underestimations in the presence of pipelines they have to use a trick which involves adding of miss penalties and subtracting them later at outer loop levels. This trick doesn't work with

[1] 100% data cache miss rate for load instructions and 0% for stores are assumed because of the SuperSPARC store buffer.

[2] We also did some measurements on a Sun SPARCstation 10 with a SuperSPARC I under NetBSD. However these measurements are only statistical results, since we couldn't yet measure without the influence of a non real time multiuser/multitasking operating system. But we believe, that the statistical approximated run time values give a fairly good impression of the capabilities of our pipeline analysis. The ratio of predicted run time and statistical evaluated measurements are 1.01 (lsimple), 1.10 (matmult), 1.03 (fib_r) and 2.40 (pi with normal operands).

[3] Cache miss penalty of 9 cycles plus a minimum of 4 cycles (8 half cycle pipeline stages) for an integer instruction.

superscalar processors since for superscalar pipelines a cache hit can result in a speed gain that is higher than the miss penalty due to grouping effects [18].

Another approach to predict the WCET of real time programs is presented in [10] by Li, Malik and Wolfe. This is an integrated solution, where both program path analysis, and cache and pipeline behavior prediction are based on integer linear programs. The target processor is the i960KB from Intel, which has a pretty simple pipeline. Thus Li, Malik and Wolfe can limit their efforts to the detection of structural hazards. While analyzing simple pipeline and direct-mapped caches is fast, increasing levels of cache associativity sometimes lead to prohibitively high analysis times.

Widely based on [10], Ottoson and Sjödin [17] have developed a framework to estimate WCETs for architectures with pipelines, instruction and data caches. To predict the pipeline behavior they also use a kind of path concatenation like Healy et al., where the maximal overlapping of two instructions is computed. Ottoson and Sjödin restrict themselves to pipeline stages as resources. In an experiment to predict the cache behavior of a very small program they report analysis times of several hours.

In [11] Lim et al. describe a general framework for the computation of WCETs of programs in the presence of pipelines and caches. To model the pipeline behavior they construct a *reservation table* of resources for each instruction. Registers and pipeline stages are regarded as resources. Lim et al. also use a kind of path concatenation. They use a bottom up algorithm that starts with isolated program constructs. A new reservation table is computed, each time an instruction (or a path) is appended to a path. The reservation tables are shortened if possible, by keeping only information from the beginning and the end of the path. Lim et al. focus on the R3000 processor from MIPS, which has a simple five level integer pipeline.

The more recent work of Lim, Han, Kim and Min [12] replaces the reservation tables by *instruction dependence graphs*. In this work they focus on a virtual processor with an idealized multiple issue pipeline. Like in [11] concatenation and pruning of paths is done during the execution of the bottom up algorithm. Lim, Han, Kim and Min don't consider caches or prefetch queues.

Lundqvist and Stenström describe a simulation based approach in [13]. The theoretical advantage of a simulation is that the values of all operands are known and infeasible paths can thereby be eliminated. Usually this is also the drawback of a simulation approach, since the input is generally unknown. To circumvent this problem Lundqvist and Stenström introduce *unknown values*, i. e. their simulation is capable of handling programs even if the input values are not known. The introduction of unknown values leads to several problems. For instance if the target address of a store instruction happens to be an unknown value, the whole main memory becomes unknown. Lundqvist and Stenström try to shrink this problem by reducing the amount of effected memory through relinking of programs in case of statically linked routines. To circumvent the simulation of each path in a loop iteration a path merging strategy is used. The merging of paths leads to loss of information, i. e. to unknown values. The authors report that this loss of information frequently leads to non-termination of the simulation, even if the simulated program terminates. The detection of infeasible paths is also affected by the information loss. The target processor is a PowerPC.

We are aware of two retargetable pipeline analysis approaches that are based on Maril (Marion's machine description language) of the Marion [2] system of David G. Bradlee. Hur et al. [9] have developed a retargetable timing analyzer that has been used to generate analyzers for the MIPS R3000/R3100 and the Motorola 88000. Narasimham and Nilsen describe in [16] a retargetable tool called pipeline simulator compiler that determines the number of cycles necessary to execute a given instruction sequence assuming 100% cache hits. For their tool there are processor descriptions modeling the pipeline behavior of the MIPS R2000, the Power PC 601, and a SPARC computer architecture. For a more detailed discussion of these approaches see [4].

8 Conclusion and Future Work

We have shown that the semantics based approach can be used to predict the behavior of modern pipelines. The presented semantics was designed for superscalar processors, but is also suitable to model out of order execution processors.

The results of our first implementation for the SuperSPARC I processor show a clear improvement against the naive approach.

Our implementation has shown that with the VIVU approach it is possible to realize the instruction cache and pipeline behavior prediction independently without significant loss of accuracy. The advantages of this approach are that there is no need to bother about worst case paths during our pipeline analysis since our results reflect all paths, and that a subsequent path analysis can access context specific information about the behavior of instructions.

Future work includes the development of pipeline analyses for other processors, especially for processors with out of order execution. Work on the integration of the pipeline analysis with the data cache analysis is also in progress. For target systems with out of order execution of data memory accesses it is not sufficient to

treat data cache and pipeline behavior prediction independently.

Additionally it is planned to integrate the pipeline analysis with the path analysis, like it has been done for the cache analysis [20].

A further step can be to incorporate the results of an analysis of floating point operands. This can be important for processors like the SuperSPARC I which show a significant different execution time of floating point operations in dependence of their operand values.

Our goal is to develop a set of tools which allow to create a pipeline analysis for a new processor from a concise description of this processor.

Acknowledgements

Many members of the compiler design group at the Universität des Saarlandes, especially the members of the USES (Universität des Saarlandes Embedded Systems) group deserve acknowledgement. Reinhard Wilhelm provided many valuable hints and suggestions, and has carefully read draft versions of this work. Florian Martin provided his vast knowledge about PAG and adjusted this tool for the special needs of the pipeline analysis.

We 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), and the anonymous reviewers for their helpful comments.

References

[1] T. Ball and J. Larus. Optimally Profiling and Tracing Programs. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, pages 59–70, Jan. 1992.

[2] D. G. Bradlee. Retargetable Instruction Scheduling for Pipelined Processors. PhD Thesis, Technical Report 91-08-07, University of Washington, 1991.

[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, Los Angeles, CA, 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 and R. Wilhelm. On predicting data cache behavior for real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 1998.

[7] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 288–297, Dec. 1995.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 2 edition, 1996.

[9] Y. Hur, Y. H. Bea, S.-S. Lim, B.-D. Rhee, S. L. Min, Y. C. Park, M. Lee, H. Shin, and C. S. Kim. Worst Case Timing Analysis of RISC Processors: R3000/R3010 Case Study. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 308–319, Dec. 1995.

[10] 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.

[11] 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.

[12] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A worst case timing analysis technique for multiple-issue machines. In *Proceedings of the IEEE Real-Time Systems Symposium '98*, 1998.

[13] T. Lundqvist and P. Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1998.

[14] F. Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[15] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of Loops. In *Proceedings of the 7th International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*. Springer, March/April 1998.

[16] K. Narasimham and K. D. Nilsen. Portable Execution Time Analysis for RISC Processors. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1994.

[17] 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.

[18] J. Schneider. Statische Pipeline-Analyse für Echtzeitsysteme. Diplomarbeit, Universität des Saarlandes, Oct. 1998.

[19] SPARC International, Inc., Menlo Park California, U.S.A. *The SPARC Architecture Manual, Version 8*, 1992.

[20] H. Theiling and C. Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proceedings of the IEEE Real-Time Systems Symposium '98*, pages 144–153, Dec. 1998.