Pipeline Behavior Prediction for Superscalar Processors

J. Schneider
Ch. Ferdinand
R. Wilhelm

Technischer Bericht Nr. A/02/99

Technischer Bericht Nr. A/02/99

Jörn Schneider
Reinhard Wilhelm

FB 14 – Informatik
Universität des Saarlandes
Postfach 151150
D-66041 Saarbrücken
Germany

e-mail: `js@cs.uni-sb.de`,
`wilhelm@cs.uni-sb.de`

Christian Ferdinand

AbsInt Angewandte Informatik GmbH
Universität des Saarlandes
Starterzentrum – Gebäude 45
D-66123 Saarbrücken
Germany

e-mail: `ferdinand@AbsInt.de`

Abstract

For real time systems not only the logical function is important but also the timing behavior, i. e. 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 microarchitectural 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 report describes a method to predict the behavior of pipelined superscalar processors and an implementation of this approach for the SuperSPARC I microprocessor. The results of a preceding cache behavior prediction is taken into account. The method uses static program analysis. The implementation has been realized using the PAG (Program Analyzer Generator) tool. The approach is independent of the source language as it works directly on the instruction level.

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. A systematic approach to check whether these temporal requirements are met is schedulability analysis. Information necessary to analyze the schedulability of a given set of tasks embraces upper bounds for the worst case execution times (WCETs) of the particular tasks. In general, it is not possible to determine these upper bounds by measurements or simulation, because all possible combinations of input values would have to be considered[1]. The presented work uses static program analysis to compute the WCET of programs. Our method uses abstract interpretation, a semantics based method of static program analysis.

To obtain sharp bounds for the WCET it is necessary to consider the features of the target processor. Modern processors gain speed through an increase of their clock rate, the use of caches and pipelines, and the parallel execution of instructions.

The influence of the clock rate is clear. To consider the remaining three features within WCET estimation isn't trivial at all. Caches have been addressed in the work of Christian Ferdinand [7, 8, 9]. In this report the prediction of pipeline behavior combined with parallel execution on a superscalar

[1] The worst case input is in general unknown.

microprocessor is presented. For a first implementation of the superscalar pipeline analysis [22, 23] the SuperSPARC I microprocessor [4, 25, 1] has been chosen.

To derive the WCET of a program the worst case program path and the worst case execution time of the instructions of this path has to be known. Clearly there is a dependence between these two. The worst case path depends on the execution time of the instructions. Unfortunately in times of microarchitectural features like cache and pipelines, there is also a dependence in the other direction. The execution time of instructions depends on the taken path (history of execution).

We solve this dilemma by predicting the worst case execution time of instructions for all paths[1] (microarchitectural analysis) and deciding the worst case path (path analysis) based upon this results. This report focus on the microarchitectural analysis part. The path analysis is treated in [26, 27].

1.1 Analysis framework

The microarchitectural part of the execution time prediction is divided into several steps. First of all the control flow information is extracted from the executable program. This information is used by all later steps. At this point a value analysis can be started. This can be necessary to predict the addresses of memory references when analyzing data cache behavior. Thereafter the cache analyzer classifies memory accesses as cache hits or cache misses. This classification is used by the pipeline analyzer. The results of the pipeline analysis are the latencies[2] of any instruction in the considered path classes. These results are used by the path analysis to predict the worst case path and thereby an upper bound for the actual WCET of the analyzed program. The interplay of the different analyses is shown in Figure 1.

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) and by simultaneous processing of different stages. E. g.

[1] We do not explicitly compute the execution time for each path, but distinguish only between first and other iterations of loops (and recursions), thereby building a finite set of path classes. For details see Section 6 and [19].

[2] No. of elapsed clock cycles until instructions have entered the pipeline.

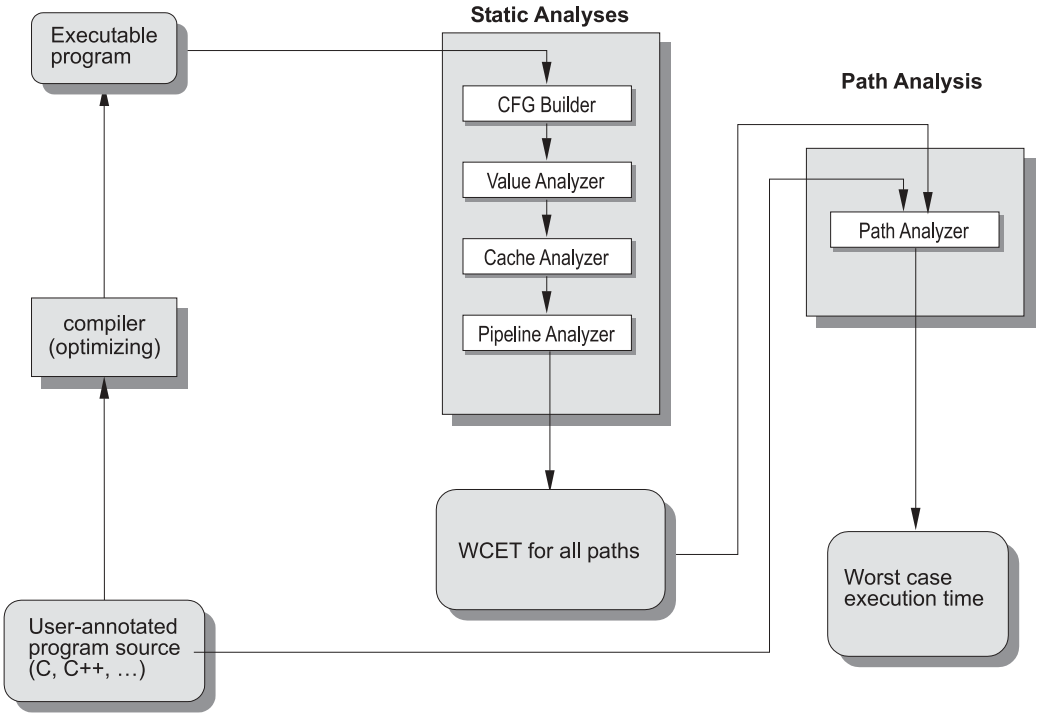

Figure 1: Analysis framework

if each instruction is divided into five stages there can be up to five simultaneously processed instructions, i. e. the first instruction is in stage five, the second in stage four, and so on.

The pipeline in Figure 2 consist of five pipeline levels. Each stage represents one phase of instruction execution: *Fetch*, *Decode*, *Execute*, *Memory Access*, and *Write Back*.

The tasks of the individual pipeline stages of this example are described in the following.

2.1.1 Fetch

The task of the FI (*Fetch Instruction*) stage in the pipeline is to provide the processor with instructions. Therefore, instructions are fetched from main or cache memory. The duration of this stage depends on whether these instructions are in the cache or not, i. e. in case of a cache miss the *cache miss penalty* is added. The typical cache miss penalty increases the processing time of this stage by factor ten. During the processing of a cache miss the earlier instructions, i. e. the ones in higher pipeline stages, can still advance through the pipeline. The resulting "holes" in the pipeline are called

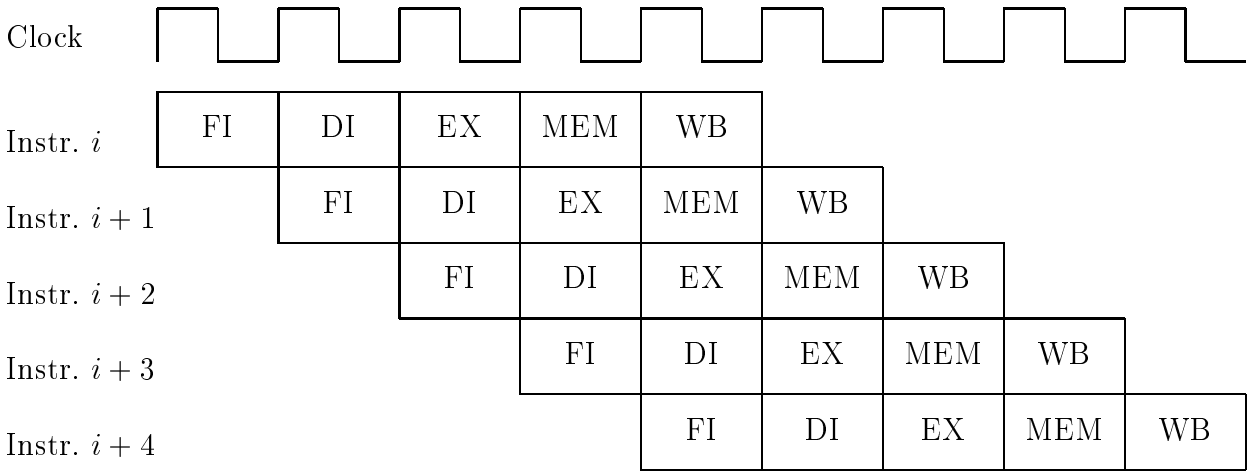

Figure 2: Pipelined execution of instructions in a five stage pipeline: FI (*Fetch Instruction*), DI (*Decode Instruction*), EX (*Execute*), MEM (*Memory access*) and WB (*Write Back*).

pipeline bubbles. In the fetch instruction stage of the pipeline all instructions are treated equally. The processor does not yet care (even doesn't know) about the different types of instructions.

2.1.2 Decode

The main task of this stage is to identify the individual instructions and their resource requirements. The type of an instruction determines which functional units are required to execute these instruction. The operands of an instruction are registers or immediate values.

Since the available resources are limited, there is potential for conflict. To resolve this and other kinds of conflicts (see Subsection 2.2) pipeline bubbles are inserted between the competing instructions.

A further task of this pipeline stage is to prepare the operand values for further computation, i. e. the appropriate registers are read and the immediate values are converted if necessary.

2.1.3 Execute

In this stage the actual computation is done. These computations can embrace one or more of the following actions:

- Arithmetic/logic computations by the ALU (Arithmetic Logic Unit)

- Address computation for subsequent memory accesses
- Destination address computation for control flow changes, e. g. branches

For many processors the duration of this pipeline stage depends on the complexity of the particular computations.

2.1.4 Memory Access

The data access to main memory or cache memory is done in this pipeline stage. This means that the source or destination address must have been already computed. In case of a data cache miss the duration of this stage is prolonged by the miss penalty.

2.1.5 Write Back

After the computation/arrival of the results these are written to their destination, e. g. a register, the data cache or the main memory (maybe via a store buffer). This is usually the last execution step of an instruction.

2.2 Hazards and Stalls

The ideal case for the pipelined execution of instructions is as follows:

- The pipeline can always be filled with instructions
- The pipeline doesn't stall

Unfortunately, this is not always the case. As mentioned earlier there are several possible reasons for a delay of the pipelined processing of instructions. The potential reasons for a pipeline stall are called *hazards*. There are three kinds of hazards:

1. *Structural hazards* are the situations where resource conflicts arise.
2. *Data hazards* are caused by data dependences.
3. *Control hazards* can arise in case of control flow changes.

The occurrence of a hazard can cause a pipeline stall, i. e. the affected instructions wait until the earlier instructions have advanced, so that the hazard vanishes.

2.2.1 Structural Hazards

Most modern processors with pipelines do not have enough resources (e. g. functional units) to execute all possible combinations of instructions fully overlapped. Thus resource conflicts occur. The combinations which can lead to resource conflicts are called structural hazards. If these hazards are *exposed* (i. e. there are no reasons, e. g. a cache miss, that prevent the involved instructions from occupying the same resources simultaneously) the pipeline is stalled to avoid the concurrent access.

Consider a processor with only one memory port. While one instruction uses this port to access data in memory, it is impossible to fetch another instruction through this port. The resulting situation for our example pipeline is displayed in Figure 3.

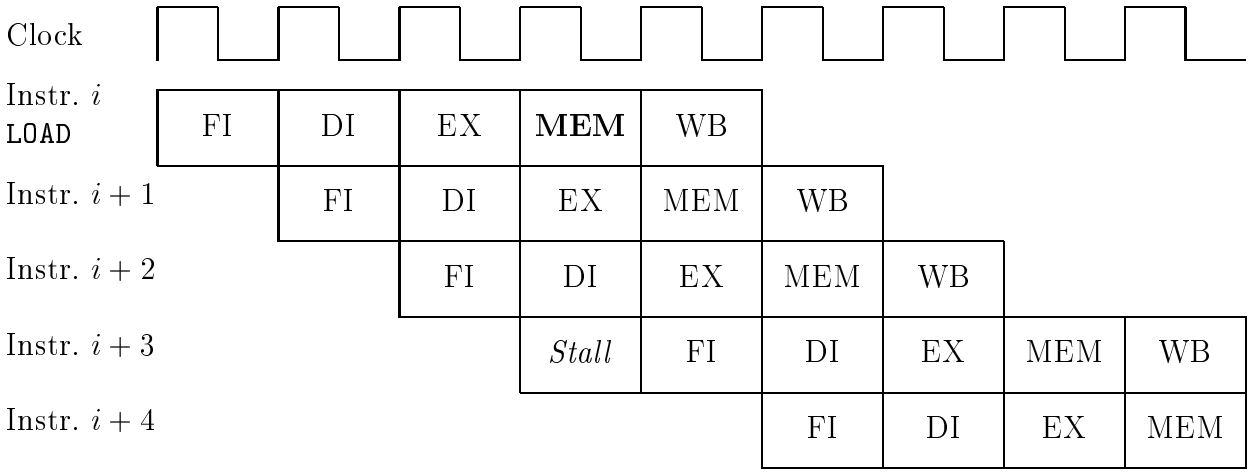

Figure 3: *Structural hazard* – `LOAD` on a microprocessor with one memory port

2.2.2 Data Hazards

The predetermined order of operand accesses of sequential instructions can be destroyed due to overlapped execution in a pipeline. Whenever this effect can lead to miscalculations it is called a *data hazard*.

Consider our example pipeline. Registers are read in DI and written in WB. That means the registers of the second instruction are read before the registers of the first are written. This causes no problems as long as the operands are independent. Otherwise, a data hazard can arise.

Example 2.1
Consider the following example:

```
ADD R2,R3,R1
SUB R4,R1,R5
```

The `ADD` instruction adds the two values in R_2 and R_3 and writes the result into R_1. The `SUB` instruction subtracts the values of R_4 and R_1 and writes the result into R_5. The resulting situation for our example processor is displayed in Figure 4.

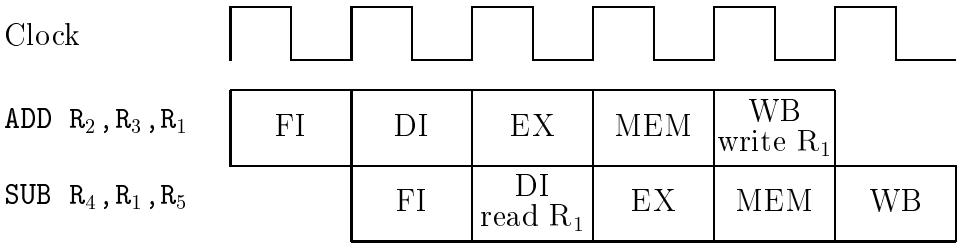

Figure 4: *Data hazard*

While `SUB` reads its operands in the DI pipeline stage, `ADD` is in EX. The register R_1 is not written until `ADD` is in WB. □

The problem described above is usually solved using *forwarding paths*, i. e. the results of the first instruction are immediately forwarded from the *execute* pipeline stage to the *decode, execute* and *memory* stages of the subsequent instruction. Where this solution is not possible a pipeline bubble must be inserted.

Let `A` and `B` be two consecutive instructions. The three possible types of data hazards are as follows:

- RAW, *read after write (dependence)*—`B` tries to read out of a location which is not yet written by `A`. This is the most common case of a data hazard. It is the case of the above example.

- WAR, *write after read (antidependence)*—`B` wants to write to a destination which is not yet read by `A`. Without prevention `A` would read the wrong new value. This case cannot occur in our example pipeline, since all write operations are performed in the last pipeline stage.

- WAW, *write after write (output dependence)*—`B` wants to write to a destination that is not yet written by `A`. Without prevention the value written by `B` would be overwritten. This case can only emerge in a

pipeline where write operations are permitted in more than one stage or in out-of-order execution processors.

The RAR (*read after read*) case is not a hazard, since no change of data occurs.

2.2.3 Control Hazards

Control flow changes (e. g. jumps or branches) can cause pipeline stalls if the destination address is not known early enough. These situations are called control hazards. There are two possible reasons why the destination address might not be known in time:

- The destination address is not yet computed.
- The condition of a conditional control flow change (e. g. a branch) is not yet resolved.

Example 2.2
Let `BC` be a conditional branch, `T,T1,T2` the jump target and its successors, `S, S1, S2` the sequential successors of the branch instruction:

```
BC L
S
S1
S2
...
L: T
T1
T2
```

The resulting situation for the taken branch is displayed in Figure 5. □

To reduce the influence of control hazards on the system throughput a concept called *delayed control transfer* is used in many microprocessors. The sequential successor of the branch is the *delay instruction*. The delay instruction is executed regardless of the branch direction, i. e. even if the branch is taken the immediate sequential successor of the branch is executed. It is up to the compiler to find a proper delay instruction (if anything else fails a `NOP` instruction is used).

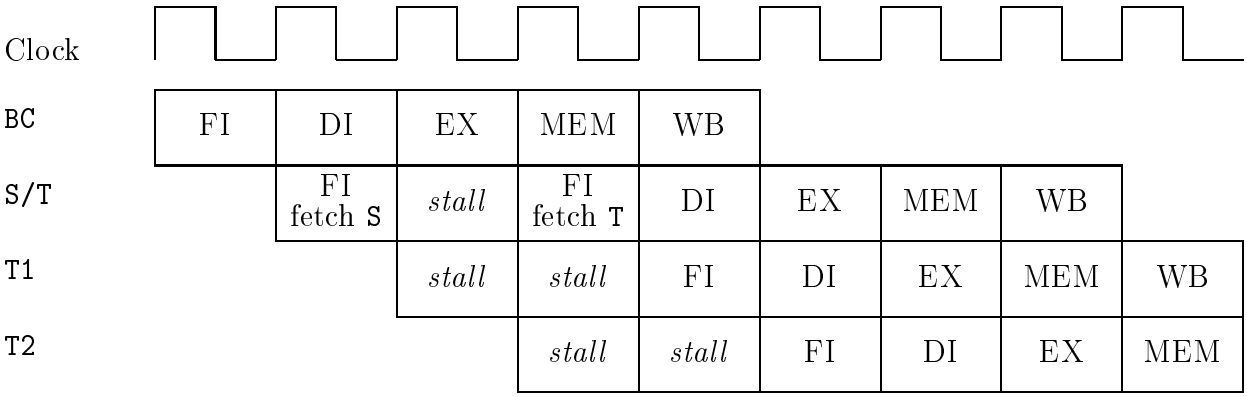

Figure 5: *Control hazard*—taken branch. Initially only the sequential successor (`S`) is fetched. After identifying the branch, resolving the branch direction, and computing the destination address the target instruction (`T`) is fetched.

2.3 Superscalar Execution

A measure for the throughput of a processor is the CPI value (Cycles Per Instruction). The average cycles per instruction of a microprocessor for a program P are:

$$CPI = \frac{\text{Nr. of cycles for } P}{\text{Nr. of instructions in } P}$$

With 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 [11], where the compiler is responsible for selecting the instructions to execute simultaneously. In superscalar microprocessors the dynamic approach is used, i. e. the processor selects the instructions to be processed concurrently during run time.

The additional tasks of a superscalar pipeline are presented in the following paragraphs which are structured according to the stages of our example pipeline.

Fetch For superscalar processors more than one instruction should be available for concurrent execution. The already fetched but not yet started instructions are usually held in a prefetch queue, which is filled by fetching more instructions than the processor actually starts. If the prefetch queue is large enough, the effect of a cache miss might be reduced.

Decode Additional tasks are selection and grouping of instructions for concurrent execution. Therefore, the resource conflicts and data dependences between and inside the groups of instructions must be resolved. Also control flow changes have to be taken into account.

Execute In this phase grouped instructions are executed concurrently. To allow data dependences between grouped instructions it is necessary to split this phase into multiple pipeline stages.

Memory Access The number of data memory accesses is limited by the number of memory ports. Usually groups are chosen, so that there is only one data memory reference per group.

Write Back If groups with multiple writing instructions are allowed WAW hazards must be prevented.

3 A real Pipeline

The above given description of superscalar pipelines are rather general. In a real pipeline like the one of the SuperSPARC I microprocessor some peculiarities can be observed. Some of these are discussed now.

3.1 Integer Pipeline

The SuperSPARC I is a highly integrated superscalar RISC microprocessor, fully compatible with the SPARC version 8 architecture [24]. It executes up to three instructions concurrently. It has a main or integer pipeline and a floating point pipeline. The stages of the main pipeline are shown in Table 1.

3.2 Floating Point Pipeline

The processing of a floating point instruction of the SuperSPARC I 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 after the decode phase. The stages of the floating point pipeline are shown in Table 2.

Floating point operands are classified either as normal or subnormal operands depending on the range of their values [11]. The floating point pipeline of the SuperSPARC I is capable to execute floating point operations with normal and subnormal operands. The duration of some floating

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

Table 1: Stages of the integer pipeline.

point operations depends on the value of its operands, since operations on subnormal values are more complicated.

4 Pipeline Analysis by Abstract Interpretation

Abstract interpretation is a well developed theory of static program analysis [6, 28]. 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

Stage	Tasks
FD	Decode Floating point operations
FRD	Read 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.

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. Since we are interested in the pipeline behavior of a program we consider an operational pipeline level program semantics. To predict the pipeline behavior of a program, we approximate its "collecting semantics[1]". 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 programs [7, 9, 27].

The approach works as follows: in a first step, we define the "concrete pipeline semantics" of programs. The concrete pipeline semantics is a simplified version of the operational pipeline level program semantics that describes only the interesting aspects of the pipeline behavior but ignores other details of execution, like input values, 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 [7]. An abstract

[1] In [6], the term "static semantics" is used for this.

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 [7]).

The fact that the concrete pipeline semantics ignores some details of execution doesn't mean that the analysis can't benefit from additional information. The results of the value analysis phase for instance can be used either immediately (e. g. to classify floating point operands as normal or subnormal) or indirect (e. g. via a data cache analysis).

The computation of the abstract semantics has been implemented with the help of the program analyzer generator `PAG` [18], 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.

5 Pipeline Semantics

Before we present our pipeline semantics, which allows to analyze the behavior of programs on superscalar pipelines, we try to motivate its design. The pipeline semantics must at least allow to detect all kinds of hazards and to model the dynamic decisions of the processor (e. g. the selection of instructions for concurrent execution).

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 when the data passes through memory. Dependences over cache entries can be treated within a data cache/store buffer analysis [9]. We assume 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 and checking for dependences, antidependences and output dependences. We can also detect control hazards with information about resource usage: A *control transfer instruction* (CTI) must write to a special resource, e. g. the *next program counter register*. Thus a CTI can be identified and the availability of the target address can be predicted.

To model the dynamic processor decisions (regarding the pipeline) the foundations of this decisions must be observed. These foundations can be both static (e. g. resource demand of instructions) and dynamic (e. g. state of resources). For the selection of instructions for concurrent execution the state of the prefetch queue and the cache behavior is important. The cache behavior is predicted by a separate cache analysis [8]. 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 [7]. We

consider an in-order superscalar processor that can execute a *group* of up to N instructions concurrently.

Traditionally a kind of *static reservation table* for resources is used to predict pipeline behavior. For superscalar processors this is not sufficient, since the assignment of resources to pipeline stages of instructions can change dynamically during the grouping process and the (predicted) state of some resources must be known.

5.1 Concrete Pipeline Semantics

Definition 5.1 (resource association)
Let $R = \{r_1, \ldots, r_m\}$ be the set of resources and resource types 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 5.2 (resource association sequence)
A sequence $\bar{r} \in \bar{R} = \mathcal{R}^*$ is a *resource association sequence*. Let "." be the concatenation operator for resource associations. □

Definition 5.3
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 5.1
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 demands 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.
- The allocation of a resource can occur in a higher pipeline stage.
- The number of repetitions of pipeline stages can be increased.

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 5.4 (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 $r_{\#} \in R_{\#} = ((\bar{R})^N)^{PS}$ and the state $s_R \in S_R$ 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 5.5 (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 5.6 (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 $\mathcal{U}(p, i)$. □

[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.

Definition 5.7 (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 number of cycles needed to reach the empty pipeline state P_ε. □

5.2 Control Flow Representation

We represent programs 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: $res_v : V \to \bar{R}$.

We extend the update function $\mathcal{U}$ 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 for 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:

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

5.3 Abstract Semantics

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

Definition 5.8 (abstract pipeline state)
An *abstract pipeline state* $\hat{p} \subseteq P$ is a set of concrete pipeline states. $\hat{P} = 2^P$ the set of all abstract pipeline states denotes the *abstract domain*. The abstract domain $\hat{P}$ forms a complete lattice, i. e. the power set lattice, with set inclusion $\subseteq$ as its partial order, set union $\cup$ as its least upper bound, set intersection $\cap$ as is greatest lower bound, $\bot_{coll} = \emptyset$ as its least element and $\top_{coll} = 2^P$ as its greatest element. □

An abstract pipeline state at a program point reflects all concrete pipeline states that may occur at this point. It may contain concrete states that cannot occur at this point, due to infeasible paths.

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 5.9 (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 we extend the join function to sequences of instructions:

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

5.4 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 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 fixed point iteration. The iteration starts with the empty abstract pipeline state.

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

Let $\hat{p}_1, \ldots, \hat{p}_n$ be the abstract pipeline states at the immediate predecessors of v 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 v to enter the pipeline is determined with the help of the *cycle function* as defined in 5.1:

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

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 5.1

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

5.5 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 an explicit parameter $cr \in CR = \{hit, miss\}$ is used. 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. We model the prefetch queue as a resource with its own state. The following notation is used: $p.s_R$ denotes the prefetch queue state, $p.R_{\#}[s]$ denotes the resource allocation sequences of the instructions in stage s, nPC^w denotes the write port of the next program counter register.

$\mathcal{U}(p, ni, cr)$:

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

addr := *get_address*(ni);
if *head*($p.s_R$) $\neq$ *addr* **AND** $cr = miss$ **then**
// ni neither at top of prefetch queue, nor in cache
// $\Rightarrow$ 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 := insert_pipeline_bubble(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 := insert_pipeline_bubble(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);

We model the grouping process 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 deeper in the pipeline.

Example 5.2
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 arising problems 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 5.3
Consider the first rule of Example 5.2. Only the resource allocation sequence of the various types of load and store instructions contain the data cache. Therefore, we can trigger this rule just by looking at the instruction type. In this case the precondition is not only neccesary but also sufficient. □

We model the stall behavior of the SuperSPARC I also by rules.

Example 5.4
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 5.5
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 5.5.

Figure 6 on the following page 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 cascaded 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. □

call to update/prefetch queue | resource allocation sequences

`A` starts new group
$p_1 = \mathcal{U}(p_\varepsilon, \texttt{A}, hit)$
[8004, 8008, 8012]

	F0	F1	D0	D1	D2	E0	E1	WB
`A`	-	-	-	-	R_1^r,R_2^r	R_1^r,R_2^r ALU_1	-	R_3^w

. . . . . .

grouping `A` and `B`
$p_2 = \mathcal{U}(p_1, \texttt{B}, hit)$
[8008, 8012]

	F0	F1	D0	D1	D2	E0	E1	WB
`A`	-	-	-	-	R_1^r,R_2^r	R_1^r,R_2^r ALU_1	-	R_3^w
`B`	-	-	-	-	R_3^r,R_5^r	R_3^r,R_5^r	R_3^r,R_5^r ALU_2	R_4^w

. . . . . .

`C` causes stall
$p_3 = \mathcal{U}(p_2, \texttt{C}, hit)$
[8012]

	D2	E0	E1	WB
`A`	R_1^r,R_2^r	R_1^r,R_2^r ALU_1	-	R_3^w
`B`	R_3^r,R_5^r	R_3^r,R_5^r	R_3^r,R_5^r ALU_2	R_4^w

	F0	F1	D0	D1	D2	E0	E1	WB
`C`	-	-	-	R_4^r	R_4^r	DC	DC	R_3^w

. . . . . .

`D` starts new group
$p_4 = \mathcal{U}(p_3, \texttt{D}, hit)$
[-]

	E1	WB
`A`	-	R_3^w
`B`	R_3^r,R_5^r ALU_2	R_4^w

	D0	D1	D2	E0	E1	WB
`C`	-	R_4^r	R_4^r	DC	DC	R_3^w

	F0	F1	D0	D1	D2	E0	E1	WB
`D`	-	-	-	-	R_6^r,R_3^r	R_6^r,R_3^r ALU_1	-	R_7^w

Figure 6: Application of the concrete update function on Example 5.5

6 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. The execution of the loop body usually changes the cache contents. The pipeline behavior often depends on the cache behavior. Therefore it is useful to distinguish the first iteration of a loop from the other ones.

For our analysis we treat loops as procedures so that we can use existing methods for interprocedural analysis. Figure 7 shows the transformation of a loop to a procedure. These transformations are done on the control flow graph only and don't affect the program code.

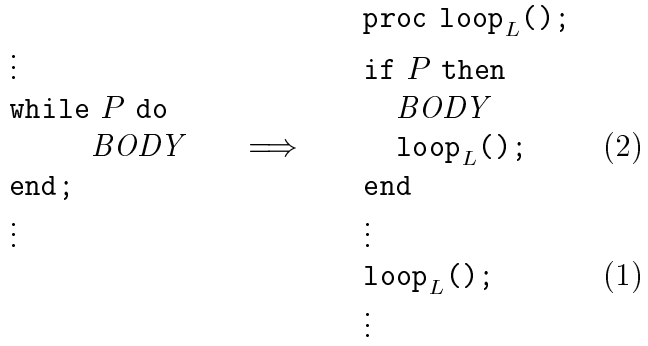

Figure 7: Loop transformation.

In the presence of (recursive) procedures, an instruction 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 an instruction within a procedure. Widely used is the *callstring approach* whose applicability to cache behavior prediction (and thereby to pipeline behavior prediction) is limited [7].

To get more precise results for the cache behavior prediction, the **VIVU approach** has been developed [7, 19]. The VIVU approach 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.

7 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 data dependences and other hazards.

We stated above, that 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.

8 Practical Experiments

In this section the results of a first implementation of the pipeline analysis are presented. We have chosen the SuperSPARC I microprocessor as target system. 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, we don't show this description here (for further information see [22]).

The analyzer takes as input the control flow graph of a program and the results of the cache analysis [8]. We conservatively assume that memory references which 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 Subsection 5.4. 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. It is represented as a sequence of first and recursive function calls ($call\_g_f$, $call\_g_r$) and first and other executions of loops ($loop\_l_f$, $loop\_l_o$) for the functions g and (virtually) transformed loops l of a program. *INST* is

[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.

the set of all instructions in a program. *CONTEXT* is the set of all contexts of a program. *IC* is the set of all instruction/context pairs.

$$\begin{aligned} CONTEXT &= \{call\_g_f, call\_g_r, loop\_l_f, loop\_l_o \mid g \text{ is a function, } l \text{ is a loop}\}^* \\ IC &= INST \times CONTEXT \\ map &: IC \to \mathbb{N}_0 \times \mathbb{N}_0 \end{aligned}$$

In general it is impractical to regard all possible contexts (paths) of a program. Instead of distinguishing all paths, path classes are considered for which similar behavior is expected.

This is realized by the **VIVU** approach (see Section 6). The advantage of this approach is, that our results reflect all paths of the input program. The result for a instruction in a path class is representative for each execution of this instruction in the included paths. There is no need to bother about worst case paths during our pipeline analysis, and a subsequent path analysis can still benefit from the knowledge about the context of an instruction.

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

To predict the WCET of a program it is necessary to find a conservative approximation of the worst case execution path. This is usually the task of a path analysis. The inputs for such a path analysis are the results of our pipeline analysis and perhaps some user annotations, like maximal iteration counts of loops. Thus an architecture dependent worst case execution profile can be determined.

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

For our experiments we used "exact" execution profiles instead of deriving them with a path analysis. This allows us to assess the effectiveness of our 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) [3, 13] that is part of the WARTS distribution. An execution profile maps instruction/context pairs to execution counts:

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

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

No.	Program	Comment
1	lsimple	simple **for** loop
2	matmult	matrix multiplication
3	fib_r	recursive computation of the 23. Fibonacci number
4	pi	approximative computation of Pi

Table 4: List of test programs

8.1 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 a (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 with 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 of the cache analysis against the best possible results of an analysis without cache/pipeline behavior prediction is shown in the third row. The improvement factor of the combined 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.

[1] We assumed 100% data cache miss rate for load instructions and 0% for stores 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.

Program	*CPI*	Cache Analysis improvement factor	Additional improvement factor by Pipeline Analysis	Combined improvement factor
lsimple	0,555570	3,24	7,19	23,29
matmult	3,435746	3,24	1,16	3,75
fib_r	1,800034	3,24	2,22	7,19
pi	3,137949	3,24	1,27	4,11

Table 5: Improvements by the pipeline analysis.

9 Related Work

Healy, Whalley and Harmon developed an approach [10] to predict worst case execution times in consideration of instruction cache and simple pipeline behavior. 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 through 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 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 [22].

Another approach to predict the WCET of real time programs is pre-

sented in [14] by Li, Malik and Wolfe. This is an integrated solution, where both the program path analysis and the cache/pipeline behavior prediction are based on integer linear programs. The target processor is the i960KB from Intel, which has a pretty simple pipeline. Li, Malik and Wolfe consider only structural hazards. The enhancement of this approach to less simple pipelines or even superscalar or out-of-order pipelines seems to be not very promising, since solving integer linear programs is an NP-hard problem.

Widely based on [14], Ottoson and Sjödin [21] have developed a framework to estimate WCETs for microarchitectures 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, so it is impossible for them to detect data hazards. In an experiment to predict the cache behavior of a very small program they report analysis times of several hours.

In [15] 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 microprocessor from MIPS, which has a simple five level integer pipeline.

The more recent work of Lim, Han, Kim and Min [16] replaces the reservation tables by instruction dependence graphs. In this work they focus on a virtual microprocessor with an idealized multiple issue pipeline. Like in [15] concatenation and pruning of paths is done during the execution of the bottom up algorithm. Each concatenation step necessitates the completion of a three step algorithm and the creation of five tables. Lim, Han, Kim and Min don't consider caches or prefetch queues. The space requirement of their instruction dependence graphs is $O(n^2)$, where n is the number of instructions in the graph.

Lundqvist and Stenström describe a simulation based approach in [17]. 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 simulate the execution of a program for each possible input combination is in general even more impractical than actually executing it with all input combinations. To circumvent this problem Lundqvist and Stenström intro-

duce *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 can lead 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 [5] system of David G. Bradlee. Hur et al. [12] 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 [20] 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 [7].

10 Conclusion and Future Work

We have shown that abstract interpretation can be used to predict the behavior of modern pipelines. The presented semantics was designed for superscalar processors, but is also suitable to reflect the behavior of even more complicated pipelines, e. g. of out-of-order execution processors.

Our work is the first implementation of a superscalar pipeline analysis for a real microprocessor we know of.

The displayed results of our first implementation for the SuperSPARC I microprocessor 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 we need not 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 pro-

cessors, especially for processors with out-of-order execution. We are also working on the integration of the pipeline analysis with the data cache analysis. 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 [27].

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 allows to create a pipeline analysis for a microprocessor from a concise description of this processor. This *retargetable* pipeline analysis includes the generation of a front end and of a description of the analyzer, which can be used to generate the analyzer with PAG.

References

[1] F. Abu-Nofal et al. A Three Million-Transistor Microprocessor. In *Digest of technical papers, IEEE International Solid State Circuits Conference proceedings*, pages 108–109. IEEE, 1992.

[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] 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.

[4] G. Blanck and S. Krueger. The SuperSPARC Microprocessor. In *Spring COMPCON 92*, pages 136–141. IEEE Computer Society Press, 1992.

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] J. Larus. Abstract Execution: A Technique for Efficiently Tracing Programs. *Software Practice and Experience*, 20(12):1241–1258, Dec. 1990.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] J. Schneider and C. Ferdinand. Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, May 1999.

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

[25] Texas Instruments. *TMX390Z50/Z51 Integrated SPARC Processor, data sheet*, May 1994.

[26] H. Theiling. Über die Verwendung ganzzahliger linearer Programmierung zur Suche nach längsten Programmpfaden. Diplomarbeit, Universität des Saarlandes, Sept. 1998.

[27] 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.

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