Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Master thesis

Evaluating compositional timing analyses

submitted by
Claus Michael Faymonville

submitted
September 2015

Advisor:
Sebastian Hahn

Reviewers:
1. Prof. Dr. Jan Reineke
2. Prof. Dr. Sebastian Hack

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,______________________ ______________________________
(Datum/Date) (Unterschrift/Signature)

Abstract

Determining upper bounds on execution times of computer programs running on a certain microarchitecture is a resource-consuming task.

Recent publications suggested that analysing pipeline and caches of the microarchitecture *compositionally* might be a beneficial approach, reducing resource usage during analysis while simultaneously providing only slightly worse bounds.

While compositionality was formally defined and decompositions were suggested, an experimental evaluation is essential to back up ideas and arguments as well as to assess the impact of compositionality, i.e. the trade-off between analysis resource usage and precision.

To assess the compositional approach against the state-of-the-art integrated approach, we create a worst-case execution time analyser supporting both approaches. Afterwards, we analyse example programs using both approaches on two different microarchitectures, an in-order and an out-of-order pipeline.

During the evaluation, a multitude of microarchitectural settings are varied to gain insight into their impact on the performance of worst-case execution time analysis.

Reduced resource usage when analysing pipeline and caches compositionally does not justify the increase of the determined upper bounds in the considered settings. However, the evaluation exposes strengths and weaknesses of the compositional approach and contributes to rate other possible decompositions.

Acknowledgments

First of all, I want to thank my supervisor Jan Reineke for the opportunity to write this thesis, discussing ideas and providing an excellent work environment.

Another thanks goes to the second reviewer, Sebastian Hack.

I want to give a special thanks to my advisor Sebastian Hahn for his patient help on compositionality, LLVM, templates, reviewing the thesis and much more. His discussions, objections, and ideas were a major inspiration for my work.

Further thanks go to

- Michael Jacobs for discussing and reviewing both implementation and thesis,
- my brother Peter for reviewing the thesis and his support throughout my studies,
- as well as Tomasz Dudziak for being always helpful and open discussions about Pitcairn Islands.

Finally, thanks to my parents, family and friends for their support throughout my life.

Contents

1 Introduction

An airbag in a car not deploying when it should may result in the death of its passengers. This is an example for a *hard real-time system*, which expect its components to perform within a given amount of time. It is considered hard, since missing its deadline may result in a catastrophe. Therefore, hard real-time systems need guarantees on both their functional behaviour and timeliness. Using computer programs in hard real-time systems introduces new sources of errors: Unexpected input may cause the program to fail, unlikely behaviour might result in a deadlock (a state it cannot leave by itself), or executing the program takes too long, exceeding the requirement imposed by the physical environment. This thesis is concerned with the last aspect, the runtime of a program in the worst case, denoted as the *Worst-case execution time (WCET)* [WEE$^+$08].

Determining the runtime of a program is not simple. It depends on the program itself, its inputs and lastly, the system or microarchitecture it is running on. In modern microarchitectures with multi-core processing, the runtime also depends on other programs executed in parallel. Inputs to a program are mostly unknown or limited to be in a certain range. Analysing a program — while considering all possible inputs — yields multiple paths a program can take.[1] Analysing the microarchitecture results potentially in several runtimes for each program path. These runtimes, however, also vary due to different initial states of the microarchitecture, which arise from different other programs executing before our program under analysis. Finally, the number of combinations of each program path and each microarchitectural initial state is too large to be able to determine their runtimes individually.

In previous work, WCET analysis has been done in two different fashions: The first method is *dynamic analysis*, where a program is run multiple times while measuring the runtime. Afterwards, a safety margin is added. In order to use a small safety margin, dynamic analysis relies on observing one execution run near the actual WCET. However, program runs with near-to worst runtime may be very unlikely. Therefore, using a measurement-based approach is not sufficient to guarantee a maximal runtime of a program. Using a high safety margin introduces high overestimation, i.e. the difference between determined upper bound and actual WCET, as shown in Figure 1.1. The lower, blue curve, represents possible measurements obtained by a dynamic analysis.

[1]Deciding whether or not a program terminates is undecidable in general, cf. the halting problem. However, we assume to only analyse programs for which termination has been proven in advance.

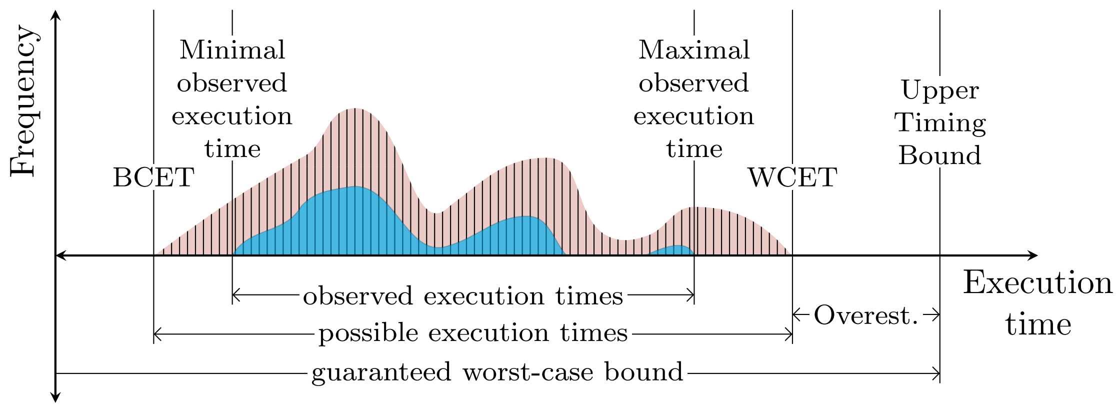


Figure 1.1: Overview of WCET-related notions, based on [WEE+08]

This thesis follows the second kind of WCET analysis, namely *static analysis*. A program analysis is called static, if its result can be obtained without executing the program, and still providing information about all possible executions. Incorporating all possible executions also ensures that the analysis is sound, i.e. that the determined bound is at least as large as the WCET.

To avoid analysing each possible program execution individually, we *abstract* from microarchitectural states, to reduce the amount of program paths and states. This enables to analyse multiple paths at once. The result is an upper bound on the execution times, as depicted in Figure 1.1, which overestimates the WCET.

For some abstract microarchitectural states in a path, certain behaviour is not known and the analysis has to process with uncertainty. A common example for uncertainty is whether the content at a specific memory address is currently cached or not. In this case, the analysis performs a case distinction assuming each scenario once. These *splits* lead to a time-consuming analysis. Assuming the local worst case is not sufficient, since the local worst case does not necessarily lead to the global worst case, as shown in [RWT+06] and [The05].

There are two major goals when performing WCET analysis: Maximizing precision of the upper bound (i.e. minimizing overestimation) and to a lesser extent, minimizing resource usage of the analysis.

Instead of prioritizing precision as much as in the state-of-the-art integrated analysis [The05], the compositional approach [Hah14] decomposes the behaviour of the microarchitecture into several parts and analyses them separately. Uncertainty in one component does not necessarily affect the analysis of another component. Splitting in different components does not lead to a multiplicative grow of states, and thus does not increase the analysis runtime of the other component as much. However, since the real components are typically not fully independent of each other and overlap, analyses do account for some interactions in all components to be sound. Therefore, we expect that the precision of a sound, compositional analysis is not as good as in the integrated approach.

The goal of this thesis is to evaluate the performance of the compositional analysis in comparison to the state-of-the-art approach for one specific decomposition. Measurements for the performance are the above mentioned analysis precision and resource usage. The proposed decomposition covered in this thesis is to separate the pipeline and the memory hierarchy, consisting of caches and a common background memory.

Thesis structure Starting in the next chapter, we explain how state-of-the-art WCET analyses work and present a toolchain to obtain the WCET. WCET analysis is divided in different phases, which are all described here. The integrated and compositional approach differ in one phase. Chapter 2 however will focus on the integrated approach, which is used as a benchmark to evaluate the compositional approach.

Since they are an important part of the timing analysis, Chapter 3 will present two microarchitectures – one in-order and one out-of-order pipeline – which are used in our evaluation. They also serve as examples for common microarchitectural features and the challenges they introduce for WCET analysis.

After the integrated approach and microarchitectures have been introduced, Chapter 4 focuses on the compositional approach. After an introduction to compositionality, we explain how compositionality is applied to pipeline and memory hierarchies. New challenges arise from this application, and we present our solutions to tackle them.

Chapter 5 covers the experimental setup and evaluation and compares the results of both integrated and compositional analysis. We examine and analyse the results. Afterwards, specific microarchitectural features are evaluated in terms of their effect on compositional WCET analysis.

Chapter 6 contains a discussion of possible improvements and future work which should be considered.

Finally, Chapter 7 concludes the thesis with a summary of the results.

Contributions Besides the theoretical insights in Chapter 4, and the experimental findings in Chapter 5, this thesis contributes in multiple ways. It extends the WCET analyser *llvmta* by two pipeline models, the described in-order and out-of-order pipeline, a framework for memory topologies as well as two topology instances. Additionally, a state-sensitive path analysis based on [Ste10] enables more precise results for both approaches of WCET analyses. Furthermore, we provide a multitude of tools to evaluate the analyser, i.e. conduct experiments, collect its results, and convert the results into comprehensible representations.

2 Static Worst-Case Execution Time Analysis

As mentioned in the introduction, there are various factors in a real-time system which affect the runtime of a program: The program code, the input given to the program and the microarchitecture the program is running on. The following paragraphs describe how these factors affect the runtime. Afterwards, the state-of-the-art approach to analyse all factors is presented.

As program code, static timing analysis typically uses the machine code, given by the binary executable. This is beneficial since the machine code is directly executed by the processor, and the code is not modified by a compiler anymore. Also, the memory layout of instructions and variables used is fixed. Accurate addresses for these are important for a precise microarchitectural analysis later.

On the other hand, it is harder to determine bounds on the number of loop iterations in machine code, which can often easily be seen in the source code or an intermediate representation. There is an analysis component which later reconstructs these loop bounds and the Control flow graph (CFG), a structured code representation, which is also lost in the machine code.

The second major influence on execution time, the program input, might change from one execution to the next, and we need a guarantee for all possible inputs to be sound. Therefore, we need to consider all possibilities with different timing behaviour. Although the program input is unknown, it is bounded, at least by its representation in computer hardware, but often even further by possible values provided by the physical environment. More accurate values may eliminate some of the alternatives relating to different timing behaviour.

The last factor affecting the execution time is the microarchitecture the program is running on. Execution time is measured in clock cycles.[1] Depending on the microarchitecture, an operation might take a different amount of cycles. Another difference is the time it takes to obtain the content at a memory address. Yet another factor which causes variance in the microarchitecture is its initial state, i.e. the state when the program starts to execute. The effects of this factor on the execution time can even be unbounded for some microarchitectures, cf. [LS99].

[1]This can later be converted back to some appropriate time unit, e.g. milliseconds, by using the processor clock rate. In this thesis, clock cycles are the standard time unit.

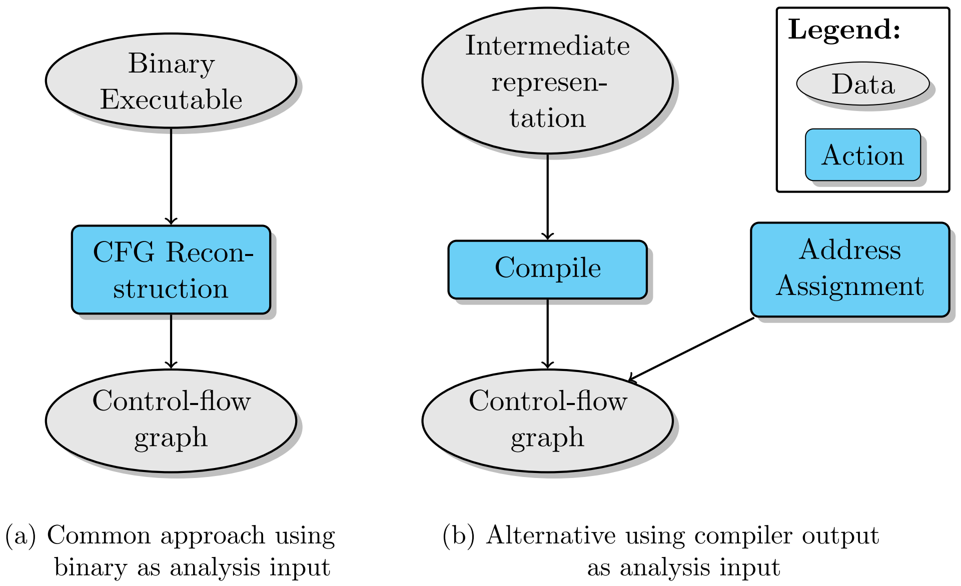
To formalise the dependencies explained above, the WCET of a program P can be described as the maximum execution time of all possible inputs and all possible initial states of a microarchitecture MA, given that the execution time of each given pair of input and initial state can be determined. [WEE+08]

$$WCET_{MA}(P) = \max_{i \in Inputs} \max_{h \in States(MA)} ET_{MA}(P, i, h) \tag{2.1}$$

The execution time $ET_{MA}(P, i, h)$ for a pair of input and initial state could be found by simulating the program execution using a model of the microarchitecture. However, as in dynamic analysis, simulating all combinations is in general not possible, since there are too many. Instead, static approaches use abstraction to analyse multiple paths at once and give an upper bound for the WCET as a result.

We present possible toolchains for static WCET analysis, that is split in three major phases, which are run sequentially: In the first phase, the Control flow graph is obtained. Afterwards, phase two gathers useful information to annotate the CFG. Finally, phase three analyses the program on the microarchitecture and combines all information into a WCET bound.

2.1 Phase 1: Obtaining the control flow graph



(a) Common approach using binary as analysis input (b) Alternative using compiler output as analysis input
Figure 2.1: WCET-analysis, phase 1

Phase 1 starts with the program, usually given in binary form, and constructs the CFG, as described in [The04]. This is another representation, which is more structured. All instructions are separated into Basic blocks. A basic block is defined to be a set of consecutive instructions which are executed together, meaning that the basic block can only be entered at the beginning, and exited at the end, not somewhere in between. Each basic block may have multiple predecessors and successors, except for the first, entry basic block, and the last, terminating basic blocks, cf. [All70, p. 2].

In our evaluation, we use the Low-level virtual machine (LLVM) compiler infrastructure ([LA04]), which enables another method of obtaining the CFG. When compiling, LLVM also produces the CFG at machine code level from its intermediate code representation. The only thing missing from the binary are the addresses of instructions and global variables. Therefore, we need to do an additional address assignment, that must coincide with the addresses in the binary, to be able to proceed with the other stages.

The LLVM compiler infrastructure relaxes the definition of a Basic block (BB) in two aspects. It can have multiple exits, if there are conditional branches before the last instruction. When a conditional branch is taken, instructions afterwards are not executed. Additionally, a basic block might call other functions and continue executing itself afterwards, resulting in a non-consecutive execution. Still, all instructions are executed when calling a function.

Figure 2.1 shows both alternatives for the first phase, with Figure 2.1b referring to the method used in this thesis.

2.2 Phase 2: Supporting analyses

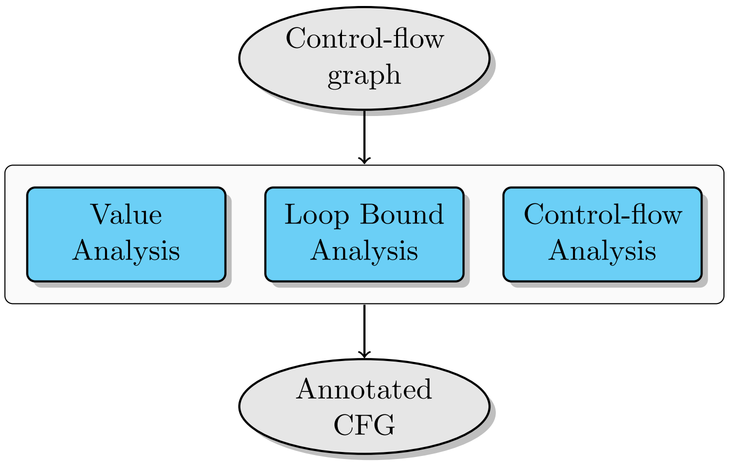


Figure 2.2: WCET analysis, phase 2

The second phase of static WCET analysis uses the result of the first phase as its input, and provides additional information on the program. Three different analyses provide either necessary information (loop bound analysis) or helpful insight to tighten the bound (value analysis, control-flow analysis).

Value analysis The first analysis which provides further information not included in the binary itself is the value analysis. It determines register contents at each program point, i.e. before each instruction. Register contents may contain values of variables or addresses of memory accesses. This enables a precise analysis of the memory behaviour later, e.g. whether an access hits the cache. The memory behaviour is explained in Chapter 3. There are different types of value analyses available, which differ in terms of precision and costs. The variant used in the evaluation here uses a constant value domain, i.e. it determines values which are constant Other abstract domains include interval analysis and octagons, cf. [Cou01].

Context sensitivity Register contents at the beginning of a function called at multiple positions in the program differ significantly. Differentiating between the different function calls can lead to a more precise value analysis. Apart from calls, different contexts can also be used when considering conditional flow or loops. Consider a *if-then-else* section of a program: Register contents afterwards will differ depending on whether the then-branch or the else-branch was taken. In loops, the first iteration often differs from later iterations, where many data accesses can be satisfied by the cache. A generic way of adding context sensitivity is via *trace partitioning*, cf. [MR05].

Loop bound analysis Second in phase two, the loop bound analysis determines the maximum numbers of iterations for each loop in the program. Using the LLVM compiler, loops are identified on intermediate code, where it is easier to obtain bounds by analysis, and again on the machine code level. A bound is found by matching these loops. Additionally, the user is able to provide a loop bound. We use the bounds provided by scalar evolution analysis, cf. [Goh09].

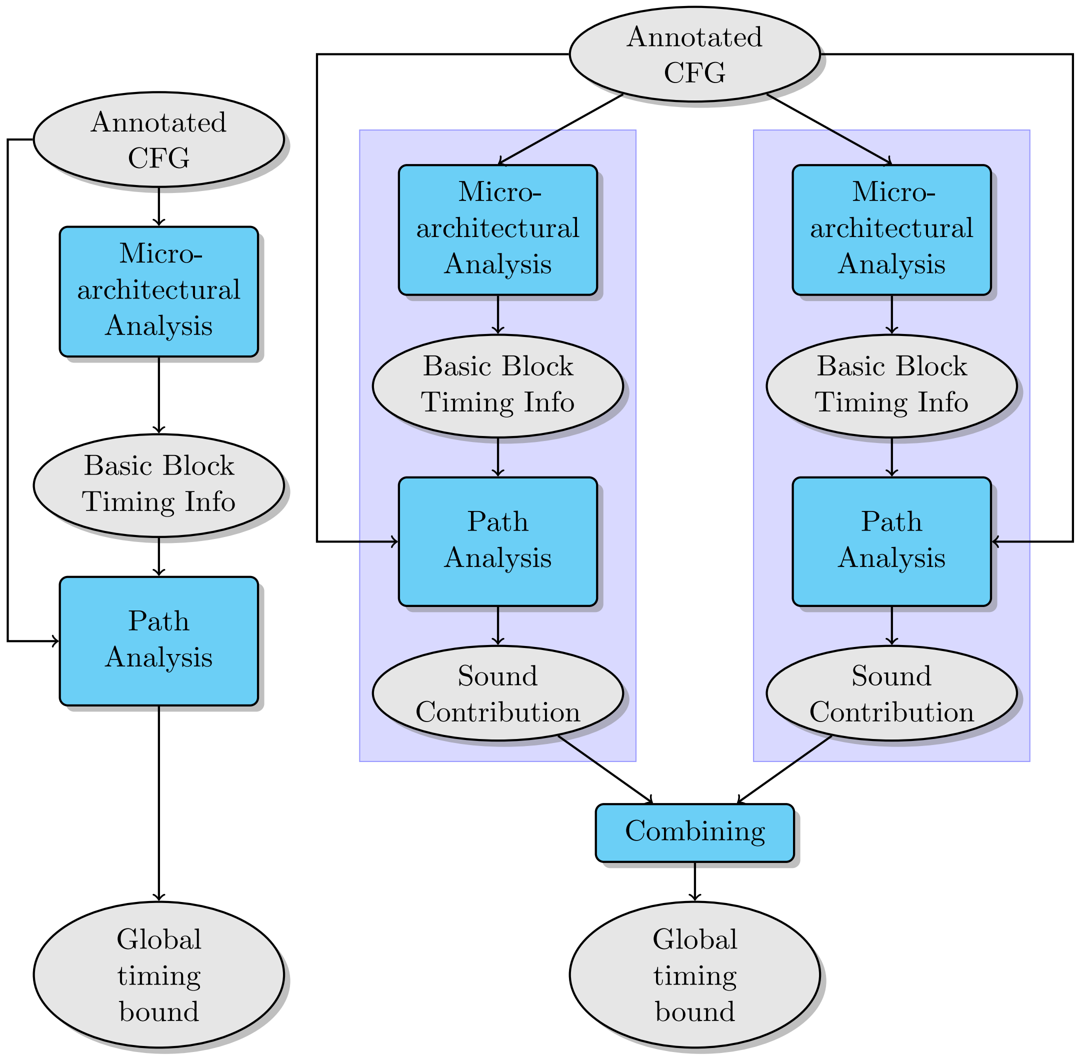
Control-flow analysis Finally, control-flow analysis tries to find infeasible paths, e.g. combinations of conditional flow which are not possible, cf. [GESL06].

While loop bound analysis is mandatory to obtain a finite upper bound for the WCET, control-flow analysis is used to tighten the upper bound.

The phase ends with the CFG and the annotated analysis results.

2.3 Phase 3: Analysing microarchitecture and combining all phases

In the third phase, the microarchitecture is the central element, and one can distinguish between the integrated state-of-the-art approach and our new compositional approach. The following section follows the integrated approach outlined in Figure 2.3a. Details about the actual microarchitectures are explained in Chapter 3. Figure 2.3b shows the toolchain of the compositional approach, which will be explained in Chapter 4.



(a) Integrated approach (b) Compositional approach

Figure 2.3: WCET-analysis, phase 3

Microarchitectural analysis Using the annotated CFG, the microarchitectural analysis determines timings for each basic block. All basic block timings are combined to an upper bound of the WCET in the path analysis afterwards.

Obtaining a timing bound for each Basic block (BB) is done by abstractly simulating the program execution. A *cycle-based* model of the microarchitecture is needed, i.e. the model can determine successor states by processing one cycle starting in a given state. Using this model and starting with the first basic block of the program, the states are cycled until all instructions of the basic blocks finished, i.e. the instruction is no longer present in the pipeline. When all instructions are finished, the number of cycles simulated gives the timing for this basic block.

End states from the first basic block are propagated to their corresponding successor basic blocks. In case of a call inside of a basic block, the current state at the call will get propagated to the first basic block of the called function, and after finishing analysis of the function, states are propagated back to the next instruction after the call. This is done in the same manner with subsequent basic blocks.

Until now, we only have one path through each basic block. As we have seen before, context sensitivity allows for multiple paths, which are then analysed separately.

During the abstract simulation, not all information compared to a real execution is available. As mentioned before, inputs to a program are not known in static analysis. Therefore, the simulation has to account for all possible successor states when cycling. As an example, a branch instruction which depends on a register value may either alter the Program counter (PC) or not. Since this information may not be available, the simulation returns two different states after the cycle. Other sources of *splits* due to uncertainty in the analysis are explained in Chapter 3.

These steps are repeated until a valid bound for each BB is found. However, when propagating states to the succeeding basic blocks, it might happen that the BB was analysed already, e.g. after a loop. If any newly propagated state is different from any already analysed start state, the BB is analysed again. This may cause other BBs to be analysed again as well and is done until no new states arise, i.e. until a fixed point is reached.[The05] The fix-point iteration is guaranteed to terminate, since in each iteration, there can never be less states than before (monotonicity), and the number of possible states is finite.

Path analysis The path analysis is the last component of the WCET framework. It uses the CFG and the results from the preceding analysis, i.e. timings for each basic block, to construct a weighted graph with timing as edge weight, and searches for the longest path in terms the weight. Inter-basic block edges are assigned zero as weight. This is encoded using Integer Linear Programming (ILP) ([LM95]).

Before solving the ILP problem, the CFG is converted into a new graph representation based on [Ste10]. Instead of using only one edge for each basic block, we use all available states from the analysis and create start- and end-states for each basic block accordingly. The resulting state-sensitive graph then has multiple intra-basic block edges per basic block, weighted with the cycle count for this path. Variables in theILP represent how often edges are executed. The objective function tries to maximize the weighted sum of all executions of basic blocks, cf. Equation 2.2.

$$\max \sum_e w_e \cdot x_e \quad \forall e \in Intra - BB - Edges \tag{2.2}$$

The maximum is restricted by several further constraints, which can be categorized by the following:

Start constraint Since we analyse one execution of a program, we need to make sure that we find exactly one path through our graph. We add a new, special vertex, which is further referenced with index 0. From this vertex, edges go to each start-vertex of the first basic block with weight zero. Analogously, we add edges from each end-vertex of terminating basic blocks to the special vertex. We then add the constraint:

$$\sum_i x_{e_{0,i}} = 1 \quad \forall i \in Start(BB_0) \tag{2.3}$$

Flow constraints Conservation of flow is needed to ensure the program is executed properly, i.e. an execution finishes completely. For each vertex i of the graph, the amount of executions of incoming edges has to be equal to the executions of outgoing edges:

$$\forall i \sum_j x_{e_{j,i}} - \sum_k x_{e_{i,k}} = 0 \tag{2.4}$$

Loop constraints In the state-sensitive graph representation, loops are included, however, they are no longer bounded by a maximum iteration count. To encode this, we add a constraint for each loop. The sum of executions of each backedge of the loop can only be executed as often as the determined loop bound. In case of nested loops, any loop can be started multiple times during one program execution. Therefore, the number of executions of all backedges $x_{e_{k,l}}$ is bounded by the number of execution of each incoming edge $x_{e_{i.j}}$ into a loop header, multiplied with the bound:

$$\sum_{e_{k,l}} x_{e_{k,l}} - \sum_{e_{i,j}} x_{e_{i,j}} \cdot Bound \leq 0 \tag{2.5}$$

Call/Return constraints Whenever an instruction calls a function, the program will return to the instruction after the call instruction. However, a function might be called from multiple points in the program.

```
int f(){
        return 37;
}
int main() {
        f();
        f();
        return 0;
}
```

Listing 2.1: Infinite loop due to calls

Listing 2.1 gives a simple example where the ILP can create an unbounded loop. In order to maximize the runtime, the ILP returns to the first call, when the second call was performed. To prevent this, we introduce the following constraint for each state which concerns the edges to called Basic blocks x_{e_i} and back to the callee x_{e_j}. These have to be equal.

$$\sum_i x_{e_i} - \sum_j x_{e_j} = 0 \tag{2.6}$$

In our setup, the ILP is solved using CPLEX[2].

[2]http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

3 Microarchitecture

This chapter presents different microarchitectures implementing an instruction-set architecture. The described microarchitectures do not correspond to specific processors, instead, generic microarchitectural concepts are used to evaluate compositional analyses. Implementing an analysis of a concrete processor is difficult for two reasons: First, implementing all details which may affect WCET analysis takes a long time, and second, not all implementation information is available by the manufacturers. On the other hand, creating a microarchitecture which features certain generic concepts is sufficient to evaluate their impact on WCET analysis.

We present two different processor pipelines will be described, a simple in-order pipeline, and a more complex pipeline with out-of-order execution. Additionally, we distinguish between different memory topologies connected to the pipeline. These can be combined with either pipeline.

3.1 In-order Pipeline

The first and main microarchitecture for evaluation is a five-stage in-order pipeline based on the architecture described in [HP12, App. C]. Stages comprise fetching instructions from memory, decoding and executing them, accessing attached data memory, and writing results back to registers. These stages contain consecutive instructions, work in parallel and therefore theoretically allow for a throughput of one instruction per cycle.

Instruction fetch (IF) In this first stage, the instruction is fetched from the memory using the address given by the Program counter (PC). In case there is no branch instruction executing, the PC is incremented by four[1], which is the address of the next instruction in the binary.

Instruction decode (ID) This stage uses the fetched instruction and decodes it, i.e. it fetches the values of all operands from the register, that are required by the instruction. In case there is an immediate value given in the instruction, it is also computed and provided for the succeeding stage.

[1]One instruction is 32 bit long, and since our memory is byte-adressable, we increment by four to get the address of the next instruction.

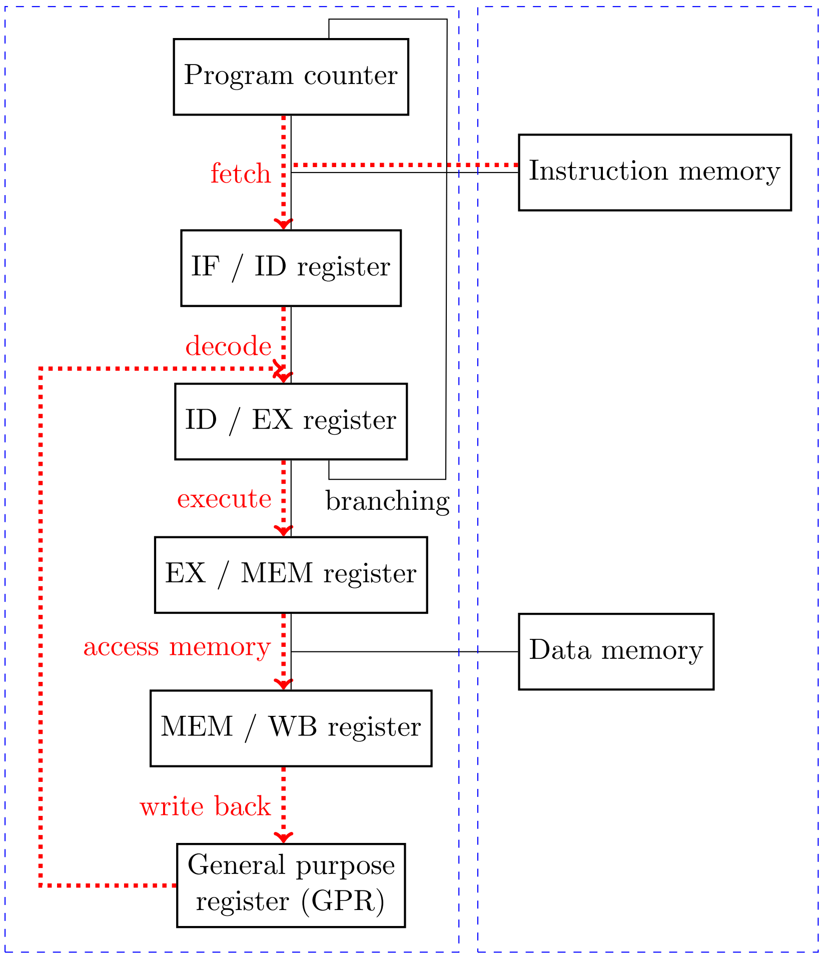


Figure 3.1: Simplified schematic view of in-order pipeline

Execute stage (EX) During execute, the Arithmetic logic unit (ALU) computes the results of the instruction. These can be different calculations, comparisons or branch targets. While some calculations like an addition take a fixed number of clock cycles, a multiplication might depend on the operands provided. In case of a branch, the PC is set in this stage to continue with subsequent instructions at the target address.

Memory access (MEM) The memory access stage is connected to the data memory and handles load and store instructions. In case of a load, the value at the computed address is fetched from memory, which takes variable time depending on how fast the memory can provide the value. Details about this are explained later in the memory section. In case of a store, it can either be blocking or non-blocking. The former means the instruction stays in the memory stage until the access is finished completely, the latter means it proceeds to the next stage while memory takes care of resolving it. This implies that the instruction might finish before all its effects have been performed.

Write back (WB) The last stage, write back, changes the value(s) of the register(s) to either the result of a calculation or a loaded value from memory. The value can then be used by subsequent instructions. This stage always takes one cycle to perform. After this stage, any instruction leaves the pipeline and is considered to be finished.

Dependencies and hazards

Dependencies and hazards describe special situations which arise from certain instruction sequences. Not tackling these situations may cause the pipeline to lose or even compute false results. Not every hazard is present in each microarchitecture, the following is a general overview. Afterwards, techniques to handle these hazards are presented.

Data hazards A data hazard occurs when multiple instructions in close succession use the same register, without the register being written after the first instruction. There are three different types of dependencies, namely read-after-write, write-after-write and write-after-read. Whenever one instruction writes to a register, subsequent instructions need to use the updated value. Write-after-read dependencies are only hazards in certain microarchitectures, e.g. when using reordering of instructions. Here, the read-instruction must not use the updated value for the register.

Control hazards Control hazards describe situations in which an instruction branches, i.e. alters the program counter. If an instruction execution depends on a branch prior to it in program order, it must not take effect if the branch condition evaluates accordingly.

Structural hazards Structural hazards arise if multiple instructions use the same resource, e.g. the background memory. The pipe has to decide how to allocate the resource, such that both instructions execute correct.

Forwarding and stalling To handle data hazards, register results of the ALU are forwarded in our pipeline to a subsequent instruction without having to wait for them to be written back to the General purpose register (GPR). In case of a computation directly after an operand is loaded from memory, the pipeline stalls the execute stage until the operand is ready.

Branching and speculation When branching, the PC is set directly after the execute stage. In the instruction fetch stage however, up to two instructions have already been fetched after the branch instruction, which are not supposed to take effect if the branch is taken. The instruction fetch and instruction decode stage are thus flushed when a branch is taken. This behaviour of continuing execution despite an unresolved branch in the pipeline is called *speculation.* If the branch is not taken, the speculation is correct, and the processor performance is increased compared to no speculation. Speculation introduces complications to compositional analysis because the instruction fetches from memory have still taken place, although the instructions are not executed.

As we have seen, most stages here take one clock cycle to perform, given that no hazards are present. However, the latency of the execution stage depends on the instruction and sometimes even its operands. The second source of variance in latency is the memory, which is accessed in the fetch stage as well as the memory stage. How long it takes to resolve the memory access varies and depends on multiple factors, which are explained in a later section. Finally, instructions cannot proceed to a later stage in the pipeline if this stage is still occupied by another instruction. Since precise effects in the pipeline cannot be seen when considering instructions in isolation, determining the exact runtime for each instruction individually is not feasible in general.

3.2 Out-of-order Pipeline

The second microarchitecture features *dynamic scheduling*, which includes out-of-order execution. This means, that instructions are not guaranteed to execute in the order of the program, but may be rearranged dynamically in the pipeline. It allows to use capacity of functional units like integral ALUs more efficiently and in general increases the pipeline's performance. [HP12]

The out-of-order pipeline model used in this thesis is based on the Tomasulo algorithm, also described in [HP12]. While possibly rearranging instructions during execution, the Tomasulo algorithm always maintains the program order and commits the instructions in-order, i.e. writes changes to the register in program order. This ensures that instructions which are in the pipeline, but not meant to be executed (due to speculation, or interrupts), do not change the program state, i.e. register values.

In the following paragraphs, the components of the Tomasulo algorithm are explained briefly. Six actions are performed on every instruction. Each action involves one or more components, as shown in Table 3.1. The table indicates in which components the instruction is present after the described action.

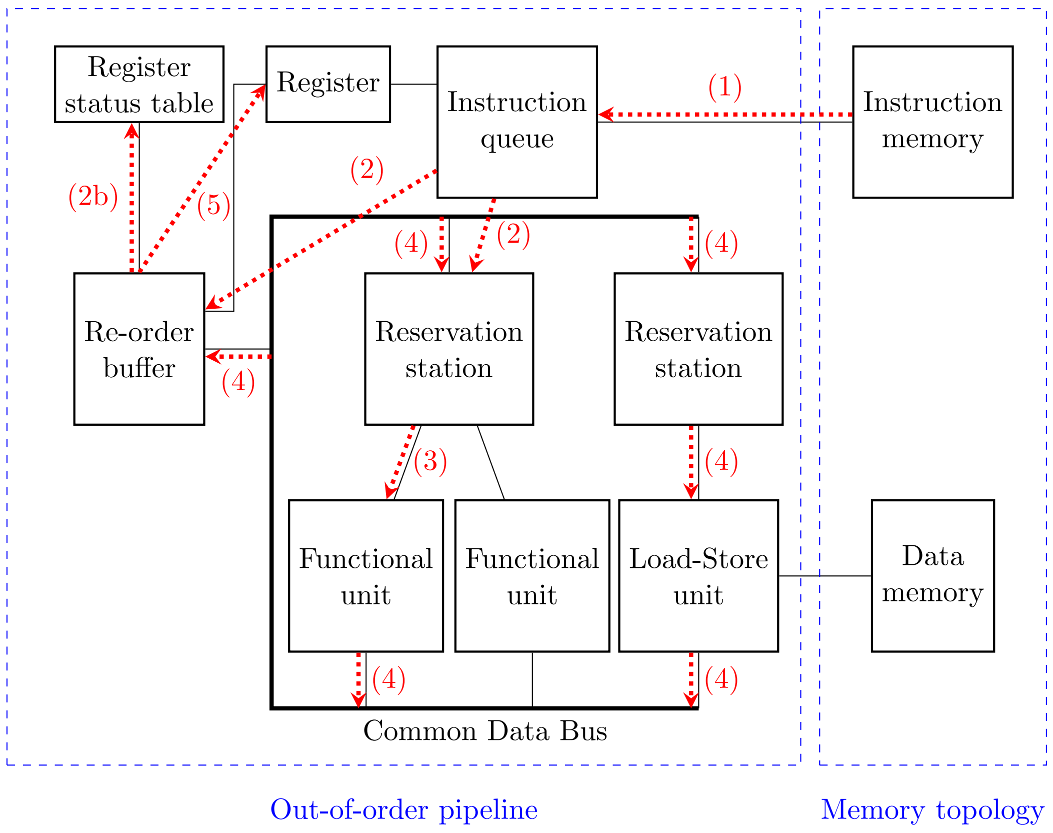


Figure 3.2: Schematic view of out-of-order pipeline

Instruction queue (IQ) The first component of our out-of-order pipeline is the instruction queue. The instruction queue contains fetched instructions from memory until they are issued to the pipeline. The purpose of this queue is to provide a buffer and hide memory latency. Issuing is possible when both the re-order buffer and a suitable reservation station, i.e. one that serves a functional unit which can perform this type of instruction, have a free position. Fetching instruction continues each cycle, resulting in a number of speculative accesses to the instruction memory after a branch.

Re-order buffer (ROB) Central component of Tomasulo's scheme is the re-order buffer. When an instruction is issued, the re-order buffer assigns a tag to the instruction. This tag is used by various components, e.g. to identify values on buses. When instructions finish, the result is captured here and held until the instruction commits[2]. Committing an instruction comprises writing the result back to the register file if needed and removing the instruction from the re-order buffer. This is done only

[2]Instead of commit, the term *complete* is also used in literature.

Action	Description	IQ	ROB	RS	FU	CDB	RST
(1)	fetch						
		o	-	-	-	-	-
(2)	issue						
		-	o	o	-	-	o
(3)	dispatch						
		-	o	-	o	-	o
(4)	execute						
		-	o	-	o	-	o
(5)	finish						
		-	o	-	-	o	o
(6)	commit						

Table 3.1: Out-of-order pipeline stages

if all previous instructions have committed. Tomasulo's algorithm therefore finishes all instructions in-order.

Register status table (RST) The register status table is an extension of the register file to store a ROB tag per register. In case an instruction writes to the register, its ROB tag is stored when the instruction is issued. When an instruction is issued and needs the value of a certain register, there are two scenarios: If some ROB tag is stored at a register, it corresponds to the most recent instruction writing this register. The value to be used can either be obtained from the ROB or by snooping the Common data bus (CDB). In case there is no ROB tag, the value in the register is up-to-date and can be used. The RST essentially solves all data hazards by *register renaming.*

Reservation station (RS) A reservation station buffers instructions in front of functional units and allows for rearrangement. Issued instructions are placed in a corresponding reservation station. All operands are fetched from the register, or the ROB tag is stored when the operand is not yet available. An instruction is dispatched when all operands are ready and the functional unit able to execute it. When an instruction is waiting for an operand, but a later instruction has all operands ready, the later instruction passes the former and both instructions execute out-of-order.

Functional unit (FU) Functional units execute instructions, which do not access the data memory. Known functional units include e.g. integer and floating-point arithmetic units. Execution latency of instructions might be either fixed or depend on the value of the operands, e.g. in a floating-point multiplication/division. For the latter case, a accurate value analysis might allow for a more efficient and more precise WCET analysis. In the case that values are not known, the analysis splits to consider all possible cases.

Load / Store unit The load / store unit is a special kind of Functional unit. It serves as a queue for all load and store instructions. Similar to the memory stage of the in-order pipeline, it is connected to the data memory. While re-ordering of data accesses is generally possible, this is excluded in our microarchitecture, since rearranging cache accesses increases interdependency between pipeline and cache. Thus load and store instructions still execute in-order.

Common data bus (CDB) CDBs are used to transfer the results from the functional units to the re-order buffer and the reservation stations. A common data bus consists of two parts: One contains the re-order buffer tag, the other one contains the result. Receiving components (ROB and reservation stations) compare the tag and use the result if needed.

Figure 3.2 is a schematic view of our microarchitecture. Two parts are shown: the out-of-order pipeline with its components, and a simplified version of the connected memory topology. The latter is described in detail in the Section 3.3. For now, we focus on the pipeline part. Most components explained earlier are shown in black rectangles, the only exception being the common data bus, which is represented by a thick line. In addition, red lines indicate the flow of an instruction through the pipeline. There are six stages an instruction passes, which are listed in Table 3.1.

The pipeline shown can be extended by using multiple common data buses (up to the extreme case of one CDB per functional unit), more reservation stations, larger instruction queue or re-order buffer, issuing more than one instruction per cycle or remove the restrictions on in-order execution of data accesses. However, the architecture is complex enough to evaluate the concept and limits of compositional analyses.

3.3 Memory Topologies

The memory topology describes all hardware components which contain or manage data, apart from the registers, and components that performs accesses to memory. This includes separate caches for instructions and data as well as a common background memory, e.g. DRAM or SRAM. While the background memory is large enough to hold every instruction and data the program might possibly need, the caches only contain a small subset of instructions or data at a time. Both caches use Least recently used (LRU) as their replacement policy. A schematic view of the memory topology is shown in Figure 3.3. The outer topology, indicated by blue dashes, provides two buses that are connected to the cache controllers. The inner topology again provides buses for both instruction and data accesses, however, they are both connected to one single background memory / Static random access memory (SRAM) with an arbiter to decide parallel accesses.

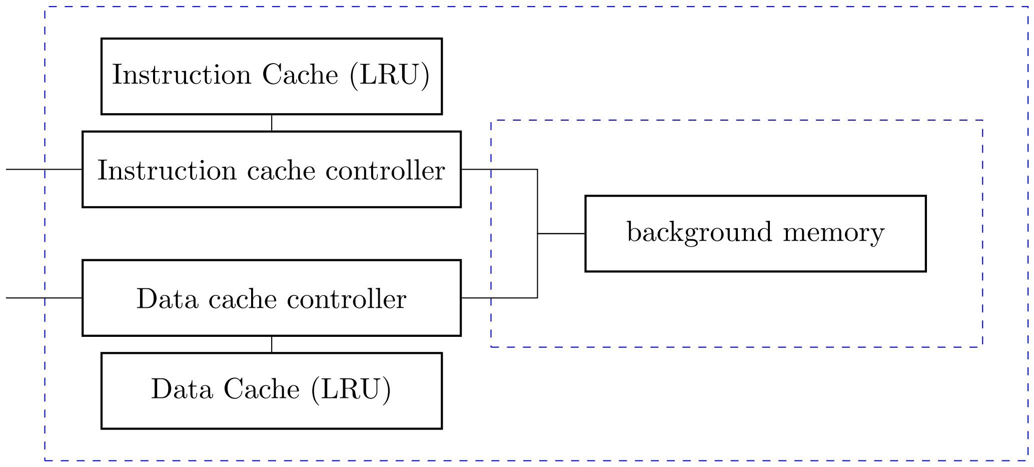


Figure 3.3: Schematic view of a memory topology with separate caches

The topology is connected to the pipelines as described in Figure 3.1 and 3.2, respectively, i.e. the instruction cache is connected to the instruction fetch stage for both pipelines, the data cache is connected to the memory stage or the load/store unit, respectively. Each connection consists of both an address and a data bus, and can be used for one access per cycle.

The cache controller checks whether content for the given address is present in the cache in one cycle. In case of a cache hit, the content is returned to the pipeline in the same cycle. In case the cache is not hit (cache miss), we need to access the background memory. Due to uncertainty it might not be possible to decide which case happened, so both possibilities are pursued by the analysis (*split*).

In case of a cache miss, the background memory contains the desired data. As shown in Figure 3.3, both caches are connected to this background memory, and one can block the access of the other, since the background memory allows only one access at a time. When both an instruction and data is requested at the same time, the arbiter data prioritises on the bus to the background memory. Prioritising instructions is often not beneficial since processing the instructions is blocked by the earlier data instruction waiting in the pipeline. The time to obtain data from the background memory is assumed to be fixed (SRAM). In the evaluation, we consider different settings for this parameter. In a real scenario, the latency depends on the type of memory. Cache misses might cause a significant share of the overall execution time. However, the effect of one cache miss might be *hidden* by preceding or subsequent instructions, e.g. a preceding instruction might take a long time to execute, overlapping with a miss when fetching an instruction. Then, not the entire additional time of a cache miss affects the execution time.

Finally, there are different modes on how a store operation is handled by the memory hierarchy. There are two modes with two settings each:

1. **Write-back / write-through** This mode defines when a store is written to main memory. When the write-back setting is enabled, a store is written to the cache, but not to the main memory. Instead, a dirty bit is set in the cache. When a dirty cache line is replaced, the content of the cache line is written back to main memory. In the case of write-through, a store is immediately written to the main memory. Therefore, if a cache line needs to be replaced, it can be done without further actions.

2. **Write-allocate / write-non-allocate** This mode decides what is done when an address of a store is currently not cached. In the write-allocate case this means that a store to an address not present causes a load to that address first, and afterwards a store into this loaded cache line. In write-non-allocate, the store is done without loading it into the cache, which causes a succeeding access to the same address to miss as well.

Notice that selecting write-non-allocate means that a store to a non-cached address requires a write-through to background memory, making a combination with write-back obsolete.

In the following chapters, we assume caches to use the write-through and write-allocate setting. The other settings are out of scope of this thesis.

4 Compositional Approach

The basic idea of the compositional approach is to decompose the entire system into components, analyse each component individually and combine the results of their analyses to an overall bound. The chapter follows the formal definition of timing compositionality (Equation 4.1) from [Hah14] and instantiates it for pipeline and caches.

Starting point is a *decomposition* of the system into a set of components I. A *configuration* c is defined as the pair of system state and the input it receives, thus determining the system's behaviour. For each component, a projection function $(p_i : C \mapsto C_i)_{i=1..n}$ defines which part of the system is relevant for the component. A family of functions $(tc_i : C_i \mapsto T)_{i=1..n}$ determines each components' timing contribution. Lastly, a combination operator $\bigoplus : T^n \mapsto T$ uses all individual timing contributions to obtain the overall timing bound. A decomposition is called *timing compositional*, if for each configuration $c \in C$ of the system, the obtained compositional timing bound is at least as large as the timing of the entire system $(tc : C \mapsto T)$.

$$\forall c \in C.\ tc(c) \leq \bigoplus_{i=1}^{n} tc_i(p_i(c)). \tag{4.1}$$

Definition of timing compositionality from [Hah14].

The compositional approach aims to tackle the *state space explosion problem* of the integrated approach. In both integrated and compositional approach, uncertainty may lead to splits in the analysis, causing the creation of a large number of states to be analysed. To understand why the compositional approach can simplify the analysis, consider the simple example in Figure 4.1. Red and green states produce a split when cycled, however the reason for the split once lies in one component, and once in the other. In the integrated approach, both types of split follow each other, resulting in a large number of states afterwards. In the compositional approach, red states' splits occur in one component, green states' splits in the other. These respective uncertainties do not influence the analysis of the other component, and thus do not *multiply*. However, there is one analysis needed per component. Regarding runtime and while disregarding the analysis result, the compositional approach is advantageous, if the reduced complexity saves more runtime than the time added by running the analysis for multiple components.

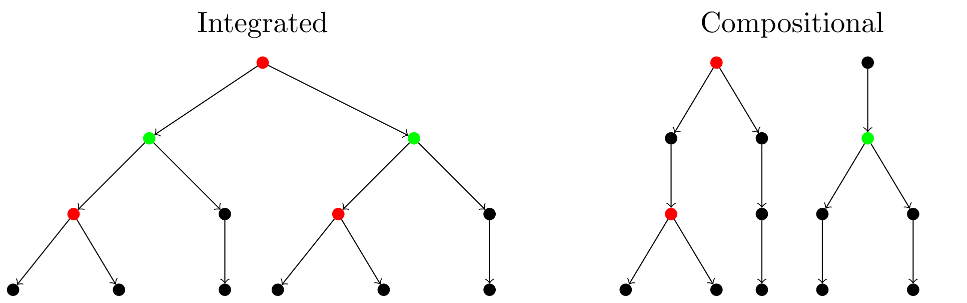


Figure 4.1: Consequences of uncertainty in both integrated and compositional approach

This seems conceptually rather simple, however in a real microarchitecture, different components are not fully independent of each other. Their (timing) behaviour depends on the other components. A good choice for a decomposition ensures that its components are sufficiently independent from each other. Since soundness is a requirement for WCET analysis, every dependency needs to be tackled by a conservative approximation of the other components' behaviour in the analysis.

4.1 Instances

While there are different proposals for possible or efficient decompositions, this thesis focuses on compositionality regarding pipeline and caches in the microarchitecture. Specifically, three different decompositions are considered.

Pipeline and data cache

This first decomposition separates the data cache from the rest of the system. Whether an access to the data cache is a hit or a miss no longer affects the analysis of the pipeline and other parts of the system. Data addresses are often hard to determine statically, which makes the content of the cache hard to predict, and this introduces uncertainty into the cache analysis.

Pipeline and instruction cache

In the second decomposition, only the instruction cache is separated. While instruction addresses are known and precise, joining and speculation still cause the cache to be imprecise. Also, there are instruction accesses in almost every cycle, possibly resulting in many splits.

Pipeline, data cache, and instruction cache

This decomposition consists of three components, each contributing to the total bound. This combines both of the other decompositions. As we will see later, this might be more precise than separating only one cache.

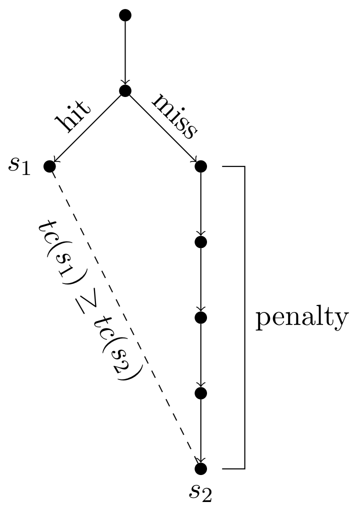


Figure 4.2: How cache misses are handled when analysing compositionally, based on [Hah14, Fig. 14b]

The reason it is possible to decompose the microarchitecture into the variants given above is depicted in Figure 4.2. The condition required for compositionality is: $tc(s_1) \leq tc(s_2)$. This means, for any program point, the configuration after the resolved miss must not lead to longer execution time (timing contribution) than the configuration after resolving the hit cache. In this case, we can just assume a cache hit, save the split, and add the time it takes to resolve the miss separately.

In addition to the advantage of having less splits in the compositional approach, there are new opportunities to include information in the analysis. Cumulative information, e.g. the maximal number of cache misses of a basic block, can be easily included by lowering the number of cache misses for the basic block. In the integrated approach, new constraint are needed to exclude all paths that have more misses than the maximum. Since there may be a large number of paths, the complexity of adding these constraints is not constant.

In the following section, it will be shown how to determine the timing contribution for the pipeline analysis and the number of cache misses for the cache analysis.

4.2 Contributions

Pipeline analysis The cache considered compositionally is replaced by an *always-hit* cache, i.e. a (conceptual) cache containing all data being requested. The analysis is then run in the same manner as explained in Chapter 2 in the integrated approach. During cycling, the pipeline assumes that accesses to the cache are resolved in the shortest possible time. This defines the *timing contribution* of the pipeline.

Cache analysis The timing contribution of the second component is determined by analysing the cache on its own. Again, the same cycle model as for integrated analysis is used, states however consist only of the current state of the cache, and the position in the program. Object of investigation is no longer the runtime, but the number of cache misses. The analysis extracts the order of accesses from the program, and provides this order to a cache analysis, which is already studied thoroughly, e.g. in [AFMW96] and [Gru12]. Extracting the access order is simple for the data cache, since all accesses occur in program order.

Considering the instruction cache is more complex. When branching, additional cache accesses occur after accessing the branch instruction, and before the branch is resolved.

In the in-order pipeline, the branch instruction needs to be decoded and executed before the new PC is set. This leads to a minimum of one access to the instruction cache, and a maximum of two accesses, one being fetched, and one being decoded before flushed when branching. This speculation needs to be taken into account for all branches of the program for the cache analysis.

For the out-of-order pipeline, it is even more complex, since the branch might need to wait some time before getting executed, giving the instruction queue time to fill up and even issue instructions to the pipeline. A trivial bound on the number of accesses is given by equation 4.2, which assumes the pipeline to be fully occupied. Applying the equation for an actual analysis is suspected to give very imprecise results and thus a high WCET bound, so it was not considered in the evaluation.

$$0 \leq \#\mathit{Accesses} \leq |IQ| + |ROB| - 1 \tag{4.2}$$

4.3 Penalty Computing and Combining

Pipeline and cache analysis deliver execution time and number of cache misses, respectively. To combine them, we need to convert the number of cache misses to a contribution to the execution time. From the microarchitecture, we know the time to resolve an access to the background memory, which defines the *direct effect* of a cache miss. Additionally, an *indirect effect* contributes to the execution time as well.

We distinguish the overall penalty for the different decompositions:

Data cache compositional The penalty for all data misses is determined by Equation 4.3. The first part consists of the direct effect of all data misses. The second part refers to the indirect effect due to possible blocking caused by an instruction cache miss. To understand the indirect effect, consider Figure 4.3. A data cache access in the beginning can be either a hit or a miss. When the data cache miss occurs, the access to the background memory might be blocked by

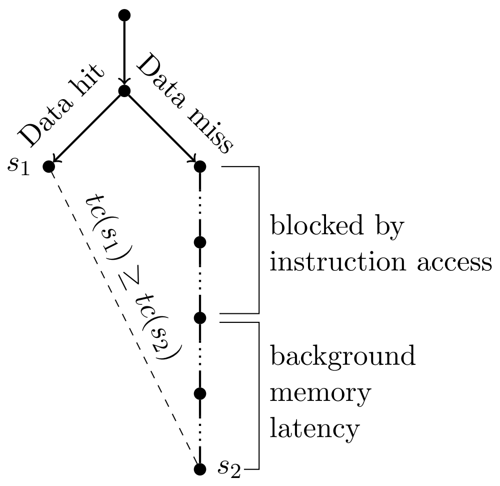


Figure 4.3: Background memory latency is not sufficient as penalty

an ongoing instruction access. To reach state s_2, where the data miss is resolved and thus the remaining execution time is comparable, we need to account for the time the access can be blocked, which defines the indirect effect. This satisfies the condition required for compositionality, $tc(s_1) \geq tc(s_2)$ (cf. Figure 4.2). It is worth noting that each data access cannot be blocked more than once by an instruction access, since data accesses are prioritized.

Data accesses cannot be blocked more often than instruction misses occur, as only these cause bus accesses. This explains the minimum operation in Equation 4.3. In order to use this, the number of instruction accesses has to be provided by the integrated analysis.

$$\begin{aligned} \mathit{DataContrib} = {} & \#\mathit{DataMisses} * \mathit{BgLat} \\ & + \mathit{min}\,(\#\mathit{DataMisses}, \#\mathit{InstrMisses}) * (\mathit{BgLat} - 1) \end{aligned} \tag{4.3}$$

In case stores are handled in an unblocked manner in the microarchitecture, a data access can be blocked by a store. The contribution needs to account for this indirect effect:

$$\begin{aligned} \mathit{DataContrib} = {} & \#\mathit{DataMisses} * \mathit{BgLat} \\ & + \mathit{min}\,(\#\mathit{DataMisses}, \#\mathit{InstrMisses} + \#\mathit{Stores}) * (\mathit{BgLat} - 1) \end{aligned} \tag{4.4}$$

Instruction cache compositional The contribution of the instruction cache consists of similar parts as the data cache contribution above. Again, the equation first accounts for the direct effect, i.e. the time it takes to resolve the accesses from the background memory. Secondly, the indirect effect is considered, in case an instruction access is blocked by a data miss happening in parallel. The data access is prioritized, so it is not required that the access is already ongoing. Still, an access can only be blocked by one data access, since our pipeline does not allow for two data misses in two subsequent cycles.

In contrast to an instruction access, a data access cannot only load from memory, but also store. Assuming the write-through policy is used, as it is in this thesis, each store causes an access to the background memory. The number of data cache misses containing load accesses as well as the number of stores, and is determined by the integrated analysis.

As in the previous case, we cannot be blocked more often than either instruction misses or data misses plus stores, and thus use the minimum, resulting in Equation 4.5.

$$\begin{aligned} \mathit{InstrContrib} = {} & \#\mathit{InstrMisses} * \mathit{BgLat} \\ & + \mathit{min}\,(\#\mathit{InstrMisses}, \#\mathit{DataMisses} + \#\mathit{Stores}) * \mathit{BgLat} \end{aligned} \tag{4.5}$$

Both caches compositional When both caches are considered compositionally, we do not need to account for indirect effects as described in the first paragraph, since both cache analyses account for their accesses, which contains the indirect effects of the other cache. However, the instruction cache still needs to account for the possibility of being blocked by stores to the background memory. This simplifies the equations for data cache timing contribution (Equation 4.6) and instruction cache timing contribution (Equation 4.7).

$$\mathit{DataContrib} = \#\mathit{DataMisses} * \mathit{BgLat} \tag{4.6}$$

$$\begin{aligned} \mathit{InstrContrib} = {} & \#\mathit{InstrMisses} * \mathit{BgLat} \\ & + \mathit{min}\,(\#\mathit{InstrMisses}, \#\mathit{Stores}) * \mathit{BgLat} \end{aligned} \tag{4.7}$$

In case of unblocked stores, we need to alter the data contribution analogously to Equation 4.4:

$$\begin{aligned} \mathit{DataContrib} = {} & \#\mathit{DataMisses} * \mathit{BgLat} \\ & + \mathit{min}\,(\#\mathit{DataMisses}, \#\mathit{Stores}) * \mathit{BgLat} \end{aligned} \tag{4.8}$$

Combining results To conclude the calculation of the WCET in the compositional approach, only the combine operator is missing. In our case, combining is done by adding the results of pipeline and cache contributions, as formulated in Equation 4.9.

$$\mathit{WCETbound} = \mathit{PipelineContrib} + \mathit{DataContrib} + \mathit{InstrContrib} \tag{4.9}$$

4.4 Refining penalties

In the last section, we assumed that we need to account for all the time being blocked as indirect effect, because it defines the time the pipeline is prevented from advancing due to blocking. The in-order pipeline described in Chapter 3 however, can not advance indefinitely, but is limited, since the instruction accessing the memory prevents further operations in the pipeline. Figure 4.4 depicts this situation.

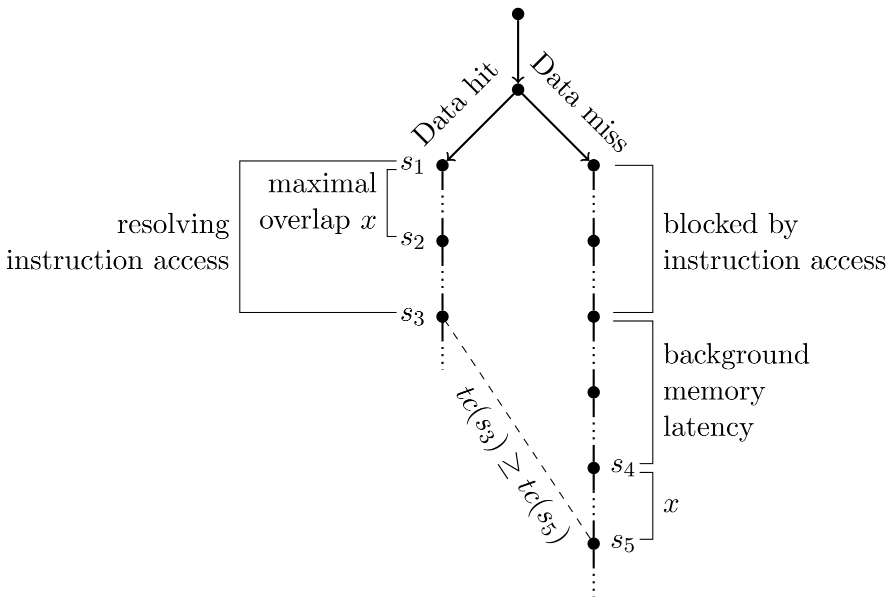


Figure 4.4: Refinement of penalty according to maximal overlap

We assume that the maximal time that the pipeline can advance is represented by x, the *maximal overlap*, depicted in the figure between states s_1 and s_2. After s_2, the pipeline does not advance any more, until the instruction access is resolved. This can be used to refine the indirect effect. Instead of using the relation as before, $tc(s_4) \leq tc(s_1)$, we add the maximal overlap x to the background latency. This allows the pipeline to advance to a state comparable to s_2, i.e. a state which does not lead to a higher execution time. We get $tc(s_5) \leq tc(s_2)$. From s_2 to s_3, the microarchitecture only resolves the instruction access. This is already done in s_5, therefore the execution time after s_3 can not be less than the execution time after s_5, and we get $tc(s_3) \geq tc(s_5)$.

The idea described in Figure 4.4 can also be applied to instruction misses, only the maximal overlap changes, which is also dependent of the microarchitecture.

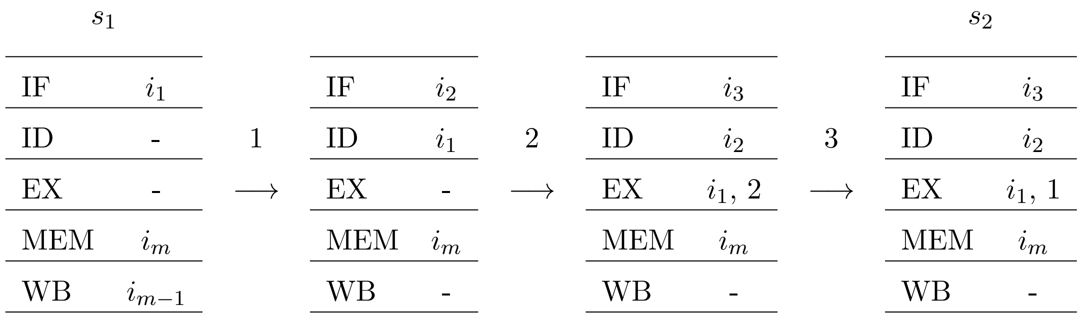


Figure 4.5: Worst-case scenario for an instruction access getting blocked

Determining the maximal overlap For the in-order pipeline, we determine the maximal overlap x for both instruction and data misses, when blocked by an access to the other cache. This maximal overlap is individual for a microarchitecture, and we argue about worst-case scenarios, i.e. a series of pipeline states as long as possible. These overlaps are strongly conjectured, a formal proof is out-of-scope for this thesis. The evaluation will thus also contain a section which uses non-refined contributions.

Instruction access Given an instruction access to the background memory, which is blocked by a data access to the background memory, we want to find the sequence of pipeline states with maximal overlap. Since the data access is ongoing, we know there is an instruction currently in the memory stage of the pipeline. As shown in Figure 4.5, the first state shows the state after the instruction access to i_1, which in the case analysed is a cache hit. In the first cycle, i_1 is decoded, and i_2 is fetched from the cache. Additionally, an instruction before i_m, i_{m-1} will be written back and leaves the pipeline. In the second cycle, the pipeline starts to execute i_1.[1] i_2 is decoded, and a third instruction i_3 is fetched. In the third cycle, i_1 finishes executing and could advance to the memory stage occupied by i_m in the next cycle. i_2 and i_3 can not advance, since the next stages are still occupied. Therefore, there cannot be more overlap after three cycles.

Data access In the second scenario, a data access to the background memory is blocked by an instruction access. Figure 4.6 uses i_1 as the instruction getting blocked when accessing the data memory. The blocking instruction is currently fetched and named i_f. Using the same cycling as before, we obtain the shown sequence. After six cycles, no further progress in the pipeline is possible.

Adjusting the contributions Using the maximal overlap in the formulas established in Section 4.3, we can refine the penalties for the indirect effect. The refinement can not be applied when the microarchitecture uses unblocked stores. Since the instruction

[1]In the in-order pipeline used for evaluation, no instruction takes more than two cycles to execute.

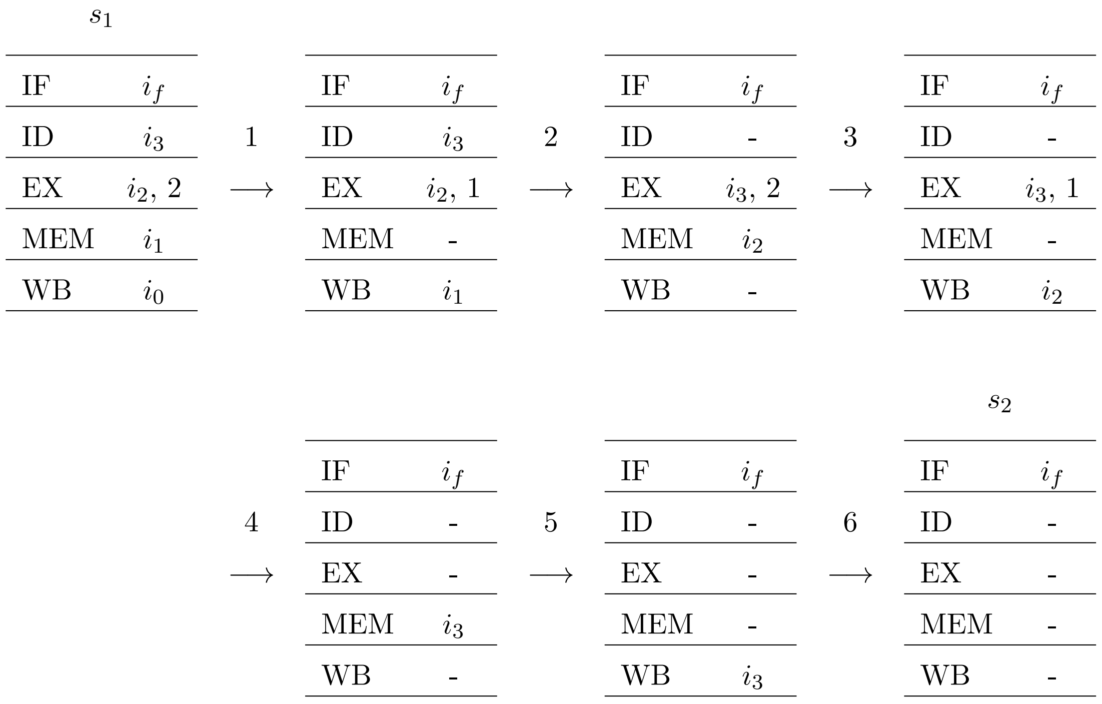


Figure 4.6: Worst-case scenario for a data access getting blocked

causing the store is not kept in the pipeline, the overlap can be larger, and we need to account for the maximal blocking time. We will distinguish equations for both blocked and unblocked stores.

Data cache compositional Since the indirect effect of a data miss is limited to six cycles, we change Equation 4.3 to:

$$\begin{aligned} \mathit{DataContrib} &= \#\mathit{DataMisses} * \mathit{BgLat} \\ &\quad + \mathit{min}\left(\#\mathit{DataMisses}, \#\mathit{InstrMisses}\right) * 6 \end{aligned} \tag{4.10}$$

Again, using unblocked stores introduces the possibility of being blocked by a store. Any data misses not able to be blocked by a store can still be blocked by an instruction access.

$$\begin{aligned} \mathit{DataContrib} &= \#\mathit{DataMisses} * \mathit{BgLat} \\ &\quad + \mathit{min}\left(\#\mathit{DataMisses}, \#\mathit{Stores}\right) * \mathit{BgLat} \\ &\quad + \mathit{min}\left(\mathit{max}\left(0, \#\mathit{DataMisses} - \#\mathit{Stores}\right), \#\mathit{InstrMisses}\right) * 6 \end{aligned} \tag{4.11}$$

Instruction cache compositional The indirect effect of an instruction access being blocked was limited to three cycles, Equation 4.5 can therefore be refined to:

$$\begin{aligned} \mathit{InstrContrib} &= \#\mathit{InstrMisses} * \mathit{BgLat} \\ &\quad + \mathit{min}\left(\#\mathit{InstrMisses}, \#\mathit{DataMisses} + \#\mathit{Stores}\right) * 3 \end{aligned} \tag{4.12}$$

Since an instruction access to the background memory blocked by an unblocked store does not benefit from the reduced penalty, we slightly adjust Equation 4.12 for this setting, analogously to Equation 4.11:

$$\begin{aligned} \mathit{InstrContrib} &= \#\mathit{InstrMisses} * \mathit{BgLat} \\ &\quad + \mathit{min}\left(\#\mathit{InstrMisses}, \#\mathit{Stores}\right) * \mathit{BgLat} \\ &\quad + \mathit{min}\left(\mathit{max}\left(0, \#\mathit{InstrMisses} - \#\mathit{Stores}\right), \#\mathit{DataMisses}\right) * 3 \end{aligned} \tag{4.13}$$

Both caches compositional Finally, considering both caches compositionally, the data contribution did not need to account for indirect effects, cf. Equation 4.6. We can improve on the instruction cache contribution, changing Equation 4.7 to:

$$\begin{aligned} \mathit{InstrContrib} &= \#\mathit{InstrMisses} * \mathit{BgLat} \\ &\quad + \mathit{min}\left(\#\mathit{InstrMisses}, \#\mathit{Stores}\right) * 3 \end{aligned} \tag{4.14}$$

In case unblocked stores are used, it is not possible to refine the equations, as discussed in previous paragraphs.

This concludes the chapter about compositionality. We have seen how the compositional analysis differs from the integrated, and which instances we consider in this thesis. We also described how analysing the components lead to different contributions, and how cache miss contribution can be made as precise as possible.

5 Evaluation

This chapter contains the results of evaluating both the integrated and compositional approach as described in the previous chapters. After a short introduction to the methodology of the evaluation, we consider a standard setup first, which serves as a reference for the evaluation. Afterwards, different microarchitectural features and parameters like the latency of background memory or cache sizes are explored to measure their impact on both approaches.

5.1 Evaluation setup

Benchmark setup Results are obtained by analysing the timing for two sets of programs. The first set contains 31 benchmarks from Mälardalen WCET research group [GBEL10]. The second set consists of seven control programs generated with the SCADE®[1] suite. Tables and plots in this thesis contain only a subset of the test cases due to clarity. The test cases were chosen with respect to the variety of programs and the obtained results. However, average values contain all 38 test cases.

Standard setup For each evaluation, all test cases are run using the same set of options. These options include the choice of microarchitecture, memory hierarchy and latency of background memories. The standard setup uses the in-order pipeline, two separate caches for instructions and data and a common background memory with a fixed latency of 10 cycles. Both caches have 32 cache sets, are two-way associative and have a line size of 16 bytes, resulting in a total size of 1 KiB. Caches use write-through and write-allocate policies. Store instructions are blocked in the pipeline until the store is completely finished.

These settings will be adjusted individually in each section to evaluate the impact of this setting or microarchitectural feature on both analysis approaches.

Tables and plots Evaluation results are presented in both tables and plots. Tables are structured in the following way: The first column contains the name of the test case. For evaluation of the in-order pipeline, the remaining four columns contain measurements of the four analysis options: integrated analysis, compositional data

[1]http://www.esterel-technologies.com/products/scade-suite/

cache, compositional instruction cache, and finally fully compositional analysis. For each table cell, the obtained WCET bound is displayed in the first row, the analysis time in the second row, and the maximal used memory (peak memory) during analysis in the third and last row.

Plots work in the same manner. Each bar represents one option: First the integrated approach, then data cache compositional, instruction cache compositional and finally fully compositional. All bars are normalised with respect to the integrated analysis, resulting in the first bar always ending at 1. They are grouped by each test case. The last group represents the geometric mean over all test cases. For plots regarding analysis time, each bar is divided to show the portions of the following analysis parts:

Preprocessing consists of computing value analysis and address analysis as well as calculating information about loops and control-flow from the LLVM intermediate representation. Additionally, all overhead not included somewhere else is included here, except for the compilation of the program.

Timing MuArch Analysis describes the microarchitectural analysis concerned with the pipeline. In case of the integrated analysis, the caches are contained in this part as well. In the compositional cases, there are separate microarchitectural analyses for the caches.

Timing Path Analysis builds the state-sensitive microarchitectural graph described in Section 2.3, using the calculated fixed point microarchitectural state for each basic block. Then, the graph is converted to an ILP and solved using CPLEX, a standard ILP solver.

Instruction / Data Cache MuArch Analysis describes the cache analysis of each basic block for the instruction or data cache.

Instruction / Data Cache Path Analysis uses the results from the instruction or data cache microarchitectural analysis and provides the maximal number of cache misses, analogously to the Timing Path Analysis.

Methodology To ensure that the results obtained by running the analyser are reliable, each test case for each option was executed at least five times on one core of a 3.3 GHz processor with 20 GiB RAM. The standard deviation for time and memory did generally not exceed one percent. Since displaying it reduced the readability of the plots, but did not influence further conclusions, they are omitted in the following.

Significance of results The results presented in this chapter were obtained carefully. However, some restrictions are inherent. Relative WCET bounds always refer to the best possible bound determined by a specific technique, not to the WCET itself, since it is unknown (cf. Figure 1.1). Any results thus also depend on the overestimation of the best possible bound compared to the actual (unknown) WCET.

When summarizing results of many test cases, the geometric mean is used instead of the arithmetic mean to limit the impact of test cases with different length.[2] However, the choice of test cases still impacts the results, e.g. in terms of the relation between instruction and data cache accesses. Since there is no "typical" real-time program, one has to be careful to transfer the results to another scenario.

5.2 Compositional approach in the standard setting

Motivation The first evaluation is meant to provide an overview on the precision and resource usage of the compositional approach. The second important reason is to obtain a reference for comparison. This first evaluation is run with the standard settings, i.e. the in-order pipeline, two separate caches of size 1 KiB for instructions and data and a common background memory with fixed latency of 10 cycles. The caches use write-through and write-allocate policy. Stores to the background memory are blocking, i.e. the instruction is kept until the store is fully resolved.

Results

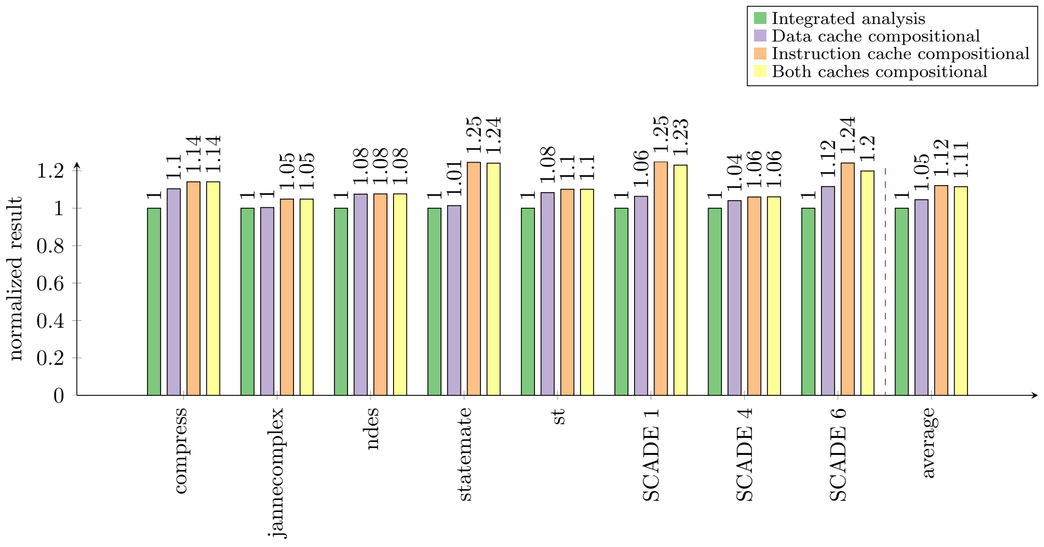


Figure 5.1: Relative WCET bound compared to the integrated approach in the standard setting

[2]In case nothing specific is mentioned, the terms 'mean' and 'average' always refer to the geometric mean in this thesis.

	Integrated	**Data cache compositional**	**Instruction cache compositional**	**Both caches compositional**
compress	$\begin{pmatrix}360,194 \text{ cyc} \\ 6.02 \text{ s} \\ \pm 0.02 \text{ s} \\ 94,687 \text{ KB} \\ \pm 675 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}397,592 \text{ cyc} \\ 5.15 \text{ s} \\ \pm 0.03 \text{ s} \\ 91,396 \text{ KB} \\ \pm 418 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}410,948 \text{ cyc} \\ 3.30 \text{ s} \\ \pm 0.02 \text{ s} \\ 80,099 \text{ KB} \\ \pm 539 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}410,948 \text{ cyc} \\ 3.57 \text{ s} \\ \pm 0.04 \text{ s} \\ 78,628 \text{ KB} \\ \pm 673 \text{ KB}\end{pmatrix}$
janne_complex	$\begin{pmatrix}19,429 \text{ cyc} \\ 0.41 \text{ s} \\ \pm 0.00 \text{ s} \\ 60,488 \text{ KB} \\ \pm 582 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}19,500 \text{ cyc} \\ 0.54 \text{ s} \\ \pm 0.00 \text{ s} \\ 59,590 \text{ KB} \\ \pm 510 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}20,377 \text{ cyc} \\ 0.45 \text{ s} \\ \pm 0.00 \text{ s} \\ 58,945 \text{ KB} \\ \pm 1,034 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}20,377 \text{ cyc} \\ 0.54 \text{ s} \\ \pm 0.00 \text{ s} \\ 58,703 \text{ KB} \\ \pm 448 \text{ KB}\end{pmatrix}$
ndes	$\begin{pmatrix}733,726 \text{ cyc} \\ 12.46 \text{ s} \\ \pm 0.12 \text{ s} \\ 106,239 \text{ KB} \\ \pm 493 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}788,960 \text{ cyc} \\ 9.28 \text{ s} \\ \pm 0.05 \text{ s} \\ 101,397 \text{ KB} \\ \pm 608 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}789,870 \text{ cyc} \\ 4.56 \text{ s} \\ \pm 0.03 \text{ s} \\ 83,712 \text{ KB} \\ \pm 448 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}789,870 \text{ cyc} \\ 4.89 \text{ s} \\ \pm 0.03 \text{ s} \\ 80,979 \text{ KB} \\ \pm 585 \text{ KB}\end{pmatrix}$
statemate	$\begin{pmatrix}17,884 \text{ cyc} \\ 18.44 \text{ s} \\ \pm 0.18 \text{ s} \\ 137,164 \text{ KB} \\ \pm 626 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}18,134 \text{ cyc} \\ 18.36 \text{ s} \\ \pm 0.18 \text{ s} \\ 133,585 \text{ KB} \\ \pm 452 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}22,266 \text{ cyc} \\ 11.20 \text{ s} \\ \pm 0.06 \text{ s} \\ 118,681 \text{ KB} \\ \pm 666 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}22,188 \text{ cyc} \\ 12.20 \text{ s} \\ \pm 0.07 \text{ s} \\ 116,307 \text{ KB} \\ \pm 594 \text{ KB}\end{pmatrix}$
st	$\begin{pmatrix}2,084,714 \text{ cyc} \\ 1.96 \text{ s} \\ \pm 0.01 \text{ s} \\ 74,736 \text{ KB} \\ \pm 1,446 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}2,258,277 \text{ cyc} \\ 1.68 \text{ s} \\ \pm 0.03 \text{ s} \\ 72,415 \text{ KB} \\ \pm 495 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}2,295,314 \text{ cyc} \\ 1.31 \text{ s} \\ \pm 0.01 \text{ s} \\ 68,263 \text{ KB} \\ \pm 971 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}2,295,314 \text{ cyc} \\ 1.41 \text{ s} \\ \pm 0.01 \text{ s} \\ 65,422 \text{ KB} \\ \pm 677 \text{ KB}\end{pmatrix}$
SCADE 1	$\begin{pmatrix}160,064 \text{ cyc} \\ 1.96 \text{ s} \\ \pm 0.01 \text{ s} \\ 79,997 \text{ KB} \\ \pm 530 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}170,214 \text{ cyc} \\ 2.00 \text{ s} \\ \pm 0.01 \text{ s} \\ 77,996 \text{ KB} \\ \pm 654 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}199,840 \text{ cyc} \\ 1.39 \text{ s} \\ \pm 0.01 \text{ s} \\ 72,750 \text{ KB} \\ \pm 515 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}196,924 \text{ cyc} \\ 1.59 \text{ s} \\ \pm 0.02 \text{ s} \\ 71,428 \text{ KB} \\ \pm 598 \text{ KB}\end{pmatrix}$
SCADE 4	$\begin{pmatrix}6,704,560 \text{ cyc} \\ 17.58 \text{ s} \\ \pm 0.42 \text{ s} \\ 156,674 \text{ KB} \\ \pm 527 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}6,975,441 \text{ cyc} \\ 12.43 \text{ s} \\ \pm 0.07 \text{ s} \\ 142,939 \text{ KB} \\ \pm 581 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}7,106,589 \text{ cyc} \\ 10.06 \text{ s} \\ \pm 0.10 \text{ s} \\ 128,016 \text{ KB} \\ \pm 564 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}7,111,569 \text{ cyc} \\ 10.65 \text{ s} \\ \pm 0.03 \text{ s} \\ 120,021 \text{ KB} \\ \pm 657 \text{ KB}\end{pmatrix}$
SCADE 6	$\begin{pmatrix}371,321 \text{ cyc} \\ 10.18 \text{ s} \\ \pm 0.05 \text{ s} \\ 129,996 \text{ KB} \\ \pm 546 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}414,327 \text{ cyc} \\ 9.18 \text{ s} \\ \pm 0.04 \text{ s} \\ 121,790 \text{ KB} \\ \pm 690 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}461,080 \text{ cyc} \\ 6.96 \text{ s} \\ \pm 0.03 \text{ s} \\ 107,641 \text{ KB} \\ \pm 824 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}445,174 \text{ cyc} \\ 7.53 \text{ s} \\ \pm 0.04 \text{ s} \\ 103,288 \text{ KB} \\ \pm 452 \text{ KB}\end{pmatrix}$
Arithmetic mean	$\begin{pmatrix}939,782 \text{ cyc} \\ 5.33 \text{ s} \\ 94,864.9 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}985,697 \text{ cyc} \\ 4.63 \text{ s} \\ 90,456.4 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}1,031,558 \text{ cyc} \\ 3.54 \text{ s} \\ 83,414.6 \text{ KB}\end{pmatrix}$	$\begin{pmatrix}1,030,278 \text{ cyc} \\ 3.88 \text{ s} \\ 80,943.6 \text{ KB}\end{pmatrix}$
Geometric mean	$\begin{pmatrix}100.0 \% \\ 100.0 \% \\ 100.0 \%\end{pmatrix}$	$\begin{pmatrix}105.129 \% \\ 95.686 \% \\ 96.522 \%\end{pmatrix}$	$\begin{pmatrix}112.145 \% \\ 73.208 \% \\ 89.707 \%\end{pmatrix}$	$\begin{pmatrix}111.708 \% \\ 81.725 \% \\ 87.518 \%\end{pmatrix}$

Table 5.2: Absolute results for both integrated and compositional approach in the standard setting

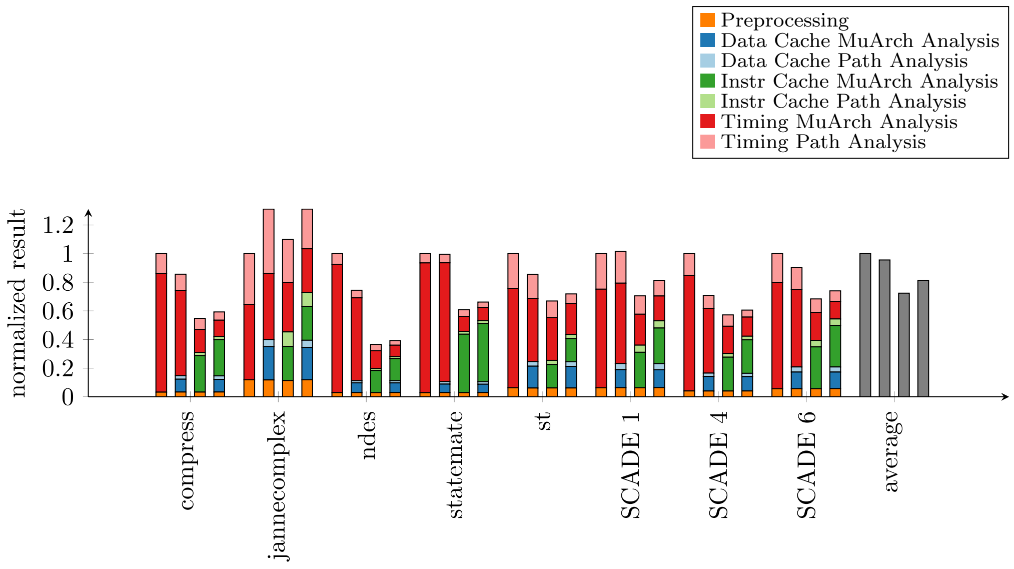


Figure 5.2: Relative analysis time in the standard setting

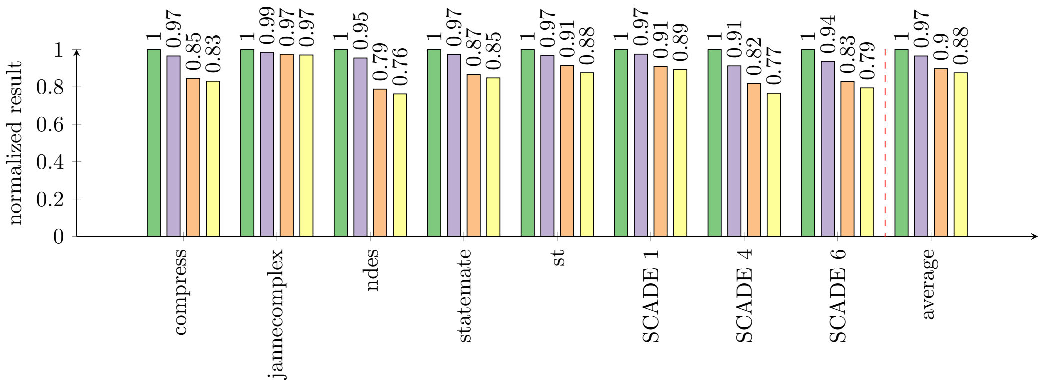


Figure 5.3: Relative analysis peak memory usage in the standard setting

Observations For the standard setting, the compositional approach introduces slightly less precise bounds for the compositional options (Figure 5.1). This is consistent over all test cases, although they differ in terms of intensity. Regarding the individual options, the increase is highest when analysing the *instruction cache* in a compositional manner. The increase in bound for the data cache compositional analysis is moderate with 5.1%. When looking at the test cases in detail, *janne_complex* and *statemate* have almost no increase in the compositional data cache option. On the other hand, only statemate and two of the SCADE tests have a high increase when compositionally analysing the instruction cache. For the fully compositional analysis, the bound is generally comparable to the instruction cache compositional analysis, and on average just a little bit more precise.

The second plot, Figure 5.2, presents the analysis runtime of the same experiments.

On average, time is saved in comparison to the integrated approach in the first column. For the data compositional option, this accounts for 4.3% of the integrated analysis time (cf. Table 5.2). When analysing the instruction cache compositionally, the saved time increases to 26.8% on average.

The time used by the analyser however can get worse depending on the test case executed. The second Mälardalen test case *janne_complex* stands out, since the analysis time increases. Also, preprocessing has the highest share of analysis runtime. The second outlier is *ndes*, with an exceptionally high decrease.

In Figure 5.15, the maximal or peak memory usage is presented. The peak memory decreased in the compositional approach by 3.5 - 12.5%. While decreasing memory usage is generally an advantage, the decrease is not large enough to allow any changes in the memory provided to the analyser. In the following, plots for peak memory are therefore mostly omitted.

For this setting, Table 5.2 contains all absolute results, based on ten runs of the analyser. For time and memory usage, the standard deviation is given and accounts only once for more then 1.5%, in the integrated option of the fourth SCADE test. As one would expect, the upper bounds on execution times do not change from one run to another.

Conclusions The first evaluation of the compositional approach results in consistently less precise bounds for all test cases. This is expected, since the compositional analysis does not account for overlaps of the pipeline and accesses to the background memory. Therefore, the time for resolving cache misses (which occur in both approaches) is fully reflected in the compositional bounds, and less so in the integrated bounds.

While the overestimation of the bounds is still reasonably small, the saved resources do not justify using the compositional approach. Precise bounds are more important than analysis time, and should be prioritized, if the new analysis time is still in the same order of magnitude.

5.3 On the maximal number of cache misses

When analysing a cache compositionally, the maximal number of cache misses is determined. During the evaluation, it became apparent that the number used differs from the number of cache misses in the integrated analysis. Further investigation yielded different causes for this, explained in this section.

Maximal misses in integrated analysis

In integrated analysis, cache misses are not object to maximization in the path analysis, instead, execution time is. Therefore, a worst-case path resulting from the analysis might not be a path with the maximal number of misses.

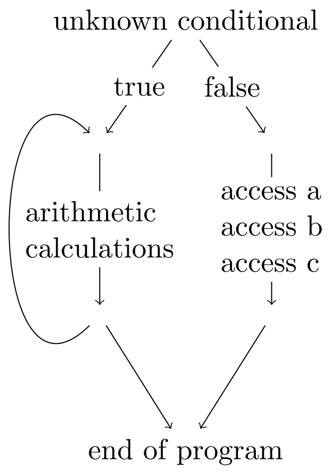


Figure 5.4: Example CFG for different number of misses

Figure 5.4 shows a program where the maximal number of misses differs from the maximal number of misses on a worst-case execution time path. Assuming the loop has a high iteration count, it is easy to see that the if-path takes longer to execute than the else-path. Considering the data cache compositionally first, the maximal number of misses equals the number of cache misses in the else-path, while the then-path contains no access. Only one of the paths can occur in a program execution. However, the compositional approach does not distinguish this, and adds the penalty for a cache miss to the worst-case timing path (the then-path).

The example still illustrates the effect when considering the instruction cache. The then-path contains instruction accesses, but if the loop is small, every access misses only in the first iteration, leading to a small, constant number of cache misses, while having a high execution time (bound). However, if the else-path is long enough, it produces enough accesses and misses to dominate the misses on the then-path. This gap can be arbitrarily large if the loop iteration count increases.

Maximal number of data cache misses in compositional analysis

The second cause concerns the compositional analysis itself. In the example depicted in Figure 5.5, we have a conditional access to some cache block a which is unconditionally accessed again directly after. The integrated analysis considers both branches with and without the first access and does not join the states, since the microarchitectural

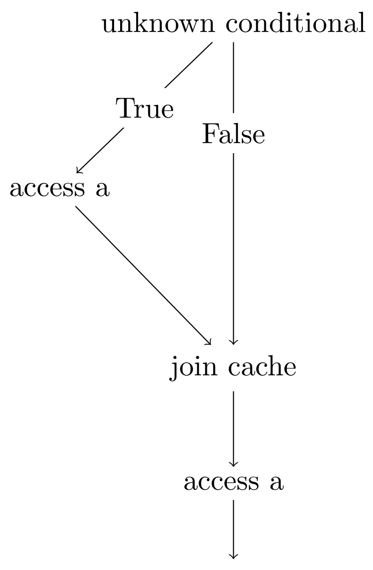


Figure 5.5: Example CFG where joining in cache analysis lead to imprecise results

states can be different due to different pipeline states. It finishes the analysis of this program segment and determines one as the number of maximal cache misses.

The compositional approach does not consider the cache in its pipeline analysis, and produces again two timing bounds for the basic block. In the cache analysis, the state after the conditional access is joined with the state from above, since joining of caches is always possible. However, the abstract cache then cannot guarantee that accessing block a is a cache hit, producing a second potential cache miss. Now, the number of cache misses in the compositional approach is higher than in the integrated approach.

Maximal number of instruction cache misses in compositional analysis

The last cause of overestimation of cache misses concerns the instruction accesses. The cache analysis relies on an accurate order of accesses to provide precise and sound results. However, whenever speculation occurs in the pipeline, this needs to be taken into account in the cache analysis as well.

Consider the in-order pipeline as an example: Whenever a branch is taken in the program (e.g. a conditional branch), the program counter is set when the instruction was processed in the execute stage. Looking closely at the in-order pipeline, one finds that exactly one or two accesses to the instruction cache can happen before the program counter is set to the next (actually executed) instruction, the branch target. These accesses are not explicit in the CFG, but must be considered in the cache analysis. Therefore, the cache analysis speculates once, splits if it cannot determine whether it is a hit or miss, and speculates a second time for the hit-case. In case the first access was a miss, the branch target is set before the second access is performed. They are then joined again, introducing a additional source to cause overestimation as mentioned in the previous subsection.

Experimental comparison of cache misses in both approaches

Table 5.3 shows the overestimation of cache misses for both instruction and data accesses for our selection of test cases, as well as the geometric mean of change rate for all test cases. This shows the impact of the reasons described in the previous subsections on compositional analysis.

A similar comparison was done in [HLTW03, Section V] using different processors and focusing on evaluating the effect of speculation. The produced results are similar.

	Instruction misses			Data misses		
Testcase	Max. misses on a wcet path	Max. misses		Max. misses on a wcet path	Max. misses	
compress	7,970	8,809	(+10.5 %)	6,349	6,349	(+0 %)
janne_complex	178	179	(+0.6 %)	14	14	(+0 %)
ndes	9,590	10,082	(+5.1 %)	17,748	17,894	(+0.8 %)
statemate	811	898	(+10.7 %)	34	37	(+8.8 %)
st	31,100	34,107	(+9.7 %)	53,055	53,055	(+0 %)
SCADE 1	6,127	7,117	(+16.2 %)	1,412	1,502	(+6.4 %)
SCADE 4	45,847	57,247	(+24.9 %)	329,703	330,033	(+0.1 %)
SCADE 6	13,146	15,006	(+14.1 %)	6,543	6,783	(+3.7 %)
average			(+16.0 %)			(+0.5 %)

Table 5.3: Comparison of maximal misses in integrated and compositional analysis

5.4 Changes of background latency

Motivation In the first change of parameters, we increase the latency of the background memory. In contrast to the standard setting of ten cycles to resolve an access, the latency is now set to one hundred cycles. Using ten cycles as memory latency is an arbitrary choice, since real latency depends on the memory used and its connection to the pipeline. Latency or parts of it may not affect the WCET, since the pipeline is executing instructions as well. In this case, latency is *overlapped* or *hidden* (cf. Section 4.3). The maximal hidden latency is determined by the pipeline, and additional latency affects analysis results directly.

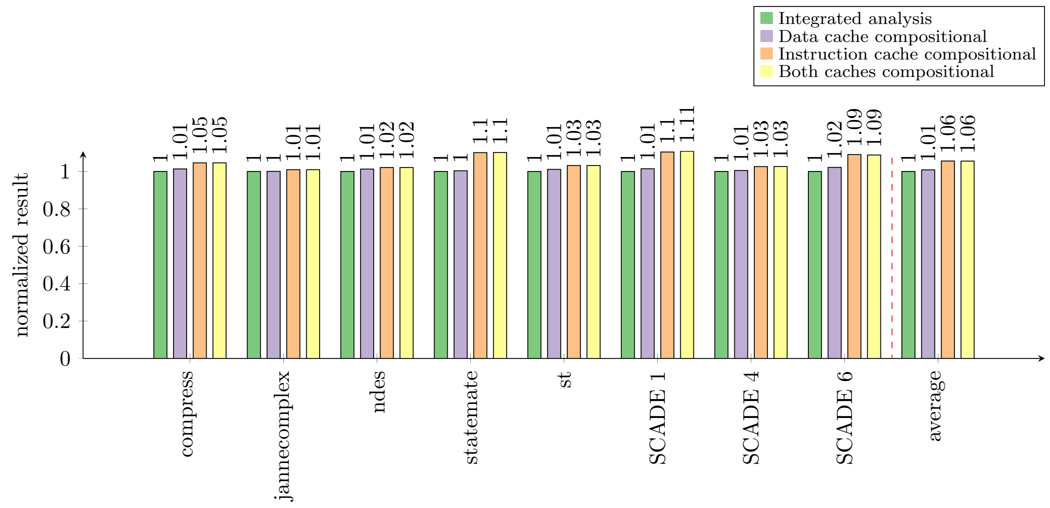


Figure 5.6: Relative bound with respect to the integrated approach with latency 100 cycles

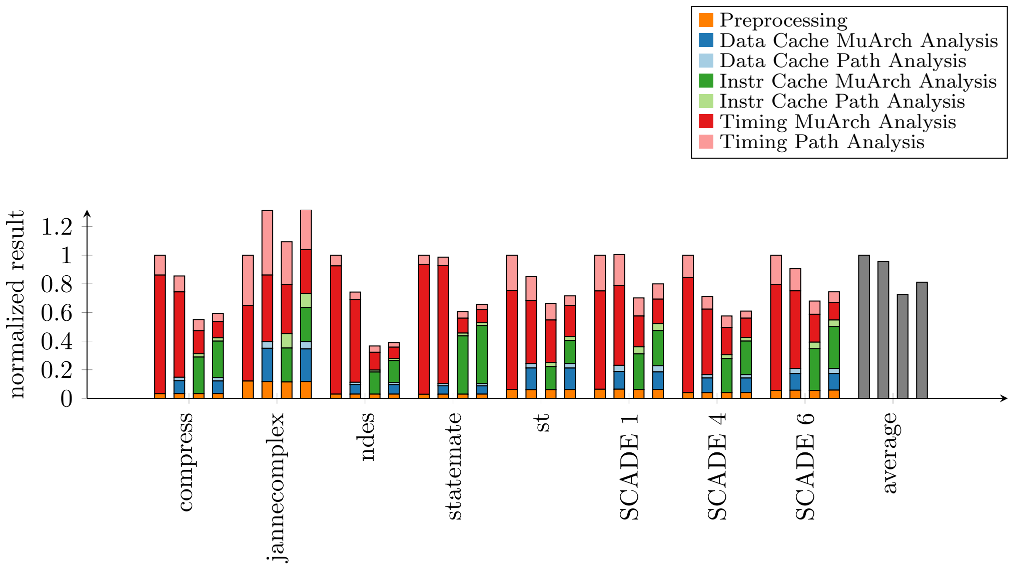


Figure 5.7: Relative analysis time with respect to the integrated approach with latency 100 cycles

Results

Geometric mean	Integrated	Data cache compositional	Instruction cache compositional	Fully compositional
Bound	100.0 %	101.032 %	106.015 %	105.998 %
Time	100.0 %	95.698 %	73.248 %	81.663 %
Memory	100.0 %	96.465 %	89.761 %	87.486 %

Table 5.4: Geometric mean for compositional behaviour with 100 cycles latency

Arithmetic mean	Integrated	Data cache compositional	Instruction cache compositional	Fully compositional
Bound	6,714,887 cyc	6,768,292 cyc	7,014,232 cyc	7,014,474 cyc
Time	5.34 s	4.64 s	3.55 s	3.88 s
Memory	94,927 KB	90,454 KB	83,513 KB	80,966 KB

Table 5.5: Arithmetic mean for compositional behaviour with 100 cycles latency

Observations First, we observe low relative overestimation of the WCET bound when considering the data cache compositionally, amounting to around one percent. Analysing the instruction cache compositional leads to an overestimation compared to the WCET bound of about 6%. Finally, the fully compositional setting has similar results in terms of the WCET bound.

Looking at the absolute values for the WCET bound, we see that the absolute overestimation of the compositional approach is slighty higher than in the standard setting. The absolute determined bound however is about seven times (7.15 in the integrated, 6.8-6.87 in the compositional approach) as high as in the standard setting. This explains why the absolute higher overestimation is relatively less overestimation.

Considering the analysis time, we can see from comparing Table 5.2 and Table 5.5 that the absolute (and thus relative) time does not change.

This is expected for the compositional analysis, since changing the background latency does only change the calculation of the contribution. For the integrated analysis, the analyser uses an optimization which fast-forwards a microarchitectural state in case the pipeline does not change apart from a ongoing memory access [JHH15]. This explains why changing the background latency may not change the analysis runtime.

Since the analysis runtime does not change, we can conclude that the added 90 cycles latency of background memory are never overlapped by the pipeline, but fast-forwarding was also used in the standard setting with 10 cycles latency.

Conclusions Analysing a microarchitecture compositionally with a high background latency of 100 cycles leads to only low overestimation compared to the standard setting. One could argue that the saved resource usage is large enough to cope with the lost precision.

The latency does not influence the resource usage of the compositional analysis. While the integrated analysis needs to cycle more and takes longer in practice, recent findings apply fast-forwarding in the integrated analysis to prevent a larger analysis time. Therefore, there is no further advantage in terms of resource usage compared to low background latency.

5.5 Changing cache sizes

Motivation Different sizes of caches affect their behaviour and therefore cache analysis and its performance. Changing the cache size does not directly affect the pipeline analysis. Therefore, we expect a larger cache to affect the integrated approach more than the compositional approach. Increasing or decreasing the cache size does lead to a different number of cache misses. Having more cache misses does affect the WCET in combination with the background latency. We therefore expect both parameters to have a large impact on the WCET, and include the last parameter, the background latency, again in the results.

Results and Observations In contrast to the other settings, results are shown differently in this section. Each table summarizes the geometrical means for one decomposition, with respect to the integrated results for each table cell. In Table 5.6 about the compositional data cache, the first data cell indicates the average relative bound and analysis time of the data compositional analysis in comparison to the integrated analysis using a data cache size of 1,024 bytes, an instruction cache size of 1,024 bytes and a background latency of 10 cycles. Each table contains three dimensions (data cache size, instruction cache size, and background latency), while the last dimension is shown as additional rows. Finally, the first cell in each table originate from Section 5.2, and the first cell in the second row originates from Section 5.4.

When comparing different table cells with each other, one has to be aware that this shows the saved time or overestimated bounds compared with the respective integrated approach. This is the only meaningful comparison: Analysing the same program for different cache sizes is not a practical application and can change their behaviour significantly, so comparing analysis performance of different cache sizes against each other is not useful.

Tables 5.7 and 5.8 work in the same manner for the remaining options: instruction cache compositional and fully compositional.

Geometric mean, data cache compositional			$\begin{pmatrix}\text{Bound}\\ \text{Time}\end{pmatrix}$
Data cache size latency	Instruction cache size 1,024 B	2,048 B	4,096 B
1,024 B 10 cyc	$\begin{pmatrix}105.129\%\\ 95.686\%\end{pmatrix}$	$\begin{pmatrix}105.091\%\\ 94.317\%\end{pmatrix}$	$\begin{pmatrix}105.090\%\\ 91.966\%\end{pmatrix}$
1,024 B 100 cyc	$\begin{pmatrix}101.032\%\\ 95.698\%\end{pmatrix}$	$\begin{pmatrix}101.023\%\\ 94.223\%\end{pmatrix}$	$\begin{pmatrix}101.021\%\\ 92.178\%\end{pmatrix}$
512 B 10 cyc	$\begin{pmatrix}105.198\%\\ 96.832\%\end{pmatrix}$	$\begin{pmatrix}105.160\%\\ 94.938\%\end{pmatrix}$	$\begin{pmatrix}105.159\%\\ 92.804\%\end{pmatrix}$
512 B 100 cyc	$\begin{pmatrix}101.043\%\\ 96.462\%\end{pmatrix}$	$\begin{pmatrix}101.035\%\\ 94.704\%\end{pmatrix}$	$\begin{pmatrix}101.033\%\\ 94.646\%\end{pmatrix}$
256 B 10 cyc	$\begin{pmatrix}105.400\%\\ 98.629\%\end{pmatrix}$	$\begin{pmatrix}105.361\%\\ 95.820\%\end{pmatrix}$	$\begin{pmatrix}105.358\%\\ 93.597\%\end{pmatrix}$
256 B 100 cyc	$\begin{pmatrix}101.069\%\\ 98.424\%\end{pmatrix}$	$\begin{pmatrix}101.061\%\\ 95.981\%\end{pmatrix}$	$\begin{pmatrix}101.056\%\\ 93.604\%\end{pmatrix}$

Table 5.6: Geometric mean for compositional data cache for different cache sizes

Analysing the data cache in a compositional manner leads to a small overestimation of 5 - 5.5% in comparison to the integrated approach. When increasing the background latency, we have a lower relative increase of about one percent (cf. Section 5.4). The overestimation of compositional bounds compared to integrated bounds does not noticeably change when changing instruction or data cache size.

In terms of analysis time, the results differ a little. Analysis time does change when changing the sizes of caches. For the compositional data cache, increasing the cache size decreases the relative time of the compositional analysis. For the instruction cache, which is analysed with the pipeline, the compositional time also decreases when increasing the cache size.

In order to explain this behaviour, we need to know how a different cache size affect the analysis. In general, increasing the cache size introduces more uncertainty, since the initial, unknown cache is larger and the number of accesses until the cache is known is longer. This leads to more splits during both integrated and compositional analysis. Using the basic idea of the compositional approach described in Figure 4.1, additional splits affect the integrated analysis more than the compositional analysis. We therefore expect increasing any cache size to increase the absolute analysis time for both approaches, while decreasing the relative analysis time of the compositional approach. This is consistent with the findings in Table 5.6 and also the absolute numbers.

Geometric mean, instruction cache compositional			$\begin{pmatrix}\text{Bound}\\\text{Time}\end{pmatrix}$
Data cache size latency	Instruction cache size 1,024 B	2,048 B	4,096 B
1,024 B 10 cyc	$\begin{pmatrix}112.145\%\\73.208\%\end{pmatrix}$	$\begin{pmatrix}112.118\%\\69.953\%\end{pmatrix}$	$\begin{pmatrix}112.100\%\\62.740\%\end{pmatrix}$
1,024 B 100 cyc	$\begin{pmatrix}106.015\%\\73.248\%\end{pmatrix}$	$\begin{pmatrix}106.034\%\\69.802\%\end{pmatrix}$	$\begin{pmatrix}106.024\%\\62.801\%\end{pmatrix}$
512 B 10 cyc	$\begin{pmatrix}112.139\%\\74.140\%\end{pmatrix}$	$\begin{pmatrix}112.112\%\\70.448\%\end{pmatrix}$	$\begin{pmatrix}112.093\%\\63.199\%\end{pmatrix}$
512 B 100 cyc	$\begin{pmatrix}105.998\%\\73.937\%\end{pmatrix}$	$\begin{pmatrix}106.017\%\\70.362\%\end{pmatrix}$	$\begin{pmatrix}106.008\%\\63.025\%\end{pmatrix}$
256 B 10 cyc	$\begin{pmatrix}112.124\%\\74.684\%\end{pmatrix}$	$\begin{pmatrix}112.097\%\\70.340\%\end{pmatrix}$	$\begin{pmatrix}112.076\%\\63.152\%\end{pmatrix}$
256 B 100 cyc	$\begin{pmatrix}105.960\%\\74.526\%\end{pmatrix}$	$\begin{pmatrix}105.979\%\\70.542\%\end{pmatrix}$	$\begin{pmatrix}105.966\%\\63.200\%\end{pmatrix}$

Table 5.7: Geometric mean for compositional instruction cache for different cache sizes

Considering the instruction cache compositionally, there are again no observable differences in the determined WCET bound when changing either of the caches. The bound only changes when the latency is increased, as discussed in Section 5.4.

Increasing the instruction or data cache size decreases the relative analysis time of the compositional analysis. For changes in the instruction cache size, the relative analysis time ranges from 62 - 74%. Changing the data cache size results in variation of the runtime of up to 1.5 percent points. While this can be explained by the same arguments as for the data compositional option, changing the size of the data cache, which is considered integrated, does not influence the runtime as much as the changing the instruction cache size in the last option. The high variation in the last option could be caused by a high number of instruction accesses.

Geometric mean, both caches compositional			(Bound / Time)
Data cache size / latency	Instruction cache size 1,024 B	2,048 B	4,096 B
1,024 B / 10 cyc	(111.708% / 81.725%)	(111.682% / 77.623%)	(111.666% / 69.106%)
1,024 B / 100 cyc	(105.998% / 81.663%)	(106.017% / 77.586%)	(106.008% / 69.115%)
512 B / 10 cyc	(111.687% / 82.394%)	(111.661% / 78.187%)	(111.646% / 69.643%)
512 B / 100 cyc	(105.985% / 82.326%)	(106.004% / 78.074%)	(105.995% / 69.572%)
256 B / 10 cyc	(111.610% / 84.104%)	(111.583% / 78.801%)	(111.566% / 70.270%)
256 B / 100 cyc	(105.939% / 83.926%)	(105.958% / 78.990%)	(105.946% / 70.223%)

Table 5.8: Geometric mean for compositional caches for different cache sizes

Finally, considering both caches compositionally, the bound again only changes on the background latency.

The analysis time sees minor change in terms of the data cache, but bigger changes when changing the instruction cache size. These range from 69 - 84 %.

These results did not contribute further than confirm our findings in the previous two options.

Conclusions The key observation when changing cache sizes is that it does neither influence analysis bound nor analysis runtime significantly. The impact of other microarchitectural parameters is generally much higher.

To a lesser extent, we find a tendency that increasing any cache size does increase the analysis runtime of both approaches. The compositional approach is not affected as much, and therefore is relatively better with larger caches.

5.6 On penalties for cache misses

Motivation From the previous results we saw that the compositionally determined bounds may contain high overestimation. In Section 4.4 about the contribution of each component, it was not proven which bounds on cache misses can be used. This section contains two experiments on the precision of different cache miss contributions.

The first one assumes precise penalties on misses, i.e. a miss can not cause more timing delay than the access itself. This describes the direct effect of a cache miss and would be ideal for compositional analysis.

The second penalty contribution uses the non-refined contributions as given by Equations 4.3, 4.5, 4.6 and 4.14. Generally, for each miss, it is assumed to be blocked by another access to the memory for the time of one access to the background memory, resulting in a doubled penalty. Proving the correctness of this cache miss penalty might be easier.

Results

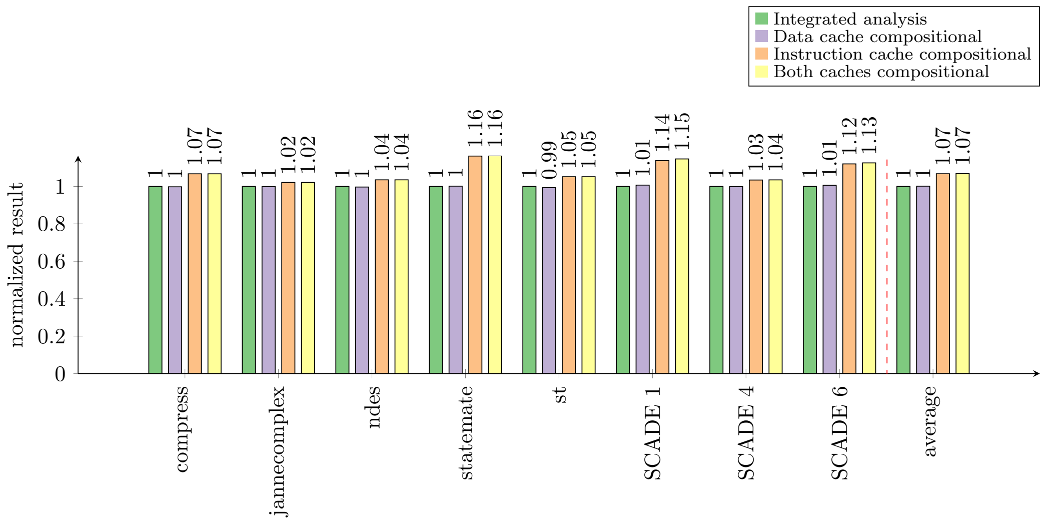


Figure 5.8: Relative bound when considering direct effect of a cache miss only

Geometric mean	Integrated	Data cache compositional	Instruction cache compositional	Fully compositional
(Bound)	(100.0 %)	(100.145 %)	(106.784 %)	(106.866 %)

Table 5.9: Geometric mean for relative bounds when considering direct effect of a cache miss only

Observations Considering the data cache compositionally, the overestimation for single cache miss penalties amounts for 0.15 % of the integrated bound. However, it is possible that some test cases even underestimate the WCET bound, as seen for *st*.

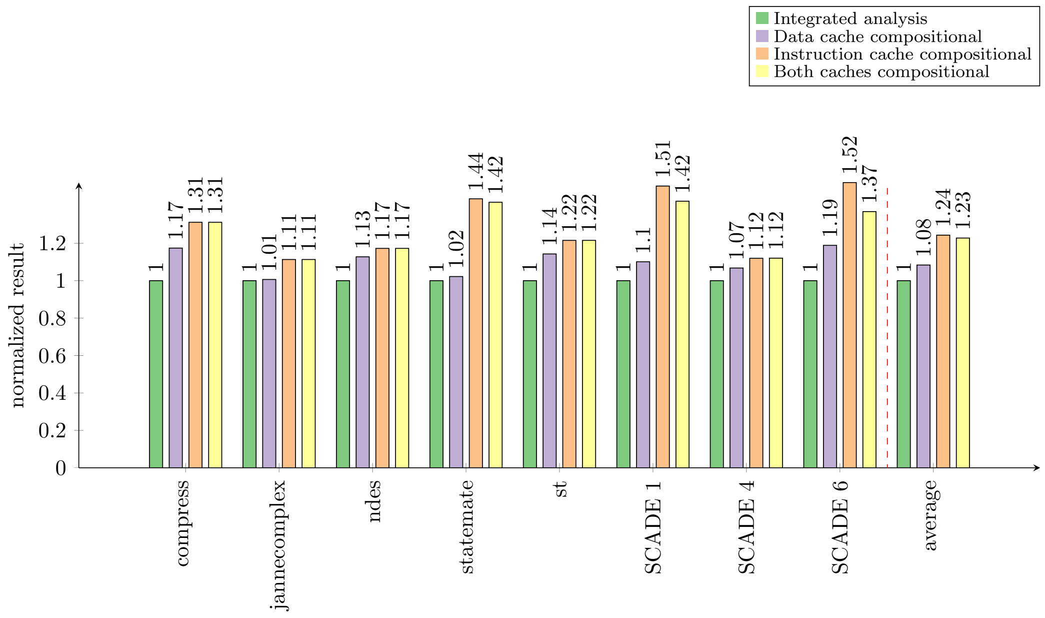


Figure 5.9: Relative bound when assuming doubled cache miss contribution

Geometric mean	Integrated	Data cache compositional	Instruction cache compositional	Fully compositional
$\begin{pmatrix}\text{Bound}\\ \text{Time}\\ \text{Memory}\end{pmatrix}$	$\begin{pmatrix}100.0\ \%\\ 100.0\ \%\\ 100.0\ \%\end{pmatrix}$	$\begin{pmatrix}108.381\ \%\\ 95.686\ \%\\ 96.522\ \%\end{pmatrix}$	$\begin{pmatrix}124.341\ \%\\ 73.208\ \%\\ 89.707\ \%\end{pmatrix}$	$\begin{pmatrix}122.801\ \%\\ 81.725\ \%\\ 87.518\ \%\end{pmatrix}$

Table 5.10: Geometric mean for relative bounds when assuming double cache miss contribution

While in the standard setting with precise bounds the overestimation amounts for just above 5 % (cf. Figure 5.2 and Table 5.9), the double cache miss penalty setting overestimates the integrated bound by 8.3 %.

The overestimation for compositional instruction cache changes from 7% in the single penalty setting to 12% in the standard setting and finally reaches 24% in the double penalty setting. The results of the fully compositional setting is similar, starting at 7% and going up to 11% and 23%, respectively.

Conclusions Using single cache miss bounds sometimes results in a more precise bound than in the integrated option, raising skepticism towards the soundness of the direct effect as cache miss penalty. Investigating in more detail, a small example could be found where the lower bound did not contain a data access being blocked by an ongoing instruction access, and a too low bound. Using the single cache miss penalties, and disregarding possible indirect effects such as blocking is therefore not sound.

In the instruction cache compositional option, the overestimation is considerably higher with 7%. This suggests that although some blocking when accessing the background latency might be ignored, missing the overlapping by the pipeline and conservative approximation of speculation (cf. Section 5.3) accounts for a large relative amount. The latter argument is supported by the low overestimation when analysing with the data cache compositionally.

For all options one thing is consistent: Reducing the penalty for one cache miss leads to an improvement in the determined upper bound for WCET. Proving small, but sound penalties for a microarchitecture is therefore crucial for compositional analysis. This might require careful design decisions when building microarchitecture. Minimising the overlap of a pipeline however is counter-intuitive for hardware builders: In terms of average execution time, it is probably more beneficial to increase the overlap as much as possible.

Finally, it is worth mentioning that time and memory usage do not noticeably change, and are thus not depicted here. Changing the contribution calculation has neglectable impact on the analysis resource usage.

5.7 Context sensitivity

Motivation Context sensitivity heavily affects both precision of bounds and resource usage. Low context sensitivity leads to a cheap analysis. However, bounds on the WCET are also very imprecise. On the other hand, high context sensitivity leads to very precise bounds, but the analysis time also rises highly, since many programs flows are differentiated. In Chapter 4, we argued that uncertainty does not affect the compositional approach as much as the integrated, and distinguishing cases is similar in that regards (cf. Figure 4.1). Therefore, using high context sensitivity might be enabled in compositional analysis. In this section, the impact of having full context sensitivity is evaluated. Using full context sensitivity, we distinguish all different call sites and loop information, and do never discard information as we get new contexts.

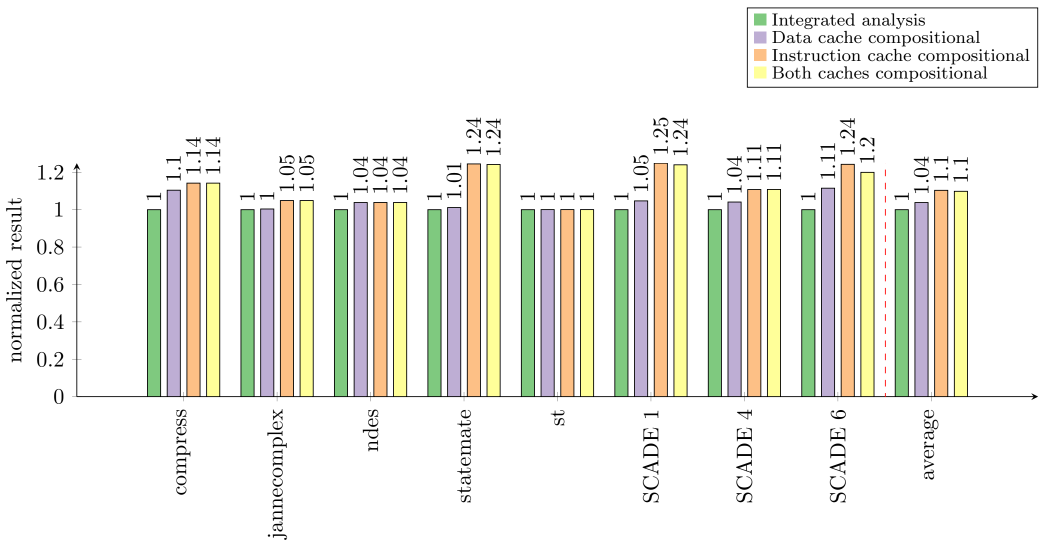


Figure 5.10: Relative bound with respect to the integrated approach with full context sensitivity

Geometric mean	Integrated	Data cache compositional	Instruction cache compositional	Fully compositional
$\begin{pmatrix}\text{Bound}\\\text{Time}\\\text{Memory}\end{pmatrix}$	$\begin{pmatrix}100.0\ \%\\100.0\ \%\\100.0\ \%\end{pmatrix}$	$\begin{pmatrix}104.247\ \%\\104.036\ \%\\96.027\ \%\end{pmatrix}$	$\begin{pmatrix}109.943\ \%\\90.449\ \%\\89.956\ \%\end{pmatrix}$	$\begin{pmatrix}109.620\ \%\\104.199\ \%\\87.549\ \%\end{pmatrix}$

Table 5.11: Geometric mean for compositional behaviour with full context sensitivity

Arithmetic mean	Integrated	Data cache compositional	Instruction cache compositional	Fully compositional
$\begin{pmatrix}\text{Bound}\\\text{Change}\end{pmatrix}$	$\begin{pmatrix}803,633\text{ cyc}\\83.513\ \%\end{pmatrix}$	$\begin{pmatrix}832,681\text{ cyc}\\84.476\ \%\end{pmatrix}$	$\begin{pmatrix}863,281\text{ cyc}\\83.687\ \%\end{pmatrix}$	$\begin{pmatrix}862,187\text{ cyc}\\83.685\ \%\end{pmatrix}$
$\begin{pmatrix}\text{Time}\\\text{Change}\end{pmatrix}$	$\begin{pmatrix}810.08\text{ s}\\15,198.5\ \%\end{pmatrix}$	$\begin{pmatrix}994.18\text{ s}\\21,472.6\ \%\end{pmatrix}$	$\begin{pmatrix}1,045.77\text{ s}\\29,541.5\ \%\end{pmatrix}$	$\begin{pmatrix}1,280.18\text{ s}\\32,994.3\ \%\end{pmatrix}$
$\begin{pmatrix}\text{Memory}\\\text{Change}\end{pmatrix}$	$\begin{pmatrix}125,643\text{ KB}\\132.4\ \%\end{pmatrix}$	$\begin{pmatrix}118,529\text{ KB}\\131.0\ \%\end{pmatrix}$	$\begin{pmatrix}110,050\text{ KB}\\131.9\ \%\end{pmatrix}$	$\begin{pmatrix}106,252\text{ KB}\\131.3\ \%\end{pmatrix}$

Table 5.12: Arithmetic mean for compositional behaviour with full context sensitivity and change from standard context sensitivity

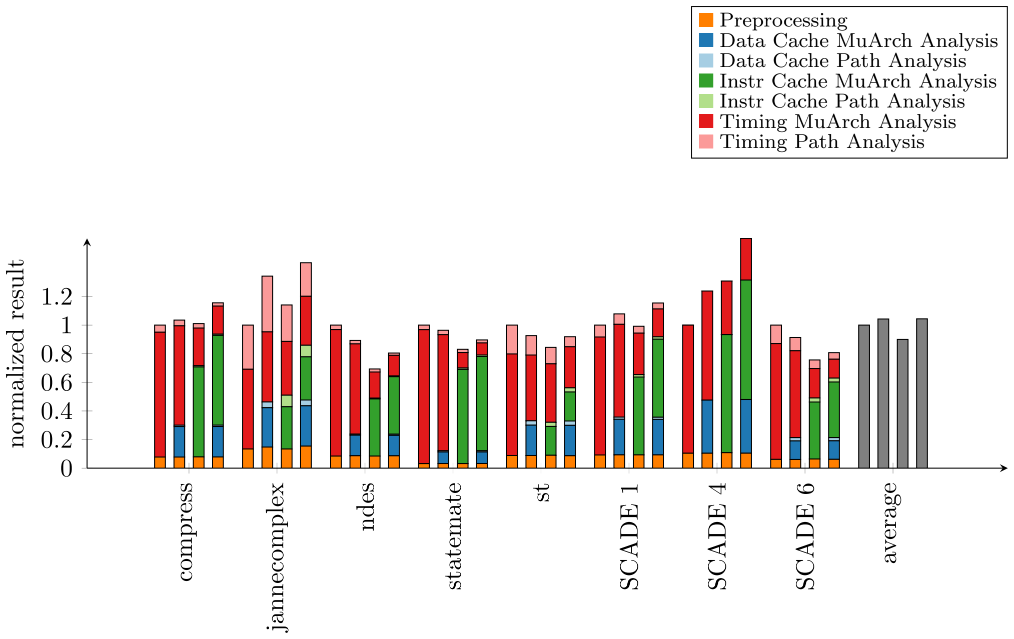


Figure 5.11: Relative analysis time with respect to the integrated approach with full context sensitivity

Observations Comparing the compositional to the integrated approach, bounds increase by about 4.2 % for compositional data cache, and about 9.5 % for the other decompositions. For all options, this is slightly less than in the standard setting. Considering the absolute values summarized in the arithmetic mean (Table 5.12), there is an improvement of 15 % in the integrated approach towards the standard setting with low context sensitivity (cf. Table 5.2).

In terms of analysis time, using full compositionality takes much more time for all options, up to a factor or 330 (32,994 % of the standard setting) for full compositionality. The compositional approach now uses more analysis time when analysing the data cache or both caches compositionally, when analysing the instruction cache compositionally, it still saves time.

Figure 5.11 shows that separating the instruction cache analysis reduces the pipeline analysis in this scenario largely for a multitude of test cases, indicating that this is a large portion of the analysis.

Memory consumption increases by about 31 % for all options, the compositional approach uses less memory than the integrated approach again.

Conclusions Using the compositional approach for full context sensitivity improves the relative upper bound. However, it does no longer save time or as much time in

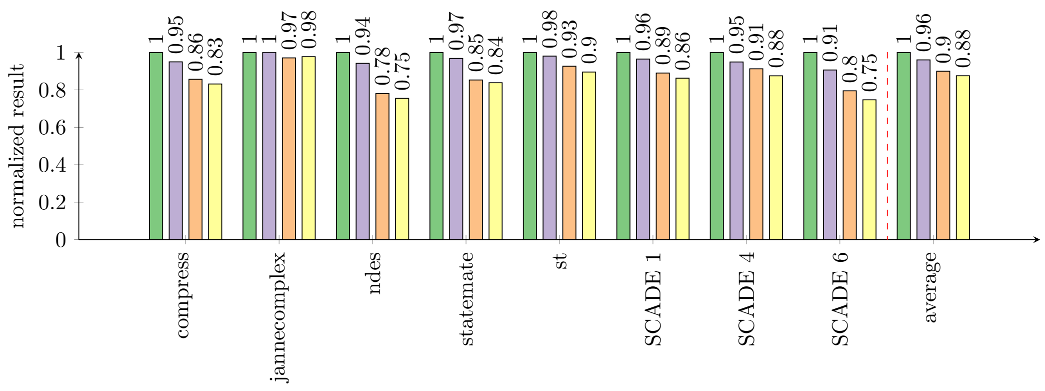


Figure 5.12: Relative analysis peak memory usage with respect to the integrated approach with full context sensitivity

comparison to standard context sensitivity. This suggests that using compositionality for higher context sensitivity is detrimental.

Apart from the evaluated option the compositional approach enables to set different context sensitivity for different components. It may be beneficial to use individual context sensitivity for the components, obtaining higher precision while not increasing the runtime too much.

5.8 Out-of-order performance

Motivation While in-order pipelines representing simple pipelining, more complex microarchitectures are also used for timing-aware systems. This section evaluates the compositional approach for the out-of-order pipeline described in Section 3.2. In general, using more complex pipeline increases the size of a single microarchitectural state, and possibly the number of uncertain events. In our out-of-order pipeline, speculating on instruction accesses is not enabled, and data accesses are performed in-order. This allows the separate cache analysis to not depend on the out-of-order pipeline.

Using speculation of instruction accesses, we expect a huge grow in the number of possible access sequences. We focused thus on a cache analysis for the data cache, and will only consider the data cache to be compositionally in this section.

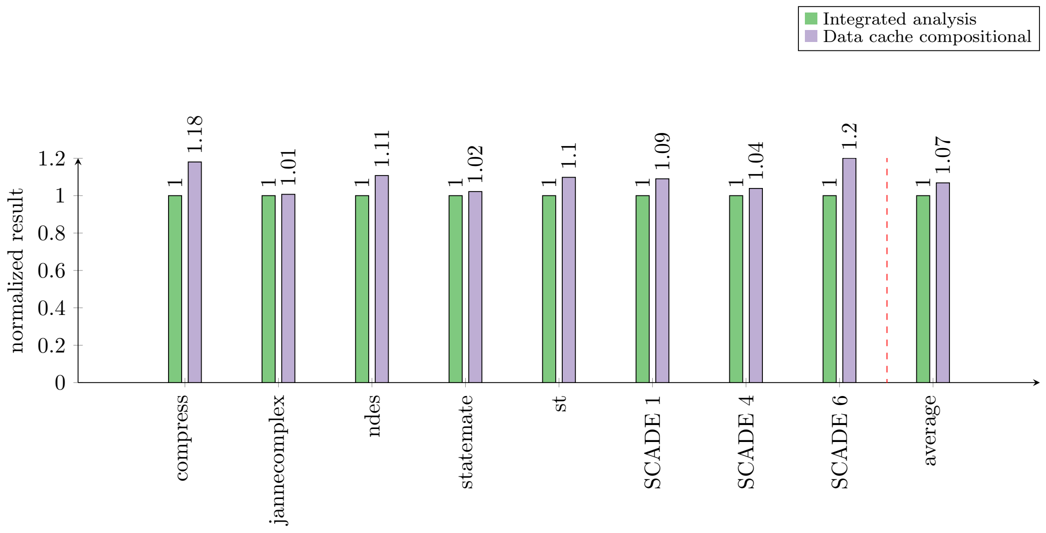


Figure 5.13: Relative WCET bound compared to the integrated approach on the out-of-order pipeline

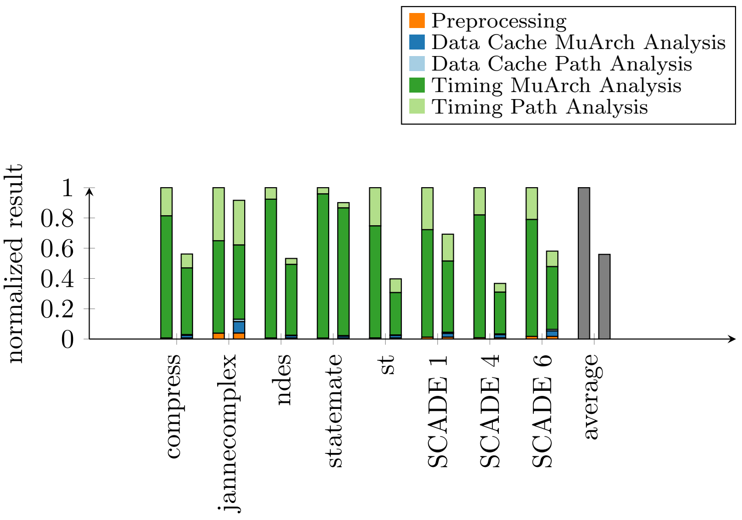


Figure 5.14: Relative analysis time on the out-of-order pipeline

Observations The results for the WCET bound of the out-of-order pipeline shows similar overestimation in the compositional approach as for the in-order pipeline with 7.7%. Considering the testcases individually, *janne_complex* shows the least overestimation, although the range of overestimation does not grow above 20%.

Considering the analysis time, there is a decrease of 43.7% when analysing compo-

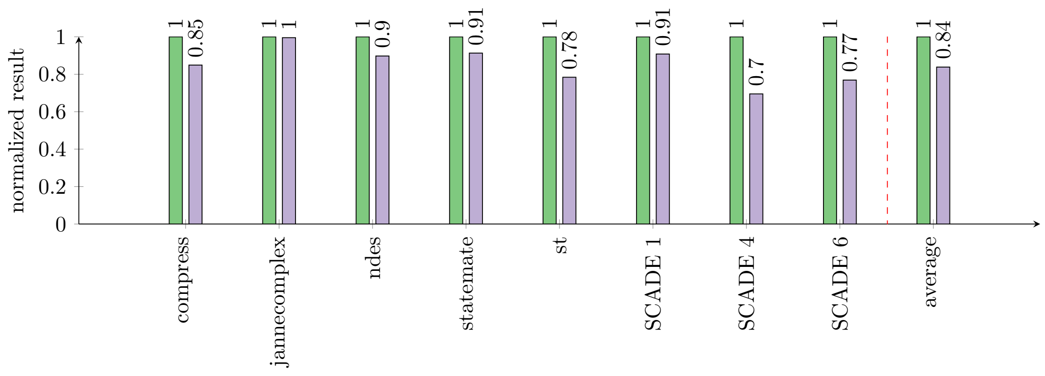


Figure 5.15: Relative analysis peak memory usage on the out-of-order pipeline

Geometric mean	Integrated	Data cache compositional
$\begin{pmatrix}\text{Bound}\\\text{Time}\\\text{Memory}\end{pmatrix}$	$\begin{pmatrix}100.0\ \%\\100.0\ \%\\100.0\ \%\end{pmatrix}$	$\begin{pmatrix}107.717\ \%\\56.233\ \%\\83.835\ \%\end{pmatrix}$

Table 5.13: Geometric mean for compositional behaviour on the out-of-order pipeline

sitionally in comparison to the integrated approach. Comparing this relative saved time to the in-order pipeline (4.3%), the saved analysis time is much higher.

Finally, the peak memory usage of the compositional approach with 83.8% of the integrated approach is less than the relative peak memory usage when using the in-order pipeline (96.5%).

Conclusions Our first results concerning a more complex microarchitecture such as the out-of-order pipeline seems more promising than for the in-order pipeline. The most important thing to consider here is the analysis time. While peak memory is not as important, determining penalties as in Chapter **??** is harder and might lead to worse penalties and thus worse WCET here than conjectured.

In order to enable compositional analysis, the out-of-order pipeline needed to be slightly adjusted: In case of a branch, fetching is stalled. While this allows for a precise cache analysis, it degrades performance of the pipeline, which is not desireable, but necessary to enable a sound compositional analysis. The degraded performance is not included here, and further investigation on complex microarchitecture is required to reach a complete conclusion.

5.9 Unblocked stores

Motivation The last setting to be altered refers to the behaviour of stores to the data memory. In the standard setting, the store instruction stays in the pipeline until the store was completed in the background memory. However, there is also the option to finish the instruction in the pipeline while the store is still processing, namely *unblocked stores*. Using unblocked stores, the pipeline in the integrated approach is not blocked as long as when using blocked stores, and allows for more overlapping. We expect the analysis to be more expensive. The compositional approach on the other hand, might not suffer from this effect.

When using unblocked stores, an instruction access to the background memory can get blocked without an instruction in the pipeline. Therefore, the penalty needs to account for one access as the indirect effect. For data accesses, it is still not possible to advance more than six cycles.

Results

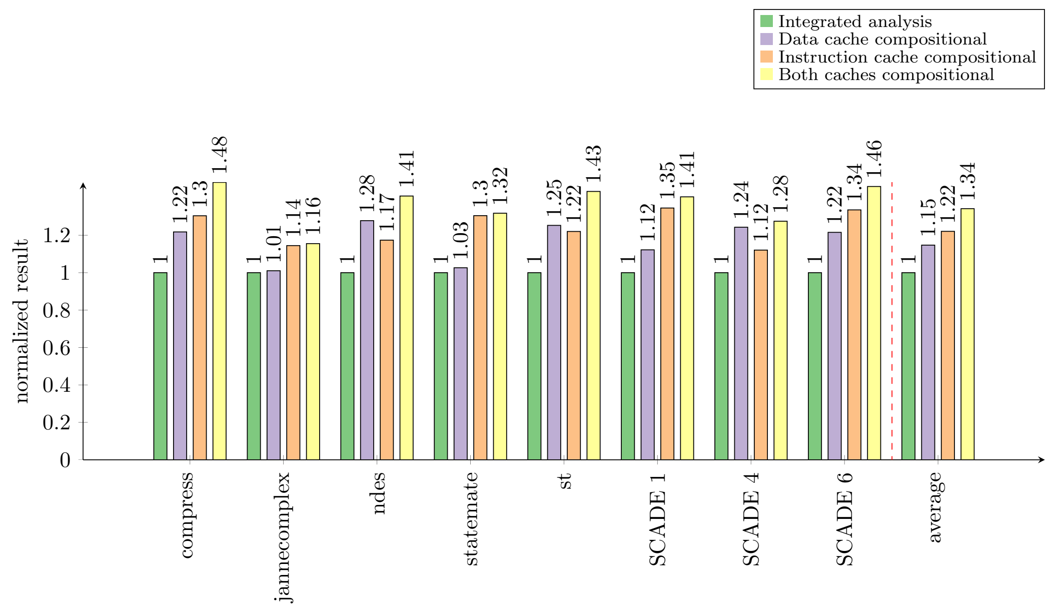


Figure 5.16: Relative bound in regard to the integrated approach with unblocked stores

Observations Using unblocked stores, each compositional option experiences a significant increase in the determined upper bound (cf. Table 5.14). Compared to the standard setting, the integrated approach shows the largest decrease in bound, while the compositional approach can only partly or not at all profit in this setting.

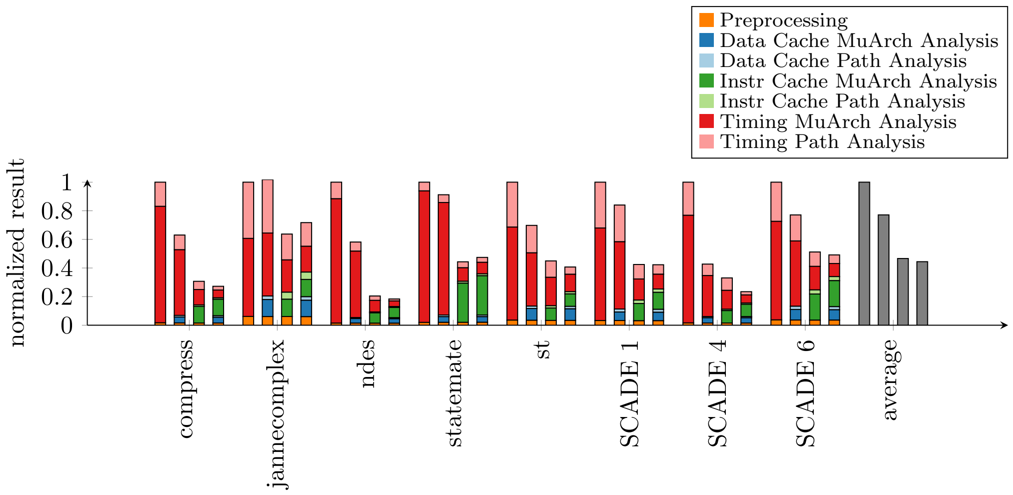


Figure 5.17: Relative analysis time in regard to the integrated approach with unblocked stores

Geometric mean	Integrated	Data cache compositional	Instruction cache compositional	Fully compositional
Bound	100.0 %	117.327 %	121.783 %	136.194 %
Time	100.0 %	77.494 %	47.500 %	45.314 %
Memory	100.0 %	89.714 %	78.877 %	72.260 %

Table 5.14: Geometric mean for compositional behaviour with unblocked stores

The time used to determine the WCET is largely reduced when using the compositional approach, up to 55 % when considering both caches compositionally. It is worth noticing that the relative time saved is also caused by the large increase of the integrated approach ($\approx$ 80%), while the compositional approach does not suffer as much from unblocked stores.

Finally, the peak memomy shows moderate reductions in the compositional approach.

Conclusions Using a microarchitecture that handles stores to memory in an unblocked manner, we see that the compositional approach provides significantly worse precision than the integrated approach. Part of the imprecision is based on the contributions as explained in Section 4.4. Since unblocked stores allow for unlimited overlap, we cannot use refined contributions. While the integrated approach determines overlap precisely, the compositional does not take this into account.

The added precision in the integrated approach does reflect in the increase in analysis time. The relation between WCET bound and analysis time alone however does not

Arithmetic mean	Integrated	Data cache compositional	Instruction cache compositional	Fully compositional
$\begin{pmatrix}\text{Bound}\\ \text{Change}\end{pmatrix}$	$\begin{pmatrix}802,407 \text{ cyc}\\ 85.4\ \%\end{pmatrix}$	$\begin{pmatrix}957,543 \text{ cyc}\\ 97,1\ \%\end{pmatrix}$	$\begin{pmatrix}957,851 \text{ cyc}\\ 92,9\ \%\end{pmatrix}$	$\begin{pmatrix}1,083,262 \text{ cyc}\\ 105.1\ \%\end{pmatrix}$
$\begin{pmatrix}\text{Time}\\ \text{Change}\end{pmatrix}$	$\begin{pmatrix}9.66 \text{ s}\\ 181.2\ \%\end{pmatrix}$	$\begin{pmatrix}6.54 \text{ s}\\ 141.3\ \%\end{pmatrix}$	$\begin{pmatrix}4.43 \text{ s}\\ 125.1\ \%\end{pmatrix}$	$\begin{pmatrix}4.07 \text{ s}\\ 104.9\ \%\end{pmatrix}$
$\begin{pmatrix}\text{Memory}\\ \text{Change}\end{pmatrix}$	$\begin{pmatrix}123,649 \text{ KB}\\ 130.3\ \%\end{pmatrix}$	$\begin{pmatrix}106,208 \text{ KB}\\ 117.4\ \%\end{pmatrix}$	$\begin{pmatrix}95,160 \text{ KB}\\ 114.1\ \%\end{pmatrix}$	$\begin{pmatrix}84,584 \text{ KB}\\ 104.5\ \%\end{pmatrix}$

Table 5.15: Arithmetic mean for compositional behaviour with full context sensitivity and change from standard context sensitivity

advise to use the compositional approach.

6 Future Work

While this thesis gains practical insight into the decomposition of pipeline and caches, there is work left to improve on this approach, or use it in other settings.

As shown in Section 5.6, having penalties as precise as possible is crucial for the practical application of compositionality. The penalties explained in Section 4.4, although strongly conjectured, are not formally proven to be sound. This proof is necessary to soundly use the refined penalties, which give the largest improvement in terms of bound precision of this decomposition.

Compositionality requires microarchitectures to not proceed too far when a timing accident happens, i.e. that in a state with a possible timing accident, both paths will at some point have comparable states again (cf. Figure 4.3). Finding microarchitectures which support this behaviour without (or with only slightly) degraded performance is still an open task.

In Section 4.1, we mentioned that cumulative information is easier to use in a compositional analysis. This option was not used in the evaluation, and can improve the bounds in both approaches. In the compositional approach however, we suspect it to be less expensive, improving the relative resource usage. One example of cumulative information is persistence information, as obtained by persistence analysis [Cul13].

Since the compositional approach performs individual analyses for the components, one can adjust the parameters for each analysis, e.g. context sensitivity, to achieve a high precision for some components. As an example, to achieve precise address information when processing an array in a loop, the analysis can peel the loop completely for the cache analysis, but keep the number of contexts low for the pipeline analysis.

Apart from pipeline and caches, there are other decompositions, partly suggested already in [Hah14], which are worth evaluating. In particular, considering Dynamic random access memory (DRAM) refreshes as a component of a decomposition seems promising, since it introduces splits for every access to the background memory (whether there is a refresh or not), although the probability of a refresh is low. However, proving a decomposition to be timing compositional might not be easy.

Finally, the out-of-order pipeline in this thesis was just analysed on a basic level. In real-time systems, larger components, complex speculation or out-of-order data accesses are not unusual. Analysing more complex microarchitectures might show better applications of compositional analysis.

7 Summary

Analysing pipeline and caches of a microarchitecture of a real-time system compositionally suffers from lost precision, and benefits from reduced resource usage when comparing it to the state-of-the-art approach. The reduced resource usage is still in the same order of magnitude, and does not enable to e.g. change the process of WCET analysis. Since the main result is the WCET bound, we valuate the precision higher, and thus the integrated approach.

We evaluated a variety of settings and parameters and demonstrated only low influence on the trade-off between bound precision and resource usage. Although there is a small tendency that more complex microarchitectures such as out-of-order pipelines can be analysed more efficiently by compositional analysis, we consider the integrated approach to be still superior, due to the higher precision.

In order to use compositional analysis, a microarchitecture has to support compositionality by ensuring the condition described in 4.2. Finding such a microarchitecture and proving it to be anomaly-free, i.e. satisfying the condition, is hard.

Changing an out-of-order pipeline to support compositionality, e.g. by stalling, while degrading the actual performance, might still be beneficial, since the resource usage when analysing the out-of-order pipeline drops significantly.

Section 4.4 shows that the compositional approach has to account for indirect effects to ensure a sound analysis. This indirect effect also applies for cache-related preemption delay (CRPD), using only the direct effect as block reload time (BRT), as e.g. suggested by [ADM12], is not sufficient and thus not sound.

Using the compositional approach for other decompositions can still be successful. There is a high interdependency between pipeline and caches, resulting in e.g. speculative accesses or overlaps when missing the cache. This prevents higher precision and degrades performance, and we expect there to be better choices on decompositions. Choosing to decompose other parts of a microarchitecture with less interdependency, such as DRAM refreshes, may lead to substantial improvements.

Finally, the compositional approach enables cumulative information or persistence analysis to be included without increasing analysis complexity in the path analysis.

In general, the compositional approach might be considered in different scenarios than the integrated approach. While there are scenarios in which the integrated approach is beneficial, we expect there to be scenarios (e.g. with low interdependency) in which the compositional approach performs better.

Bibliography

[ADM12] Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.

[AFMW96] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In Radhia Cousot and David A. Schmidt, editors, *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings*, volume 1145 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1996.

[All70] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[Cou01] Patrick Cousot. Abstract interpretation based formal methods and future challenges. In Reinhard Wilhelm, editor, *Informatics - 10 Years Back. 10 Years Ahead.*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer, 2001.

[Cul13] Christoph Cullmann. Cache persistence analysis: Theory and practice. *ACM Trans. Embedded Comput. Syst.*, 12(1s):40, 2013.

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen WCET benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, pages 136–146, 2010.

[GESL06] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006), 5-8 December 2006, Rio de Janeiro, Brazil*, pages 57–66, 2006.

[Goh09] Dan Gohman. Scalar evolution and loop optimization. 2009 LLVM Developers' meeting, 2009. Available online at http://llvm.org/devmtg/2009-10/#proceedings.

[Gru12] Daniel Grund. *Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU.* PhD thesis, Saarland University, 2012.

[Hah14] Sebastian Hahn. Defining compositionality in execution time analysis. Master's thesis, Saarland University, 2014.

[HLTW03] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.).* Morgan Kaufmann, 2012.

[JHH15] Michael Jacobs, Sebastian Hahn, and Sebastian Hack. WCET analysis for multi-core processors with shared buses and event-driven bus arbitration. In *Accepted and to appear at: Proceedings of the 23nd International Conference on Real-Time Networks and Systems*, 2015.

[LA04] Chris Lattner and Vikram S. Adve. The LLVM compiler framework and infrastructure tutorial. In *Languages and Compilers for High Performance Computing, 17th International Workshop, LCPC 2004, West Lafayette, IN, USA, September 22-24, 2004, Revised Selected Papers*, pages 15–16, 2004.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, pages 456–461, 1995.

[LS99] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA, December 1-3, 1999*, pages 12–21, 1999.

[MR05] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, pages 5–20, 2005.

[RWT$^+$06] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*, 2006.

[Ste10] Ingmar Jendrik Stein. *ILP-based path analysis on abstract pipeline state graphs.* PhD thesis, Saarland University, 2010.

[The04] Henrik Theiling. *Control flow graphs for real-time systems analysis: reconstruction from binary executables and usage in ILP-based path analysis.* PhD thesis, Saarland University, 2004.

[The05] Stephan Thesing. *Safe and precise WCET determination by abstract interpretation of pipeline models.* PhD thesis, Saarland University, 2005.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

List of Figures

List of Tables

Abbreviations

ALU Arithmetic logic unit.

BB Basic block.

CDB Common data bus.

CFG Control flow graph.

CRPD Cache-related preemption delay.

DRAM Dynamic random access memory.

EX Execute stage.

FU Functional unit.

GPR General purpose register.

ID Instruction decode.

IF Instruction fetch.

ILP Integer Linear Programming.

IQ Instruction queue.

LLVM Low-level virtual machine.

LRU Least recently used.

MEM Memory access.

PC Program counter.

ROB Re-order buffer.

RS Reservation station.

RST Register status table.

SRAM Static random access memory.

WB Write back.

WCET Worst-case execution time.