Final Thesis

Cache Prediction and Execution Time Analysis on Real-Time MPSoC

by

Carl-Fredrik Neikter

LIU-IDA/LITH-EX-A--08/046--SE

2008-10-28
Final Thesis

Cache Prediction and Execution Time Analysis on Real-Time MPSoC

by

Carl-Fredrik Neikter

LIU-IDA/LITH-EX-A--08/046--SE
2008-10-28

Supervisor: MSc. Jakob Rosén
Dept. of Computer and Information Science
at Linköping University

Examiner: Prof. Petru Eles
Dept. of Computer and Information Science
at Linköping University
Abstract

Real-time systems do not only require that the logical operations are correct. Equally important is that the specified time constraints always are complied. This has successfully been studied before for mono-processor systems. However, as the hardware in the systems gets more complex, the previous approaches become invalidated. For example, multi-processor systems-on-chip (MPSoC) get more and more common every day, and together with a shared memory, the bus access time is unpredictable in nature. This has recently been resolved, but a safe and not too pessimistic cache analysis approach for MPSoC has not been investigated before. This thesis has resulted in designed and implemented algorithms for cache analysis on real-time MPSoC with a shared communication infrastructure. An additional advantage is that the algorithms include improvements compared to previous approaches for mono-processor systems. The verification of these algorithms has been performed with the help of data flow analysis theory. Furthermore, it is not known how different types of cache miss characteristic of a task influence the worst case execution time on MPSoC. Therefore, a program that generates randomized tasks, according to different parameters, has been constructed. The parameters can, for example, influence the complexity of the control flow graph and average distance between the cache misses.

Keywords: Real-time systems, MPSoC, static timing analysis, worst case execution time, cache memory, cache analysis, data flow analysis, control flow graph, task generation, randomization
Contents

1 Introduction 1
   1.1 Background ........................................... 2
   1.2 Problem and Objectives .............................. 4
   1.3 Methods and Sources ................................. 4
   1.4 Structure ............................................ 5

2 Theory Revisited 7
   2.1 Cache Memories ........................................ 7
   2.2 Control Flow Graph .................................... 11
   2.3 Task Graph ............................................ 11
   2.4 Partial Sets and Lattices .............................. 13
   2.5 Integer Linear Programming ............................ 14
   2.6 Probability Theory .................................... 15

3 WCET on Mono-Processor Systems 17
   3.1 Measurement ............................................ 18
   3.2 Static Timing Analysis ................................ 18
   3.3 Early Work ............................................. 19
   3.4 Implicit Path Enumeration ............................. 20
      3.4.1 Program Constraints ............................... 21
      3.4.2 ILP Solver ......................................... 22
   3.5 Direct Mapped Instruction Cache ....................... 22
      3.5.1 Terminology ....................................... 23
      3.5.2 New Cost Function ................................. 24
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.3</td>
<td>Cache Constraints</td>
<td>24</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Cache Conflict Graph</td>
<td>25</td>
</tr>
<tr>
<td>3.6</td>
<td>Set Associativity and Data Caches</td>
<td>27</td>
</tr>
<tr>
<td>3.7</td>
<td>Program Segments</td>
<td>27</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Path Classification</td>
<td>30</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Timing Estimation</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>WCET on MPSoC</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>The Problem</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>Bus Access Schedule</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>Overall Scheduling Approach</td>
<td>37</td>
</tr>
<tr>
<td>4.3.1</td>
<td>BSA2 Approach</td>
<td>38</td>
</tr>
<tr>
<td>5</td>
<td>Cache Analysis</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>Data Flow Analysis</td>
<td>41</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Terminology</td>
<td>42</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Transfer Functions</td>
<td>43</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Control-Flow Constraints</td>
<td>44</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Gen-Kill Form</td>
<td>44</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Simple Data Flow Problem</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td>Direct Mapped Cache</td>
<td>48</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Predictable Memory Access</td>
<td>48</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Unpredictable Memory Access</td>
<td>52</td>
</tr>
<tr>
<td>5.3</td>
<td>Set Associative Cache</td>
<td>54</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Cache Set Model</td>
<td>54</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Refined Approach</td>
<td>55</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Universal Set</td>
<td>60</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Analysis Example</td>
<td>62</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Problems with FIFO and LFU</td>
<td>64</td>
</tr>
<tr>
<td>5.4</td>
<td>Verification of Cache Analysis</td>
<td>67</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Data Flow Analysis Framework</td>
<td>67</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Partial Order for Lattice</td>
<td>68</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Further Requirements</td>
<td>71</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Precision of Solution</td>
<td>72</td>
</tr>
<tr>
<td>5.4.5</td>
<td>Analysis Properties</td>
<td>73</td>
</tr>
<tr>
<td>5.4.6</td>
<td>Proof for Direct Mapped Cache</td>
<td>74</td>
</tr>
</tbody>
</table>
A.1.1 Fix on Windows . . . . . . . . . . . . . . . . . . . . 114
A.2 Build on Linux . . . . . . . . . . . . . . . . . . . . . . . 115
A.3 Build on Windows . . . . . . . . . . . . . . . . . . . . . 115
Chapter 1

Introduction

Today the market share for embedded systems among all kind of systems with microprocessors is around 99% [1]. This leaves only around 1% for general purpose systems, like personal computers. Embedded systems are dedicated to a few specific purposes, with the aim to perform these functions very efficiently. Many, if not most, embedded systems are real-time systems as well, as illustrated in Figure 1.1.

A real-time system [1] is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations but also on the time when these results are produced. If these systems fail to meet the time constraints, often called deadlines, it could be catastrophic. Real-time systems are frequently found in the industry among

![Figure 1.1: Most embedded systems are real-time systems, and vice versa.](image)
1.1 Background

Figure 1.2: Differences between hard, firm and soft real-time systems.

process control systems, automotive electronics, medical equipments, multimedia and telecommunications.

Real-time systems can be divided into hard, firm or soft. Figure 1.2 illustrates their differences in terms of value provided when their deadlines are violated. In words, hard deadlines must be met, otherwise it results in a catastrophic condition. Failure to meet firm deadlines results in computational work that has no value anymore. Violation of soft deadlines merely degrades the real value of the computational work with time. This thesis reflects the need to construct analysis approaches for hard real-time systems, with focus on cache analysis for multi-processor systems-on-chip (MPSoC).

1.1 Background

Static timing analysis (or formal timing analysis) is a concept used to formally predict and calculate the worst case execution time (WCET) for a given task, or set of tasks. The actual calculation and scheduling is made offline and requires detailed prior knowledge about the tasks. This analysis
The approach is essential for hard real-time systems, since it is the only way to guarantee WCETs. A good analysis approach generates tight WCETs that are conservative. For soft and firm real-time systems, the formal reasoning can be relaxed.

The emergence of today’s advanced processors and multi-processor systems has made the design space much larger and at the same time it is more difficult to guarantee tight response times. For example, when selecting a processor with pipelining, the purpose is that it will speed up the executing tasks compared to an equivalent processor without pipelining. To be safe, the engineer could assume that this new processor does not generate any execution boost for a given set of tasks, but this would not be economically defensible. When a faster (and more expensive) processor is selected, the benefits must be paid off in a guaranteed decreased WCET.

A cache memory is a hardware component that enhances the performance of memory accesses significantly in the average case. A memory access that is served directly from the cache is called a cache hit, in contrast to a cache miss for which the ordinary main memory also is involved. Without a proper analysis of the feasible execution paths, the involved instructions and the prediction of cache hits and misses, there is still no guarantee of any performance boost with regard to WCET. In that case, it is probably better to stick to a system without a cache. However, caches are nowadays so commonly used that a proper cache analysis is essential for calculation of WCETs.

With the advent of multi-processor systems, it has been increasingly difficult to guarantee timing constraints since they have yet another unwieldy property. Most MPSoC need to communicate between tasks to be able to transfer data and spread out the computation work. A very common communication technique is to use a shared memory that any task on any processor can read from and write to. The access to the shared memory is often carried out by a single bus, which makes the communication infrastructure cheap and simple. The problem is that only one single communication event is allowed at a given time and, hence, the bandwidth is shared by all nodes. Therefore, collisions must be avoided and a task may need to wait for the bus before the requested access can complete.
1.2 Problem and Objectives

Static timing analysis for hard real-time systems is a hot field in computer science. There exist a large amount of embedded systems that depend on this property. As already noted, the complexity of today’s hardware components also requires new or adjusted approaches to accurately model these architectures.

ESLAB at Linköping University has recently invented models to statically calculate the WCET for tasks running on MPSoC. As multi-processor architectures get increasingly more popular, this new analysis approach is very welcome. The approach is using an analysis tool called SymTA/P (Symbolic Timing Analysis for Processes), constructed at University of Braunschweig. However, this tool is designed for mono-processor systems and therefore some of its analysis modules need to be adjusted or replaced. The general goals of this thesis are:

1. A cache analysis designed to work for MPSoC. Previous solutions have not been targeted for MPSoC. The cache analysis should be a C++ software module that integrates into SymTA/P. Furthermore, it should be compatible with both mono- and multi-processor systems.

2. To construct facilities for future research around how different cache miss patterns contribute to the overall WCET of tasks running on MPSoC. For this purpose, a task generation program needs to be designed and implemented. The program shall construct tasks according to requested flow complexity and cache miss characteristics.

The research for this thesis has been restricted to a few of the most prominent approaches in static timing analysis. The next section briefly introduces these sources. Difficulties imposed by other hardware architectures, like pipelining and superscaling, have not been considered.

1.3 Methods and Sources

For this thesis, it has been necessary to gather knowledge from the current approaches in the field of static timing analysis. This knowledge base has
been essential in the transition to MPSoC. In addition, without a proper theoretical foundation to rely on, any implementation would be risky. The study can be divided into the following categories:

- Li et al at University of Princeton introduced a concept called Implicit Path Enumeration [2][3][4][5]. This approach has been very useful and is used as the basis for other models.

- Various contributors from University of Braunschweig have enhanced and tightened the WCET calculation of tasks by the invention of program segments [6][7][8][9][10] and a data flow based cache analysis approach [11][12][13].

- ESLAB at Linköping University [14][15] has invented bus schedule algorithms to safely produce system schedules and corresponding WCET bounds for MPSoC.

- Data flow analysis is well established and useful for a range of data flow problems in compilers [16] and other tools. This solid framework proved to be useful for our cache analysis approach and the verification of the designed algorithms.

Furthermore, the source code in SymTA/P has been investigated. In particular, the code performing cache analysis for mono-processor systems has been reviewed. This has given some insight on how their theories and algorithms have been implemented, and also the evolution after the corresponding publications were released.

The implementation of the task generation tool also required some refreshment from the mathematical discipline of probability theory, in particular discrete distributions of random variables. A method known as the alias method [17] is used to generate distributions of cache misses.

1.4 Structure

The thesis starts with refreshment on some theories that are used as the foundation of cache prediction, static timing analysis and randomized task
1.4. Structure

generation. In Chapter 3, background material and related work on execution time analysis for mono-processor systems are provided. The reasons why these approaches cannot be reused on MPSoC are discussed in Chapter 4, and a solution to these inherent problems is briefly presented.

The rest of this thesis elaborates on the work done for cache prediction on hard real-time systems and randomized task generation. In Chapter 5, two different approaches for cache analysis are established and verified. The chapter ends with a formal proof on the correctness, convergence and precision of the designed algorithms. The design of the task generation tool, used to construct tasks with different flow complexity and cache miss characteristics, is described in Chapter 6. The implementations and usage of the constructed tools are presented in Chapter 7. The end of this chapter shows how the cache analysis fits into a WCET analysis framework. In the appendix, information about how to compile these tools for different platforms can be found.
Chapter 2

Theory Revisited

This chapter introduces the foundation of theories coupled to topics within this report, such as static timing analysis, cache analysis and task generation. These theories will frequently be referred to in subsequent chapters without further explanation. Its purpose is to give the reader refreshment in these areas, but could be skipped if basic knowledge in each of the sections below already has been obtained. Furthermore, the basic principles of common processor architectures, like pipelines, are assumed to already be known.

2.1 Cache Memories

Almost every contemporary computer system today has some kind of cache memory included. Cache memories are much smaller, but also much faster than main memories and are therefore used to speed up the performance of computers. A cache memory complements the main memory in a system by making it to appear faster than it is in reality. The running programs have no knowledge about the existence of the cache, but the cache gives the feel of a large and fast main memory. Once a memory block is brought into the cache, subsequent memory accesses will be served by the fast cache memory alone. The average access time in this kind of memory hierarchy
2.1. Cache Memories

can be expressed as follows [18]:

\[
T_{access} \approx P_{hit} \times T_{cache} + (1 - P_{hit}) \times (T_{cache} + T_{mm}) \times CB_{size} + T_{check} = T_{cache} + (1 - P_{hit}) \times T_{mm} \times CB_{size} + T_{check}
\]

where \(T_{access}\) is the average access time, \(P_{hit}\) is the probability of a cache hit, \(T_{cache}\) is the cache access time, \(T_{mm}\) is the main memory access time, \(CB_{size}\) is the number of words in a cache block and \(T_{check}\) is the time needed to check if the requested data are in the cache.

The cache access efficiency is defined as \(\frac{T_{cache}}{T_{access}}\) [18]. A hit rate \(P_{hit}\) of 98% could be achievable if there is a high degree of locality in the programs that are being executed. A good cache access efficiency implies a good hit rate. According to the above formula it is easy to see that a cache memory provides a substantial improvement of memory access time and is therefore a component that cannot be neglected in static timing analysis of hard real-time systems.

Cache memories are designed around a principle called locality of reference. According to Stallings [19], this principle states that the memory references tend to cluster. When observing memory accesses over a short period of time, the processor is primarily working with the same fixed clusters of memory references. The principle is normally divided into spatial locality and temporal locality. Spatial locality refers to the tendency of execution to refer to memory addresses in clusters. Temporal locality refers to the tendency of execution to access the same memory address again soon. Both code and data accesses normally have both the spatial and temporal property and therefore benefit from this principle. Examples of such program properties are [18]:

- Sequential instruction storage.
- Loops, iterations and sub-routines.
- Sequential data storage (e.g. arrays).

A cache memory can be split into one dedicated instruction and one data cache. It can also be combined into a single unified cache memory.
There exist three kinds of organizations that can be used in a cache memory: direct mapped, associative mapped and set associative mapped. The names of these organizations originate from the mapping function. This mapping function maps a memory block into a cache line (holding the memory block). Because the main memory is larger than the cache memory, several different main memory blocks can be mapped into the same cache line. Consequently, during a memory access, there is a need for the cache memory to decide which main memory block is currently mapped to a certain cache line.

The simplest, and also the most inexpensive, cache memory is the direct mapped, illustrated in Figure 2.1. This technique provides a unique mapping between a main memory block and a cache line. The disadvantage, however, is that every memory block has a fixed cache line. Thus, if two different memory blocks, mapped to the same cache line, are used repeatedly by a program, these memory blocks will be swapped in and out continually. This results in a low hit rate.

The problem with direct mapping is solved in the associative mapped cache memory. The referenced memory block is not mapped to a specific cache line, but any cache line in the cache memory could potentially be
used for holding the memory block. The hardware is considerably more expensive in this case, since it must simultaneously check every cache line for a match during a memory access. Associative mapping requires a replacement algorithm that is used to decide which cache line to replace when a new memory block is read into the cache.

A compromise between direct mapping and associative mapping is the set associative mapped cache memory, illustrated in Figure 2.2. This kind of memory exhibits the strengths of the previously described cache memories, while reducing their disadvantages. The memory is divided into sets, where each set is holding \( k \) lines. For example, a 4-way set associative cache memory holds 4 lines in each set. Like the direct mapped cache memory, there is a unique mapping between a certain memory block and a cache set. Unlike the direct mapped cache, each cache set holds a multiple of cache lines. Any cache line in the mapped set can be used to hold a referenced memory block. Since the cache set is limited to hold \( k \) lines, the hardware needs only to check \( k \) lines for a match during a memory access. Obviously, set associative mapping requires a replacement algorithm as well.

As noted, both an associative mapped and set associative mapped cache memory requires a replacement algorithm. These replacement algorithms
must decide which cache line to replace within a cache set. They should be designed to select a cache line for replacement that will not be used again in the near future, with the goal to maximize the number of cache hits. The four most common replacement algorithms are least recently used (LRU), first-in-first-out (FIFO), least frequently used (LFU) and randomized replacement. During a cache miss, LRU replaces the line in the cache (set) that has not been referenced for the longest time. LRU is easily implemented in hardware and normally provides a very good hit ratio. Another advantage is that LRU is predictable and easily modeled in cache analysis.

2.2 Control Flow Graph

A basic block is a unit of sequential instructions. The concept is relied on in static timing analysis and compiler design as well. Informally, a basic block [16] is an instruction sequence where the control proceeds sequentially from one instruction to the next. This implies that a basic block is a segment of code which is entered only at the first statement and left only at the last statement. A basic block is constructed by putting the first instruction of a segment into it and keep adding instructions until we meet a jump, a conditional jump, or a label on the following instruction. The decomposition of a code sequence into basic blocks is illustrated in Figure 2.3(a).

A control flow graph (CFG), or basic block graph, models the possible control flow in a task, as is illustrated in Figure 2.3(b). Each node represents a basic block, and each directed edge represents the possible flow between two basic blocks. Every feasible path within a task can be enumerated with the help of a CFG. It is, however, quite common that a CFG enumerates infeasible paths as well, since other program constraints are not captured by the graph.

2.3 Task Graph

An application can be modeled as a set of tasks, which correspond to different parts of the system or application. Each node in a task graph
represents a task, which is a unit of functionality activated as a response to a certain input and which generates a certain output. Each directed edge represents a precedence constraint due to data dependencies between the tasks [1]. This normally forces some tasks to run before others. A possible task graph is illustrated in Figure 2.4.
2.4 Partial Sets and Lattices

At the foundation of cache analysis described in Chapter 5, partial orders and lattices are two important concepts. Intuitively, a partial order defines an order between elements in a given set. Every element pair in the set does not need to be related, thus the name partial order. If every element pair is related, it is called a total order. The required properties for a set to be partially ordered are reflexive, antisymmetric and transitive [20]. It turns out that the common mathematical relation $\leq$ is defining a total order for integers.

Another important construct used in formal cache analysis is lattices. According to Turesson [20], we must first establish a few definitions. Let $(A, \preceq)$ be a po-set, $B \subseteq A$, and $a \in A$. Then $a$ is:

1. a lower bound to $B$ if $a \preceq x$ for all $x \in B$,
2. a greater bound to $B$ if $x \preceq a$ for all $x \in B$,
3. a greatest lower bound to $B$, $\text{glb } B$, if $a$ is a lower bound to $B$ and $x \preceq a$ for all lower bounds $x$ to $B$,
4. a least upper bound to $B$, $\text{lub } B$, if $a$ is a greater bound to $B$ and $a \preceq x$ for all greater bounds $x$ to $B$,

The above definitions imply that there only can be one glb- and lub-element for a given subset $B$ of a po-set $A$. Now we can continue with the definition of a lattice: A po-set is also a lattice if $\text{glb } \{x, y\}$ and $\text{lub } \{x, y\}$ exist for every $x, y \in A$. Intuitively, a semi-lattice is a po-set where only one of the two conditions needs to be true.

As soon as any two elements in a po-set $A$ do not have a common lower or upper bound on themselves, the po-set cannot be a lattice. It is easy to see if a po-set is a lattice by drawing a Hasse diagram [20] according to the relations in the po-set. An example of such a diagram is found in Figure 2.5, which illustrates 15 partitions of the set $\{1, 2, 3, 4\}$. The partial order is here defined by refinement, where a finer partition is less than a coarser partition. There is no need to print directions on the edges, since it is assumed that the edges start at a lesser element and end at a greater one.
2.5 Integer Linear Programming

Static timing analysis described in Chapter 3 builds on Integer Linear Programming (ILP). This thesis does not require an understanding of how ILP-problems can be solved, only how to set them up with mathematical notations. The interested reader that wants to know how to solve ILP-problems may refer to Holmberg [21].

The goal of linear programming is to optimize a linear objective function with certain requirements, called constraints. The objective function can either be maximized or minimized, depending on the nature of the problem. The calculation of the WCET for a task is a typical maximizing problem. The number of variables and constraints also depends on the problem we are trying to model. In general, a linear objective function looks like this:

\[ f(x_1, x_2, ..., x_n) = a_1x_1 + a_2x_2 + ... + a_nx_n + b \]

with constraints in the form:

\[ a_1x_1 + a_2x_2 + ... + a_nx_n \leq b \]

ILP-problems have much in common with LP-problems. The real differ-
ence is that all the variables in an ILP-problem are required to be integers, no continuous variables are allowed. WCET calculations are often ILP-problems, since the number of executions of each basic block is an integer.

2.6 Probability Theory

The task generation method described in Chapter 6 requires a few definitions from probability theory. We will now present the definitions of discrete random variables, probability function, expectation and standard deviation [22].

Assume that we have a sample space $S$ that is a collection of all possible outcomes $s$ of an experiment. An event is a subset of the sample space. Every element of $S$ can be mapped to a real value. Let $X$ be a real valued function that maps $S$ to $\mathbb{R}$. Thus, if $s \in S$, then $X(s)$ is a real number. In Chapter 6, we will use discrete random variables, since our random variables will only map to a finite number of values (otherwise a continuous random variable would be necessary).

A probability function $p_X$ simply defines the probability that the associated random variable will yield a certain value. In mathematical notation, $p_X(x) = \Pr\{X = x\}$.

Expectation is closely related to the probability function of a random variable. The expectation is the average value that the random variable will give us in the long run, and is denoted $E[X]$ or $\mu$. It is defined as:

$$E[X] = \sum_i x_i p_X(x_i)$$

Finally, standard deviation is a measure of how much the possible randomized values are distributed relative to the average value $\mu$. Standard deviation is the square root of variance, which is defined as:

$$Var[X] = E[(X - E[X])^2] = E[(X - \mu)^2] = \sum_i (x_i - \mu)^2 p_X(x_i)$$
2.6. Probability Theory
Chapter 3

WCET on Mono-Processor Systems

This chapter will provide a review of some research conducted in formal timing analysis for mono-processor systems between 1995-2002. Today, these theories have settled and focus has shifted to timing analysis for multi-processor systems-on-chip (MPSoC), which will be reviewed in Chapter 4. The purpose of this chapter is to present some background material that is necessary before the transition to MPSoC.

Hard real-time systems have strict requirements to meet their deadlines. If they do not, a catastrophic failure could be the result. To verify if a task meets its deadline, a bound on the execution time needs to be determined. Obviously, the execution time will be influenced not only by the instruction sequence of the task, but also by the specific processor architecture the task is running on. Traditionally, the following are assumed during WCET analysis [23] of a task:

1. One task is analyzed in isolation.
2. There exist no interfering background activities.
3. No task switches or interrupts occur during task execution.
3.1 Measurement

There are two main methods used to calculate the WCET for a given task. The simplest, but also the most fragile one, is based on measuring. This method measures all executions of a task from every possible program input, more shortly called exhaustive testing. By taking the maximum time value from all these execution times, we would get the actual and optimum WCET of a task. One big issue with this approach is that in practice, the engineer does not have time to wait for the exhaustive testing to finish. Even the most simple program that takes a 4-bytes integer parameter, with no other process state to start with, requires $2^{32}$ different runs.

Although measuring has its inherent problems, it is still being used in the industry today. The engineer is forced to take educated guesses to enumerate potential program states and inputs that might contribute to the WCET. The analyzed task is then executed once for each enumerated program state and input. Finally, the WCET is calculated by adding a safety margin to the longest run. As already pointed out, the engineer cannot be sure that the worst case path really has been found in any of the measurement runs, since tests of non-trivial programs cannot be exhaustive in practice. This problem is illustrated in Figure 3.1.

3.2 Static Timing Analysis

The other method used to calculate the WCET for a given task is called static timing analysis. This approach is inherently conservative and safe, meaning that the determined WCET is guaranteed to be equal or longer than the actual one. The challenge is, however, to get a WCET that is...
not too pessimistic, otherwise it would not be very useful. If the WCET is not tight enough, the engineer might be forced to select a faster and more expensive CPU than otherwise would be necessary.

There are a number of challenges that must be resolved for static timing analysis to be successful. It is obvious that the program itself restricts the execution time by putting constraints on the possible control flow and input parameters passed for each invocation. Furthermore, the hardware used, on which the program runs, also needs to be considered. Hardware with features to boost performance, such as CPU pipelining, out-of-order execution, and cache memories, often makes the formal timing analysis much harder. This distinction is also made by Li et al [2], who divides the analysis into program path analysis and micro-architectural modeling.

We will now provide an overview of a few of the most prominent solutions within this field. The review will hopefully show how the cache analysis in Chapter 5 fits into the broader perspective. Also, it will serve as a background to the presentation of WCET analysis for MPSoC in Chapter 4.

### 3.3 Early Work

In general, it is an undecidable problem (unsolvable) to determine the worst execution path of a program. It is accepted that this problem is equivalent to the halting problem in computability theory. Li et al [2] states that the following restrictions on programs have been suggested that make the problem decidable:

- Dynamic data structures, such as pointers to objects, are not allowed.
• Function recursions cannot be allowed.

• The loops must be bounded.

There have been many approaches to define these restrictions for a program. Annotation of the source code is a common way to bound loops in programs, as well as other false paths that will never execute. Another early approach was to use a language with regular expressions [25] to model the feasible paths in a task.

3.4 Implicit Path Enumeration

Li et al [2] improved formal timing analysis considerably with help of a technique called implicit path enumeration. As the name suggests, all feasible execution paths are implicitly considered in this approach. This is accomplished by transferring the problem of determining the bounds into an ILP-formulation. This approach is always conservative. Furthermore, the more constraints the engineer puts into the ILP-formulation, the closer solution compared to the actual WCET is returned.

The basic idea is to determine the WCET and not necessarily identify the extreme case paths (as in explicit path enumeration). Assume that there are $N$ basic blocks in a program. Furthermore, let $x_i$ be the number of times basic block $B_i$ is executed when the task takes the maximum time to complete, and $c_i$ be the worst case time cost of this block $B_i$. It is important to note that $c_i$ must be a safe constant for any basic block execution. With this notation, we can maximize the following objective function to get the WCET of a task:

$$\text{Total execution time} = \sum_{i=1}^{N} c_i x_i$$

To be able to solve $x_i$ we need to add constraints to the problem. These structural and functional constraints are depending on both the task structure and intrinsic task functionality.
3.4.1 Program Constraints

The structural constraints can be deduced from the task’s control flow graph (CFG). We associate each basic block $B_i$ node a corresponding variable $x_i$, representing the number of times block $B_i$ is executed. We also associate each control flow edge a corresponding variable $d_i$, representing the number of times the control flow is taken. Figure 3.2 illustrates a CFG with these added variables.

All flow graphs have a fundamental property. It says that the incoming flow must be equal to the outgoing flow, for every node except the entry and exit. This formulation helps us to extract the structural constraints of a task. The node where the execution starts at must have an incoming flow count set to one, that is $d_1 = 1$. The execution count $x_i$ of basic block $B_i$ is equal to the sum of control flow arriving at $B_i$, as well as the sum of control flow leaving $B_i$.

$$x_i = \sum_{d_j \in \text{inflow}(x_i)} d_j = \sum_{d_k \in \text{outflow}(x_i)} d_k$$

The functional constraints cannot, in general, be deduced by a tool and must be provided by the programmer manually. The programmer should provide loop bounds as a minimum, but a closer look at the source code can give additional information about the dependency between different basic block executions. This dependency can be formulated into constraints to further tighten the WCET. According to the CFG in Figure 3.2, the following constraints can be noted (the first eight are the structural constraints, and the last two are the functional constraints):

$$d_1 = 1 \quad x_5 = d_6 = d_8$$
$$x_1 = d_1 = d_2 \quad x_6 = d_7 + d_8 = d_9$$
$$x_2 = d_2 + d_9 = d_3 + d_4 \quad x_7 = d_3 = d_{10}$$
$$x_3 = d_4 = d_5 + d_6 \quad x_3 \leq 10x_1$$
$$x_4 = d_5 = d_7 \quad x_4 = x_5$$
3.4.2 ILP Solver

When all of the structural and enough of the functional constraints have been gathered, they are passed to an ILP-solver (a free solver is lp_solve) together with the objective function. For maximization problems, the response will be an estimated WCET together with solutions for each $x_i$, which represents the execution count for each basic block in the task. Even if there exist multiple execution paths that results in the same maximum time, only one implicit path is returned.

Since every basic block $B_i$ is associated with a constant time cost $c_i$, these time costs have to be very conservative when modeling a system with a memory cache. To be safe, essentially every memory access must be regarded as a cache miss when calculating these basic block costs in a naive way. This assumption is simply too pessimistic and must be handled.

3.5 Direct Mapped Instruction Cache

Li et al [3] proposed in 1995 a refined ILP-formulation that also considers direct mapped instruction caches when calculating the execution time bound of a task. To incorporate instruction cache analysis into the ILP-
formulation, the objective function needs to be modified and a new set of cache constraints need to be added. These cache constraints represent the cache behavior.

3.5.1 Terminology

A basic block has previously been an atomic structure. We must now divide each basic block into line-blocks, or simply l-blocks. An l-block is defined as a contiguous sequence of instructions within the same basic block that are mapped to the same line in the instruction cache. All instructions within an l-block will always have the same execution count, since l-blocks are a subset of a basic block. The cache hit and miss costs for l-blocks are constants.

Figure 3.3(a) shows an example with three basic blocks. Suppose that the instruction cache has four lines. If the starting address of each l-block can be determined from the executable code, we can find all the cache lines the instruction addresses map to and add an entry on these cache lines in a cache table, as in Figure 3.3(b). The boundary of each l-block is shown by the solid line rectangle. A basic block $B_i$ is partitioned into $n_i$ l-blocks, denoted $B_{i,1}, B_{i,2}, \ldots, B_{i,n_i}$.

For any two l-blocks that map to the same cache line, they conflict with each other if the execution of one l-block will replace the cache content of the other. Otherwise, they are called non-conflicting l-blocks, e.g. $B_{1,3}$ and $B_{2,1}$ in Figure 3.3(a).
3.5.2 New Cost Function

The execution time of an l-block depends on whether it results in a cache hit or miss during each execution. Thus, the original execution count $x_i$ needs to be subdivided into execution counts of cache hits and misses for each l-block. Furthermore, the original basic block cost $c_i$ is subdivided into l-block cost for hit and miss scenarios. The total execution time is now given by:

$$\text{Total execution time} = \sum_{i}^{N} \sum_{j}^{n_i} (c_{i,j}^{\text{hit}} x_{i,j}^{\text{hit}} + c_{i,j}^{\text{miss}} x_{i,j}^{\text{miss}})$$

where $c_{i,j}^{\text{hit}}$ and $c_{i,j}^{\text{miss}}$ are the cache hit and miss costs of l-block $B_{i,j}$, and $x_{i,j}^{\text{hit}}$ and $x_{i,j}^{\text{miss}}$ are the cache hit and miss counts of l-block $B_{i,j}$. Since l-block $B_{i,j}$ is inside basic block $B_i$, its execution count is also $x_i$:

$$x_i = x_{i,j}^{\text{hit}} + x_{i,j}^{\text{miss}}, \quad 1 \leq j \leq n_i$$

This equation links the new objective function with the program structural and functionality constraints, which remain completely unchanged. More important, the cache behavior can now be specified in terms of the new variables $x_{i,j}^{\text{hit}}$ and $x_{i,j}^{\text{miss}}$.

3.5.3 Cache Constraints

The simplest cache constraints are those originating from non-conflicting l-blocks. Let us assume that only one l-block $B_{k,l}$ is mapped to a given cache line. This implies that as soon as $B_{k,l}$ is loaded into the cache, it will permanently stay there. Thus, only the first execution of $B_{k,l}$ may result in a cache miss, while all subsequent executions result in cache hits.

$$x_{k,l}^{\text{miss}} \leq 1$$

Another case is when two or more non-conflicting l-blocks map to the same cache line. This happens when a cache miss of any of these l-blocks results in loading them all into the same cache line. This is a possible scenario since a memory block, mapping to some cache line, can overlap
l-blocks from different basic blocks. For example, $B_{1,3}$ and $B_{2,1}$ in Figure 3.3(a) are non-conflicting. The sum of their cache miss counts are at most one.

$$\sum_{x_{k,l} \text{ map to same line}} x_{k,l}^{\text{miss}} \leq 1$$

If two conflicting l-blocks map to the same cache line, the situation becomes more complex. The reason is that the particular control flow decides the hit and miss counts of the l-blocks mapped to this cache line. For this purpose, Li et al [3] invented the cache conflict graph.

### 3.5.4 Cache Conflict Graph

A cache conflict graph (CCG) needs to be constructed for every cache line containing two or more conflicting l-blocks. The graph contains three different kinds of nodes, the start node $s$, the end node $e$ and a node for each l-block $B_{k,l}$ mapped to the same cache line. For every node $B_{k,l}$ there is a directed edge with source $B_{k,l}$ and destination $B_{m,n}$, if there exists a path in the CFG from basic block $B_k$ to $B_m$ without passing through any other conflicting l-blocks of the same cache line.

An example of a CCG can be found in Figure 3.4. This CCG models the l-blocks conflicting with cache line 1 in Figure 3.3. The CCG graph starts at $s$. In general, between each node transition, the task could execute some other l-blocks mapped to other cache lines. In this case, it will soon reach node $B_{1,2}$, followed by node $B_{3,2}$. Once in node $B_{3,2}$ there are three possibilities. It could go back to node $B_{1,2}$ or $B_{3,2}$ again, by passing through some l-blocks mapped to other cache lines first. The control may also go to node $e$, which ends the task.

For each edge from source $B_{i,j}$ to destination $B_{u,v}$, we assign a variable $p(i,j,u,v)$ to count the number of times the control passes through that edge, since this is also an instance of a network flow problem. At each node $B_{i,j}$, the sum of control flow arriving at this node must be equal to the sum of control flow leaving this node. Furthermore, this sum must also be equal to the execution count $x_i$ of l-block $B_{i,j}$.
3.5. Direct Mapped Instruction Cache

Figure 3.4: Cache Conflict Graph for cache line 1 in Figure 3.3.

\[ x_i = \sum_{u,v} p(u,v,i,j) = \sum_{u,v} p(i,j,u,v) \]

where \( u,v \) may also include node \( s \) or \( e \). This set of constraints are linked to the structural and functional constraints via the \( x_i \)-variables. We must also define a proper start condition for the CCG:

\[ \sum_{u,v} p(s,u,v) = 1 \]

The variable \( p(i,j,i,j) \) represents the number of times the control flows from l-block \( B_{i,j} \) to the same l-block again, without having entered any other conflicting l-blocks of the same cache line in between. The contents of l-block \( B_{i,j} \) are still in the cache every time the control flows directly back to the same l-block, and will result in a cache hit. Thus, there will be at least \( p(i,j,i,j) \) cache hits for l-block \( B_{i,j} \). Another possible reason that the execution of l-block \( B_{i,j} \) can result in a cache hit occurs when this l-block is connected to both the edges \((s,B_{i,j})\) and \((B_{i,j},e)\).

\[ p(i,j,i,j) \leq x_{i,j}^{hit} \leq p(s,i,j) + p(i,j,i,j) \]

If l-block \( B_{i,j} \) is not connected to both the edges \((s,B_{i,j})\) and \((B_{i,j},e)\), then the constraint above will collapse into:

\[ x_{i,j}^{hit} = p(i,j,i,j) \]
All these cache constraints are used to bound the cache hits and misses. They are, together with the structural and functional constraints and the new objective function, passed to the ILP solver. The ILP solver calculates the WCET, including the values of all execution count variables $x_i$, $x_{hit}^i$, $x_{miss}^{i,j}$ and $p(u,v,i,j)$. This will result in a tighter execution time than before, since we have included the cache hit and miss costs in this model. In the worst case, a CCG must be generated for each cache line.

### 3.6 Set Associativity and Data Caches

Li et al [5] formulated a solution for set associative instruction caches. The solution is based on a Cache State Transition Graph (CSTG) that models the possible states within each cache set. This graph is an extension to CCG graphs for direct mapped caches. However, much more variables need to be added to the ILP-formulation. The complexity of constructing and processing CSTG graphs for more than 2-way set associative caches can be very large. As Li points out, for an $n$-way associative cache with $m$ conflicting l-blocks, $\sum_{i=0}^{n} \frac{m!}{(m-i)!}$ states are needed in the worst case.

Li also gives an example how to put up a new cost function and additional constraints to model a direct mapped data cache. However, this can be complex even for tiny examples. It should be noted that no generic formulas or algorithms when dealing with these caches were provided. Furthermore, only a few remarks in the publication [5] deal with set associative data caches and unified caches, where instruction and data caches are combined into a single cache.

Our cache analysis approach, presented in Chapter 5, is based on data flow analysis where the computational complexity is not as large. Furthermore, the approach is also compatible on real-time MPSoC.

### 3.7 Program Segments

As processors get increasingly more complex, the precision in the original ILP-model with basic blocks is lost. Processors with pipelining, superscaling, out-of-band executions or other technologies that dramatically improve
the speed in the average case are not easy to handle in a hard real-time environment, since it is the bounding of the worst case execution time that really matters. Ernst et al [6] released in 1997 a practical solution to this, based on program segments.

The concept of program segments is an extension to basic blocks (and obviously l-blocks as well). Figure 3.5(a) shows a code sequence with two nested loops, which generates eight basic blocks in total. The code can be converted into two hierarchical program segments, as in Figure 3.5(b). A hierarchical segment can be further transformed into linear program segments, where each linear segment contains one or more basic blocks. From this moment, a program segment can be treated as an atomic structure. Figure 3.6 illustrates why program segments give better precision and more accurate worst case execution bounds, than basic blocks or l-blocks do, on a pipelined processor. The reasons can be summarized into:

1. Execution of basic blocks overlaps when pipelining and/or superscaling is active. L-blocks give even worse precision, since these blocks are often smaller than basic blocks.
2. Cache behavior is global and depends on exact instruction sequences. This is something that program segments can exploit.

3. In contrast, basic blocks often contain only a few sequential machine instructions, which increases the amount of safety margins needed to add.

Safety margins before and after each basic block are needed, since we always need to assume the worst pipeline state between any two blocks that can execute in sequence. The same kind of conservatism needs to be considered for superscaling. The execution time of an instruction increases when structural, data or control hazards occur. For example, control hazards are often the result of a conditional jump. However, because the basic block sequence inside a program segment is always known, together with information about the hardware architecture, local simulation of the program segment can give a much more accurate bounding of the execution time. Please note that safety margins between program segments still need to be considered.

For this purpose, basic block sequences need to be encapsulated into program segments [6]. The beauty of this approach is that all previous work with ILP-formulations are still valid. The cost function is also reverted to the trivial sum of basic blocks formula, given below. The cost constant $c_i$ denotes the maximum time bound for a given program segment. The time
cost is returned by program segment simulation, which must also consider
the cache behavior. Therefore, a new cache analysis approach is needed.

\[ \text{Total execution time} = \sum_{i=1}^{N} c_i x_i \]

### 3.7.1 Path Classification

The basic path classifications in a program can be divided into *single feasible path* (SFP) and *multiple feasible paths* (MFP). Most programs have program properties that always result in the execution of a single execution path within a hierarchical program segment, although the CFG alone might suggest that multiple execution paths are possible within the same program segment. In other words, a program segment that has the SFP-property always has a single execution path for any possible input pattern. Before this was discovered, the analysis approaches did not distinguish between input data dependent control flow and program structures that are only used for program structure simplification.

The real advantage with exploiting the SFP-property, is that the engineer is removed the responsibility to annotate their programs with a large number of functional constraints. These manually provided annotations require much effort to be as precise as SFP-segments, and still they do not deliver precise program segment costs (only pessimistic basic block costs). Within the SFP-segments, segment simulation will automatically select the correct instruction sequence for any input pattern.

However, most programs also contain non-SFP segments, which means that multiple execution paths are feasible within a given hierarchical program segment. Which path is actually executed depends on the program input and task state. This property is called multiple feasible paths (MFP) and is assigned to the corresponding program segments.

Obviously, methods to identify the size and property of each program segment need to be devised. Wolf et al presented something called a syntax graph [9]. According to the authors, the syntax graph was selected to cover the hierarchy of control structures. Each hierarchical node (for example, an if- or while-node) in the syntax graph needs to be annotated with the
SFP- or MFP-property depending on the input data dependency. This data dependency can be discovered by a method called *symbolic simulation* [26] on the syntax graph.

As can be noted in Figure 3.5, many consecutive basic blocks have been clustered into a single (SFP) program segment. Therefore, this technique is called *path clustering*. Program segments that have been classified with the SFP property are clustered with other neighbor SFP-segments. For the exact approach, please refer to Wolf *et al* [10].

While the classification of program segment properties has a big impact on the accuracy of the calculated WCET, formal definitions are needed to actually be able to implement the extension of basic blocks to program segments. It is outside the scope of this thesis, but the interested reader should refer to Wolf [27].

### 3.7.2 Timing Estimation

There are two major different approaches to determine the upper bound of execution time for a program segment. The simplest approach is called *instruction timing addition* (ITA). As the name suggests, the time cost for each consecutive instruction in a program segment is added until we get a total cost for it. The time cost for each instruction can be taken from a table given by the processor manufacturer.

The other approach is called *path segment simulation* (PSS). As the name suggests, the program segments are simulated and the time costs are noted. The simulation is conducted by a *cycle true processor* simulator, which means that a given hardware architecture is modeled exactly.

ITA is obviously only appropriate for very simple hardware architectures. However, ITA might be a good choice for special CISC architectures that have data dependent instruction execution times, like multiplication or memory block copy instructions. PSS cannot generally guarantee an execution time for segments with data dependent instructions, since it might use data that does not produce the worst case. On the other hand, PSS is appropriate for hardware architectures with pipelining and superscaling, since it accurately simulates these effects. Here, ITA would be inappropriate, since it cannot anticipate pipeline hazards. Remember that safety
margins must be added to the boundaries of the simulated program segments, but inside a given segment the precision is exact.

According to Staschulat [24], measurements on real hardware might be necessary, if either no cycle-true processor simulator exists for the particular hardware or it is too expensive to construct one. The program segments can be measured with *instrumentation probes* before and after each segment. The main drawback with this approach is that the probes disturb the execution time of the task, and this should be reflected.
Chapter 4

WCET on MPSoC

Traditionally, static timing analysis has only been designed for mono-processor systems. In the last few years focus has shifted to MPSoC, as these kinds of systems have become more and more common. Unfortunately, as we will see, the solutions for mono-processor systems cannot be reused for MPSoC. The implications from this chapter will be used in the cache analysis approaches described and constructed in Chapter 5, and the generation of randomized tasks in Chapter 6.

4.1 The Problem

A memory access that results in a cache miss will use the bus to fetch the data from main memory. In mono-processor systems, the time to complete this memory access by a given hardware architecture is always bound and known. This is not the case on MPSoC with shared memory and shared communication infrastructure. The total memory access time is depending on the bus contention. During periods with low bus contention, memory requests might be served directly, but during periods with high contention the wait for the bus might be arbitrary long.

For mono-processor systems, the cost of each program segment includes a total cache miss cost. This total cost can be calculated by multiplying
the number of predicted misses with a known miss cost. For a general MPSoC, each miss cost can no longer be regarded as a constant, but varies considerably depending on the bus contention. To be able to schedule tasks on different processors, the WCET of each task is needed. However, since the WCET of a task is dependent on the actual system schedule (and its resulting bus contention), it is problematic to construct such schedules.

Naive approaches to this problem have determined the WCETs of each task in isolation. These WCETs are then used in the system scheduling that takes the global view of the system. This approach is not valid on MPSoC with shared memory and a single bus. Recently, safe system scheduling [15] was provided by Rosén et al. To guarantee predictability and efficiency, their solution is to construct an optimized bus access schedule that is observing a few rules enforced by a bus access policy. A bus access schedule is used to decide which processor that has access to the bus at a certain moment.

### 4.2 Bus Access Schedule

Time Division Multiple Access (TMDA) is a common solution to share a medium (in this case a bus) between multiple users (in this case processors). The division is necessary because two processors cannot access the bus at the same time. It is important to note that the division is predictable, since at any given moment only one particular processor can have access to the bus.

Figure 4.1 illustrates a possible bus schedule for two processors. Every bus schedule contains slots with a start time and length. Each time slot is allocated to one of the available processors. A special hardware unit, called a bus arbiter, enforces the bus schedule at run-time. In practice,
this means that the arbiter will either grant a memory access or keep it waiting until the schedule allows it to be completed. The information of the bus schedule is stored in a memory that is accessed by the bus arbiter. Four different kinds of bus access schedules have been prepared, BSA1 to BSA4. Every system schedule is defined over one single application period. During an application period, all tasks in the system will run to completion. The system schedule is then repeated periodically.

- **BSA1** is designed to give the best possible WCET for the system and its tasks. Figure 4.2(a) shows that the slot sequences and slot sizes in the table are customized completely according to the need of the analyzed tasks. The use of BSA1 is, however, often not practical due to the very large bus schedule tables that would be necessary.

- **BSA2** reduces the memory requirements. Figure 4.2(b) shows that the bus schedule is divided into segments, and each segment is further divided into TDMA rounds. Every segment is associated with a start time and size. Since the size of a segment can be larger than a TDMA round, a TDMA round can be repeated several times during run-time. The TDMA round inside a segment is defined as a list of bus slots (including their size), each allocated to a specific processor.

- **BSA3** further reduces the memory requirements. The difference compared to BSA2, is that the defined bus slots in each segment must be of the same size, as in Figure 4.3. Therefore, individual slot sizes are no longer needed.

- **BSA4** removes almost all scheduling complexity. All bus slots are defined in a simple list that is repeated periodically. All time slots are also of the same size. This bus access policy will result in poor application performance.

In Chapter 3, we showed that an ILP-formulation requires a constant time cost for each program segment (or basic block). On MPSoC with a single bus, neither cache miss counters nor ILP-formulations can be used. Thanks to the bus access schedule, the cost of each program segment can be decided. Our cache analysis approaches, described in Chapter 5, will
4.2. Bus Access Schedule

Figure 4.2: Example of (a) BSA1 and (b) BSA2 schedule [15].

Figure 4.3: Example of a BSA3 schedule [15].

deliver a conservative cache hit and miss sequence in each program segment. Figure 4.4 illustrates a possible memory access sequence within a program segment. For each hit, the memory request can be served directly with a known cost. However, for each miss, the bus access schedule needs to be observed. In that case, the maximum time cost to wait for the memory request to be completed is calculated.
Figure 4.4: Cache hit (H) and miss (M) sequence inside a program segment running on CPU1.

### 4.3 Overall Scheduling Approach

In this section, the overall approach to the system scheduling of tasks is depicted, which will produce a schedule with time bounds that can be guaranteed at run-time. A wise mapping of the tasks to the available processors must have been performed in advance. This task mapping must consider the dependencies between tasks, which can be extracted from the task graph.

The first thing the engineer has to do, is to decide which bus access policy to use. Normally, this is a choice between BSA2 or BSA3 depending on the memory requirements of the produced bus schedules. The selected bus access policy is then enforced during calculation of the WCET.

The algorithm is outlined in algorithm 1. The outer loop in the algorithm is based on a list scheduling [1] technique, which decides the set of tasks that are active and which new tasks to schedule at the current time $t$. At line 6, the inner loop in the algorithm calculates the most efficient bus segment schedule, according to the selected BSA-policy. During the WCET estimation process, the bus segment schedules determined during the previous iterations are considered for the time intervals before $t$, and the new bus segment candidate $B$ for the time interval after $t$. Once a segment configuration $B$ is decided on, $\theta$ is the earliest time a task in the task set terminates. The segment configuration $B$ is now fixed for the time interval $(t, \theta]$, and the scheduling process will in the next outer loop iteration continue at time $\theta$. We will outline how a bus segment schedule candidate is calculated inside the inner loop in Section 4.3.1.

The WCETs for each task that are delivered by the bus schedule algorithms are always safe, even when the tasks execute less than their WCETs.
Input: Mapped tasks and a bus access policy (BSA1-BSA4).
Output: A system schedule for the tasks.

\[ \theta = 0 \]

while not all tasks scheduled do

Schedule new task at \( t \geq \theta \).

\( \Psi \) = set of all tasks that are active at time \( t \).

repeat

Select bus schedule \( B \) for the time interval starting at \( t \).

Determine the WCET of all tasks in \( \Psi \).

until termination condition

\( \theta \) = earliest time a task in \( \Psi \) finishes.

end

Algorithm 1: Overall approach to calculate a bus schedule.

This has been formally demonstrated by Andrei et al [14], and the intuition behind it is the following:

1. Instruction sequences completed earlier than predicted by the worst-case analysis cannot produce violations of the WCET.

2. Cache misses (or other bus requests) that occur earlier than predicted in the worst case will, possibly, be served by an earlier bus slot, but never by a later one than considered during the WCET analysis. Furthermore, this cache miss will not adversely affect any other processors, due to the fact that the bus slots are assigned exclusively to processors.

3. A memory access that results in a hit, although predicted as a potential miss during the cache analysis, will obviously not produce a WCET violation.

4.3.1 BSA2 Approach

The bus schedule optimization is performed using heuristics. This section will describe the algorithm. The BSA2 scheduling, defined in Algorithm 2, represents the inner optimization loop of the overall approach. A set of
Input: Set of active tasks $\Psi$ at time $\theta$.
Output: Optimized bus schedule $B$ for $t \geq \theta$, until next task finish.

1. Calculate initial slot sizes.
2. repeat
3. Calculate an initial slot order.
4. repeat
5. Analyze the WCET of each task $\tau \in \Psi$ and evaluate the result according to the cost function.
6. Generate a new slot order candidate.
7. until all candidates are evaluated
8. Generate a new slot size candidate.
9. until exit condition
10. Select the best configuration according to the cost function.

Algorithm 2: Bus scheduling with BSA2.

active tasks, $\tau_i \in \Psi$, at time $\theta$, are passed to the optimization loop. We want to find an optimized bus segment $B$ for the active tasks, with the goal to minimize the global application delay. This global delay includes tasks not yet active and for which no bus schedule has been defined.

The calculation of the global delay must take into account the future tasks. The future tasks are approximated by calculating their delay when no bus conflicts occur, called naive WCET. Let tail $\lambda_i$ for task $\tau_i \in \Psi$ denote the longest chain of tasks that are affected by the execution time of $\tau_i$. This chain includes dependency tasks to $\tau_i$ and subsequent tasks mapped to the same processor as $\tau_i$. The dependency tasks can be extracted from the task graph. The approximated global delay (defined as a cost function) can be expressed as:

$$ C_{\Psi, \theta} = \max_{\tau_i \in \Psi}(\theta + WCET_{\theta}^i + \lambda_i) $$

where $WCET_{\theta}^i$ is the WCET of the active portion of the task $\tau_i$, which is executed after time $\theta$, and $\lambda_i$ is the approximated global delay of all its dependency tasks. A smaller cost function value obviously denotes a smaller approximated global delay. Therefore, the bus schedule that generates the smallest cost function value is selected. For details how to generate good
candidates for the slot order and slot sizes of a bus segment, please refer to Rosén et al [15].
Chapter 5

Cache Analysis

Cache analysis is a very important topic in static timing analysis. Without a proper analysis considering the cache, either the calculated WCET would be too conservative in the best case, or not even safe in the worst case. The problem with a too conservative WCET is that hardware with better performance than necessary is required to be used, and therefore the whole system would become more expensive.

In Chapter 3, we presented how direct mapped instruction caches for mono-processor systems can be modeled, which is possible to extend for set associativity and data caches. However, the ILP-formulations tend to be very complex with a large number of variables even for small tasks. Furthermore, the cache model in that approach is not even compatible with the extension to program segments, or safe on multi-processor systems. Even for the same number of cache hits and misses, the WCET on MPSoC can differ for different cache miss and hit sequences. In the following, we will present an approach based on data flow analysis.

5.1 Data Flow Analysis

Generally speaking, data flow analysis is a method used to derive information from a control flow graph (CFG), containing basic blocks (or program
segments) as nodes and directed edges between the nodes representing the control flow. This analysis approach is primarily used in compilers to optimize the programs being compiled. Optimization problems like reaching definitions, live variables and available expressions are all based on this technique.

Following an execution path in a program, each instruction potentially transforms the task state in some way. The task state before the execution of an instruction serves as input, and the state after the execution serves as output. The task state can include variables, registers, stack, heap and even the cache, as we soon will see. It is important to note that the data flow analysis includes all potential paths through the CFG, even those that never can be executed. When the analysis is done, the calculated task state can be extracted at each instruction point in the CFG.

As already pointed out, CFGs can potentially include an infinite number of execution paths and paths with infinite length, for example in CFGs containing loops. Therefore, data flow analysis sometimes needs to abstract all possible process states at a particular point in a safe and finite way [16]. How this information is abstracted and processed is specific to the particular problem we want to solve.

5.1.1 Terminology

First, we need to introduce some basic definitions from the theory of data flow analysis.

- A program point points either to a place before or after some instruction inside a program segment (or basic block).
- A data flow value represents an abstract task state at a specific program point.
- All possible data flow values represent the domain for a given task and problem.
- The data flow value before and after each statement \( s \) is called \( in[s] \) and \( out[s] \), respectively. Accordingly, the values before and after each basic block are called \( in[B] \) and \( out[B] \), respectively. Finally,
The values before and after each program segment are called \( \text{in}[PrS] \) and \( \text{out}[PrS] \), respectively.

- The data flow problem is to find solutions to \( \text{in}[s] \) and \( \text{out}[s] \), for each \( s \), according to a set of constraints. The solutions to \( \text{in}[B] \), \( \text{in}[PrS] \), \( \text{out}[B] \) and \( \text{out}[PrS] \) can be found among \( \text{in}[s] \) and \( \text{out}[s] \). The set of constraints are those defined by the transfer functions and control flow of the task, see Section 5.1.2 and 5.1.3.

- All data flow problems are using CFGs as the primary input. Every CFG has two additional nodes beyond its program segments (or basic blocks), an entry node and an exit node, which represent the start and exit of the task.

### 5.1.2 Transfer Functions

The data flow values before and after a statement are constrained by the semantics of the statement, including the specific problem to solve. In general, transfer functions can be defined to propagate information either forwards or backwards along execution paths. For cache analysis, only forward propagation is needed. A transfer function for a statement \( s \), denoted \( f_s \), takes as input the data flow value before the statement, and the result represents the data flow value after the statement.

\[
\text{out}[s] = f_s(\text{in}[s])
\]

To simplify the data flow analysis or reduce the time to solve the particular problem, transfer functions at the level of program segments can be defined. We know that linear program segments contain consecutive statements always being executed in a strict forward order. Therefore, the transfer function for a program segment, \( f_{PrS} \), can be defined by composing the transfer function of each statement, \( f_{s_1}, f_{s_2}, \ldots f_{s_n} \), inside a segment.

\[
f_{PrS} = f_{s_n} \circ f_{s_{n-1}} \circ \cdots \circ f_{s_1}
\]

\[
\text{out}[PrS] = f_{PrS}(\text{in}[PrS])
\]
5.1.3 Control-Flow Constraints

The data flow values are also constrained by the flow control of the task. Inside a program segment containing the statements \( s_1, s_2, \ldots, s_n \), the value out of statement \( s_i \) is the same value flowing into statement \( s_{i+1} \).

\[
in[s_{i+1}] = out[s_i], \quad \forall i = 1, 2, \ldots, n - 1
\]

The control flow between the program segments is defined by directed edges between them. This flow is also considered as a constraint on how the data flow information can propagate. As already noted, many data flow problems need to propagate information forwards along execution paths. Therefore, some type of forward-flow problems can be defined as:

\[
in[PrS] = \bigcup_{P \in \text{pred}(PrS)} out[P]
\]

where \( P \) is a predecessor of program segment \( PrS \). Please note that \( \bigcup \) is one possible meet operator. Another common meet operator is \( \bigcap \). Regardless of the specific kind of meet operator used, a meet operator is always used to summarize the data flow values meeting at a point, in a conservative way.

It is important to note that the solutions to the data flow equations, in general, are not unique. Therefore, the goal is to find the most precise solution that satisfies the transfer functions and control flow of the task. However, a solution must always be conservative, meaning that those approximations that change what the program actually computes cannot be allowed. In the case of cache analysis, approximations that result in a cache hit at a certain program point, when a cache miss actually can occur at execution, are not allowed.

5.1.4 Gen-Kill Form

Many data flow problems can be solved by first computing gen- and kill- sets [16]. We will demonstrate how these sets can be used to solve a classical data flow problem in Section 5.1.5. This gen-kill approach will later be used to predict cache misses for direct mapped caches.
A data flow problem can often be formulated by observing that each statement generates and/or kills data flow values. Thus, the transfer function for statements can be defined as:

\[ f_s(x) = \text{gen}_s \cup (x - \text{kill}_s) \]

The beauty of the gen-kill form comes from the fact that transfer functions for complete program segments also have the same form. To see this, first observe how the transfer functions, \( f_1(x) \) and \( f_2(x) \), of two statements are composed:

\[
\begin{align*}
  f_2(f_1(x)) &= \text{gen}_2 \cup (\text{gen}_1 \cup (x - \text{kill}_1) - \text{kill}_2) = \\
  &= \text{gen}_2 \cup (\text{gen}_1 - \text{kill}_2) \cup (x - \text{kill}_1 - \text{kill}_2) = \\
  &= (\text{gen}_2 \cup (\text{gen}_1 - \text{kill}_2)) \cup (x - (\text{kill}_1 \cup \text{kill}_2))
\end{align*}
\]

We can extend this idea to compose the transfer functions of \( n \) consecutive statements, with transfer functions \( f_i(x) = \text{gen}_i \cup (x - \text{kill}_i) \) for \( i = 1, 2 \ldots, n \). The transfer function for program segment \( PrS \) would be:

\[
f_{PrS}(x) = \text{gen}_{PrS} \cup (x - \text{kill}_{PrS})
\]

where

\[
\begin{align*}
  \text{kill}_{PrS} &= \text{kill}_1 \cup \text{kill}_2 \cup \cdots \cup \text{kill}_n \\
  \text{gen}_{PrS} &= \text{gen}_n \cup (\text{gen}_{n-1} - \text{kill}_n) \cup (\text{gen}_{n-2} - \text{kill}_{n-1} - \text{kill}_n) \cup \\
  &\quad \cdots \cup (\text{gen}_1 - \text{kill}_2 - \text{kill}_3 - \cdots - \text{kill}_n)
\end{align*}
\]

From \( f_{PrS}(x) \) above we can conclude that the \text{gen}-set takes precedence, if two instances of the same data flow value are included in both the \text{gen}- and \text{kill}-set. Looking at \( \text{kill}_{PrS} \), we see that it does not matter where in the program segment a data flow value is killed. Finally, the definition of \( \text{gen}_{PrS} \) tells us that a data flow value is only generated if it is not killed by subsequent statements in the same program segment.
5.1. Data Flow Analysis

5.1.5 Simple Data Flow Problem

This section will provide the algorithm for a data flow problem called reaching definitions. The intention is to provide the reader with a practical example of how a simpler data flow problem is solved, before modeling the direct mapped cache problem. A definition in this context is a variable assignment. Reaching definitions can be used to help the compiler know which definitions are active at a certain program point, and sometimes also what the possible values might be. For example, if only one definition of a variable \( x \) is active at a specific program point, then the compiler might want to replace this definition with a constant. Another usage is if no definition of a variable \( x \) is active at a point of usage, then the compiler is able to warn the programmer that the variable is undefined.

A definition \( d \) reaches a program point \( p \) if this definition is not killed along an execution path up to point \( p \). A definition \( d \) is killed if there is any other definition of variable \( x \) along the path. The definition \( d \) that reaches a program point \( p \) might be the place where the value of \( x \) was last defined. Figure 5.1 illustrates how the sets \( \text{gen}[PrS] \) and \( \text{kill}[PrS] \) are computed. These sets are then passed as input to Algorithm 3.

The algorithm starts by assigning the empty set to \( \text{out}[entry] \), since no definitions reach the start of the CFG. The loop at lines 2-4 initializes the \( \text{out}[PrS] \)-set for each program segment to the empty set. The lines 1-
**Input:** CFG for which $gen[PrS]$- and $kill[PrS]$-sets have been computed for each segment $PrS$.

**Output:** $in[PrS]$ and $out[PrS]$, the set of definitions reaching the entry and exit of each segment $PrS$ in CFG.

1. $out[entry] = \emptyset$
2. foreach program segment $PrS$ other than entry do
   3. $out[PrS] = \emptyset$
4. end
5. while changes to any $out$ occur do
   6. foreach program segment $PrS$ other than entry do
      7. $in[PrS] = \bigcup_{P \in \text{pred}(PrS)} out[P]$
      8. $out[PrS] = gen[PrS] \cup (in[PrS] \setminus kill[PrS])$
   9. end
10. end

**Algorithm 3:** Iterative algorithm to compute reaching definitions.

4 could be combined into a single loop, but it is normal to separate this initialization step to keep the algorithm as general as possible. The general form of this algorithm, described in Section 5.4, can be applied to all data flow problems. At line 5, we iterate until the data flow equations converge, and lines 6-9 apply these equations to every program segment in the CFG. The while-loop will finally terminate, since there exist only a finite number of definitions in a task and these sets cannot grow infinitely. When these sets stop growing, no changes to $out[PrS]$ have occurred and the algorithm terminates. A more formal reasoning about convergence will be made later for our cache analysis.
5.2 Direct Mapped Cache

Cache analysis for direct mapped caches can now be formulated using theory from data flow analysis. We will rely on the gen-kill form. Once we have realized how these gen- and kill-sets can be computed, the algorithm itself is straightforward and has much in common with the reaching definitions algorithm.

5.2.1 Predictable Memory Access

A direct mapped cache is divided into cache lines. Every memory access will either result in a cache miss or hit. If it is a miss then a previously stored memory block in a cache line is replaced with a new block containing the data being accessed. Memory accesses originating from fetching instructions are always predictable, but accesses that come from the instruction operands might be unpredictable. To simplify the problem, we assume that every memory access is predictable and refers to an exact memory address (unpredictable accesses will be handled in Section 5.2.2).

We can now model the problem as follows:

- A cache contains cache lines, each line holding a singleton set of a specific memory block. This singleton set represents the data flow value for the line.

- When a cache miss occurs, a specific block is generated and another one is killed. The generated block is the referenced block, and the killed one is the block previously in the line. The killed one could potentially be any memory block that maps to the same cache line. Each program segment $PrS$ and cache line is associated with these calculated sets, $\text{gen}^{\text{line}}[PrS]$ and $\text{kill}^{\text{line}}[PrS]$.

- The cache analysis will, for each cache line, compute the $\text{in}^{\text{line}}[PrS]$- and $\text{out}^{\text{line}}[PrS]$-sets that are reaching and leaving each program segment $PrS$.

- Which cache line a given memory reference will map to is computed with the help of parameters describing the size of the main memory and the cache memory organization.
Figure 5.2: Simulation of program segments and the propagation of $\text{out}_{\text{line}}[PrS]$-sets between segments.

The $\text{gen}_{\text{line}}[PrS]$-sets are easy to compute. The execution of each program segment is simulated with the cache activity being recorded. Every memory reference that is a write is treated as a cache miss. Every memory reference that is a read might be a cache miss. If the read should be recorded as a miss depends on the data flow value at the specific program point and the cache line. For example, if the second instruction in $PrS_2$ reads from memory block $A$ mapped to line 3, the access is treated as a miss if $A \notin f_1(\text{in}_{\text{line}3}[PrS_2])$.

The gen- and kill-sets for each program segment and cache line can be computed by program segment simulation. As in the reaching definition problem, these gen- and kill-sets are then passed as input to the cache analysis that solves the following data flow equations:

$$\text{in}_{\text{line}}[PrS] = \bigcap_{P \in \text{pred}(PrS)} \text{out}_{\text{line}}[P];$$

$$\text{out}_{\text{line}}[PrS] = \text{gen}_{\text{line}}[PrS] \cup (\text{in}_{\text{line}}[PrS] - \text{kill}_{\text{line}}[PrS]);$$

The first equation computes the intersection of the $\text{out}_{\text{line}}[PrS]$-sets that come from all predecessors to the current program segment. This is appropriate, since we can only be sure that a memory block exists in
50

5.2. Direct Mapped Cache

**Input:** Program Segment PrS

**Output:** $gen_{line}$- and $kill_{line}$-sets for PrS

```
1 foreach cache line do
2    $gen_{line}[PrS] = \emptyset$;
3    $kill_{line}[PrS] = \emptyset$;
4 end
5 foreach instruction or data access in segment PrS do
6    block = memaddress div blocksize;
7    line = block mod linecount;
8    if is a write then
9        line is modified;
10       else if block $\notin gen_{line}[PrS]$ or block $\in kill_{line}[PrS]$ then
11          line is modified;
12       end
13       if line is modified then
14          $gen_{line}[PrS] = \{block\}$;
15          $kill_{line}[PrS] = \cup_{line} - \{block\}$;
16     end
17 end
```

**Algorithm 4:** Calculation of the $gen_{line}$- and $kill_{line}$-sets for some program segment.

this cache line, if the $out_{line}[PrS]$-sets from all predecessors contain this particular memory block. The second data flow equation is dependent on the computed $gen_{line}[PrS]$- and $kill_{line}[PrS]$-sets. Figure 5.2 illustrates the internals of this approach.

Algorithm 4 calculates the $gen_{line}[PrS]$- and $kill_{line}[PrS]$-sets by simulating the cache activity in each program segment. At line 14, we generate the singleton set containing the memory block that is being accessed. Line 15 kills any potential memory block that the cache line could contain before replacement. $\cup_{line}$ denotes the universal set containing every memory block mapping to the cache line, which is involved during the memory access. If a task does not occupy a very big chunk of the main memory, only a few potential memory blocks in the task map to this cache line.
**Input:** CFG for which \( \text{gen}_{\text{line}}[\text{PrS}] \)- and \( \text{kill}_{\text{line}}[\text{PrS}] \)-sets have been computed for each program segment \( \text{PrS} \).

**Output:** \( \text{in}_{\text{line}}[\text{PrS}] \) and \( \text{out}_{\text{line}}[\text{PrS}] \), state of each cache line reaching the entry and exit of each segment \( \text{PrS} \) in CFG.

\[
\text{foreach cache line do} \\
\quad \text{out}_{\text{line}}[\text{entry}] = \emptyset; \\
\quad \text{foreach program segment \( \text{PrS} \) other than entry do} \\
\quad \quad \text{out}_{\text{line}}[\text{PrS}] = \bigcup_{\text{line}}; \\
\quad \text{end} \\
\text{end} \\
\text{foreach cache line do} \\
\quad \text{while changes to any out}_{\text{line}} occur do \\
\quad \quad \text{foreach program segment \( \text{PrS} \) other than entry do} \\
\quad \quad \quad \text{in}_{\text{line}}[\text{PrS}] = \bigcap_{P \in \text{pred}(\text{PrS})} \text{out}_{\text{line}}[P]; \\
\quad \quad \quad \text{out}_{\text{line}}[\text{PrS}] = \text{gen}_{\text{line}}[\text{PrS}] \cup (\text{in}_{\text{line}}[\text{PrS}] - \text{kill}_{\text{line}}[\text{PrS}]); \\
\quad \quad \text{end} \\
\quad \end{end} \\
\text{Algorithm 5:} \quad \text{Iterative algorithm to predict cache miss activity in direct mapped caches.}
\]

Smarter algorithms for kill-set calculations must have knowledge about other program segments and their cache line mappings. Please note that the precision of the cache analysis is not reduced by using these large kill-sets. Furthermore, if an optimized data type is being used for universal sets with the possibility to exclude one memory block, as at line 15 in Algorithm 4, the performance of the cache analysis can be good even for very large memories.

Algorithm 5 predicts the cache contents at the beginning and end of each program segment. This algorithm is very close to the algorithm solving the reaching definition problem. The data flow equations inside the inner loop have been replaced with the equations that are abstracting the contents of cache lines. Also note that the calculation of cache contents must be done for each cache line.
5.2. Direct Mapped Cache

By knowing the $in_{line}[PrS]$ for each cache line and program segment, it is easy to predict the cache contents at any program point. For example, assume that we want to predict the contents of cache $line_4$ in program segment $PrS_2$ after the second instruction inside that segment (denoted $PrS_2[2]$). First, we need to initialize the cache contents according to $in_{line_4}[PrS_2]$. Now we can start a program segment simulation that in formal notation can be written as:

$$\text{Content of line}_4 \text{ at } PrS_2[2] = f_2(f_1(in_{line_4}[PrS_2]))$$

For a multi-processor system the exact cache miss and hit sequence for every program segment is important. The prediction of cache misses is extracted by doing a final simulation of each program segment. Before the simulation starts, the cache is initialized to the calculated $in_{line}[PrS]$-sets. During simulation, the cache hits and misses are recorded in the order they occur. This ordered list is then passed as input to the WCET analyzer. To summarize, these steps are needed to successfully predict the cache miss and hit sequence for a direct mapped cache:

1. Calculate the $gen_{line}[PrS]$- and $kill_{line}[PrS]$-sets for each program segment and cache line, according to Algorithm 4.
2. Perform data flow analysis on the control flow graph, resulting in conservative cache states $in_{line}[PrS]$ and $out_{line}[PrS]$ for each program segment and cache line, according to Algorithm 5.
3. Initialize the cache according to $in_{line}[PrS]$ and perform a final simulation of each program segment, and record the exact cache hit and miss order.

5.2.2 Unpredictable Memory Access

Until now, we have assumed predictable memory accesses, meaning that we know the exact memory address for every memory access. Often only a base address and a range of possible addresses for a given memory access can be deduced. When this occurs, all possible cache lines that a memory access can map to, must be considered. This is illustrated in Figure 5.3. An unpredictable memory access can be modeled as:
1. The possible memory blocks associated to the unpredictable access cannot be included in any $gen_{line}[PrS]$.

2. One of the lines that the possible memory blocks map to will be replaced. Since we do not know which line, we must conservatively kill the previous memory blocks in all $kill_{line}[PrS]$-sets, associated with these lines.

These conservative assumptions can result in very pessimistic WCETs. Therefore, it is important to do a careful pre-analysis to get an exact address, or a small range of possible memory addresses, before the cache analysis starts. How this pre-analysis can be performed is outside the scope of this thesis, but the interested reader should refer to Ye et al [26].

Figure 5.3: The unpredictability of certain memory accesses.
5.3 Set Associative Cache

Set associative caches are harder to model than direct mapped caches. Cache misses in set associative caches are inherently more unpredictable, since the number of cache lines are more than one, and it is not always easy to predict which line will be replaced. At run-time, the replacement algorithm of cache memories decides which exact cache line to replace.

5.3.1 Cache Set Model

We must be able to represent the contents in a cache set. This contents will constitute the data flow value, for each cache set, at any program point. Assume an \( n \)-way set associative cache with replacement policy LRU. The contents with positions in the cache set can be modeled according to:

\[
\{A_1, B_2, ..., D_n\} = \{D_n, ..., B_2, A_1\} = [A, B, ..., D] \neq [D, ..., B, A]
\]

Please note that each memory block in the set is represented by a capital letter, and the subscript of the capital letter denotes the worst case LRU position within the set. Blocks with one as subscript denote the most recently used position in the set, and \( n \) as subscript denote the least recently used position. It is convenient to be spared annotating each letter with a subscript, and we will implicitly assume the LRU position when the memory blocks are given within square brackets. Therefore, the first three sets above are equivalent. When we need to model two or more memory blocks to have the same position, we cannot use the square bracket notation. In that case we need to stick with the usual set notation.

\[
\{A_1, B_2, -3, C_4, D_4\} = \{A_1, B_2, C_4, D_4\}
\]

The hyphen symbol (\(-\)) in a cache set means that the content on this position is not known (in the model). Although this symbol is not strictly required, the use of it can visualize the cache set contents better. For example, the following cache sets are treated as equivalent.

\[
\{-1,-2,-3,-4\} = \{-1,-4\} = \emptyset
\]
**Cache Analysis**

Input: CFG of a task.
Output: $in_{cache}[PrS]$ and $out_{cache}[PrS]$, state of complete cache reaching the entry and exit of each segment $PrS$ in CFG.

1. $out_{cache}[entry] = \emptyset$;
2. foreach program segment $PrS$ other than entry do
3.    $out_{cache}[PrS] = \cup_{cache}$;
4. end
5. while changes to any $out_{cache}$ occur do
6.    foreach program segment $PrS$ other than entry do
7.       $in_{cache}[PrS] = \bigwedge_{P \in \text{pred}(PrS)} out_{cache}[P]$;
8.       $out_{cache}[PrS] = cache\text{-simulation}(\text{memaccess}[PrS], in_{cache}[PrS])$;
9.    end
10. end

Algorithm 6: Refined iterative algorithm to predict cache activity for set associative caches.

5.3.2 Refined Approach

The analysis approach for direct mapped caches is problematic to reuse. As we will see in Section 5.5.2, cache analysis for set associative caches should not use invariant gen- and kill-sets. Therefore, our cache analysis for set associative caches is performed in one single step. Both the program segment simulation and data flow analysis are merged into this step, defined in Algorithm 6.

At line 1-4, all outgoing cache states of the program segments are initialized. Please note that the entry node is initialized to the empty set, since we in the general case cannot know anything about the cache state when the program starts. Also note that all other outgoing cache states are initialized to the universal set, which means that this theoretical cache will generate a hit for every memory access. It might not be obvious that using the universal set really is conservative. But since a special meet operator (defined later) eliminates the cache state that is not common in every path up to the conjuncture point, the solution must always be conservative. In Section 5.3.3, we describe two positive effects of using the universal set.
As can be noted at line 8, this approach requires a cache simulation being performed for each program segment iteration. The input to this simulation is a list of memory accesses, \( \text{memaccess}[PrS] \), together with the cache state, \( \text{in}_{cache}[PrS] \), to start simulation from. The result of this simulation is a new cache state, \( \text{out}_{cache}[PrS] \). Please note that \( \text{in}_{cache}[PrS] \) and \( \text{out}_{cache}[PrS] \) contain the complete cache (a list of all cache sets) for a given program segment, not just a single set. One important difference is that no explicit mathematical formula (like with gen-kill) is being used here. It can be shown that the gen-kill form always converges. This refined algorithm will actually also converge, and the proof is given in Section 5.4.

The cache analysis works for both predictable and unpredictable memory accesses with a base address and range. As we will see in Algorithm 8, the simulation must perform conservative abstractions during unpredictable memory accesses. Therefore, it is important to pass as narrow data ranges as possible to the cache analysis.

Line 7 in Algorithm 6 requires that the meet between two cache states is defined. If there are more than two predecessors for a given program segment, the meet algorithm can be used as many times as necessary. The meet between two cache states is defined in Algorithm 7. For example, the algorithm will produce the following meet between two cache sets:

\[
\begin{align*}
U_{set} \land U_{set} &= U_{set} \\
\{A_1, -2, C_3, -4\} \land U_{set} &= \{A_1, -2, C_3, -4\} \\
\{-1, A_2, B_2, -3, C_4\} \land \{A_1, -2, C_3, -4\} &= \{-1, A_2, -3, C_4\} \\
\{A_1, B_2, C_3, D_4\} \land \{D_1, C_2, B_3, A_4\} &= \{-1, -2, B_3, C_3, A_4, D_4\}
\end{align*}
\]

The meet is performed in an iteration for every pair of cache sets \((set_1, set_2)\). This pair must be extracted in exact order from both \( cache_1 \) and \( cache_2 \), so that they correspond to each other. At line 2-7, we handle the universal set. If both sets are universal, the resulting set must be universal. If one of them is universal, the resulting set is the other one. At line 9-12, we handle the meet of two non-universal sets. The resulting set \( set_{out} \) can be represented by a queue data type. Every node in the queue has an implicit LRU position. Thus, the first node represents the most recently used position, and the last node represents the least recently
**Input**: cache₁ and cache₂, representing the cache states to meet.

**Output**: cacheₜₐₜ, representing the new cache state after meet.

1. foreach (set₁, set₂) in (cache₁, cache₂) do
   2. if set₁ = ∪ set and set₂ = ∪ set then
      3. setₜₐₜ = ∪ set;
   4. else if set₁ = ∪ set and set₂ ≠ ∪ set then
      5. setₜₐₜ = set₂;
   6. else if set₁ ≠ ∪ set and set₂ = ∪ set then
      7. setₜₐₜ = set₁;
   8. else
      9. Represent setₜₐₜ by a LRU queueₜₐₜ with size max(|set₁|, |set₂|);
      10. foreach line in queueₜₐₜ with position pos do
          11. Let line contain all those blocks that exist in both set₁ and set₂, and have pos as maximum position;
      12. end
   13. end
14. end

**Algorithm 7**: Conservative meeting of LRU cache contents.

used position. At line 9, we initialize queueₜₐₜ to have a node count that is the maximum of the node count of set₁ and set₂. At line 10-12, we iterate through each node (LRU position) in queueₜₐₜ, and put those memory blocks that exist in both set₁ and set₂ and have the current LRU position as the maximum in set₁ and set₂, into the current node.

The cache simulation of a program segment is defined in Algorithm 8. Each memory access inside the input program segment is iterated in order. If the memory access is unpredictable, each possible memory block for this access needs to be iterated as well. At line 3, we calculate which cache set this possible block is mapped to. Remember that a cache set is represented by a queue data type, in which each node represents a cache line with an implicit LRU position. Line 4 extracts this queue type. If the set is universal, the memory access can directly be treated as a cache hit. At line 8, we initialize nodeₙₑₜₐₑ to the singleton set containing the current block if the access is predictable. Otherwise nodeₙₑₜₐₑ is initialized to the empty
5.3. Set Associative Cache

**Input:** \text{memaccess}[PrS] and \text{in}_{cache}[PrS], representing the memory accesses and complete start cache for segment PrS.

**Output:** \text{out}_{cache}[PrS], complete outgoing cache after local simulation of segment PrS.

```
foreach access in \text{memaccess}[PrS] do
    foreach possible memory block within access do
        set = get-set-no(block);
        queue = get-queue(set);
        if set = \bigcup_{set} /*cache hit*/ then
            continue;
        if access is predictable then
            Let node\new = \{block\};
        else
            Let node\new = \emptyset;
        end
        if access is predictable and a read then
            Lookup node\found in queue that contains block;
            if node is found /*cache hit*/ then
                Let node\found = node\found - \{block\};
                Merge node\found with predecessor(node\found);
            end
        end
        Put node\new in front of queue;
        if sizeof(set) > associativity then
            Truncate queue to maximum possible length;
        end
    end
end
```

**Algorithm 8:** Cache simulation of a program segment with LRU.

set, since we still need to reserve a position for this unpredictable access. At line 12-18, we handle the scenario where the access is a predictable memory read. If the block is found within a node (node\found) in the queue, the memory access is treated as a hit. We remove this block from node\found
Cache Analysis

<table>
<thead>
<tr>
<th>Block</th>
<th>Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>hit $\Rightarrow {A_1, B_2, -3, D_4}$</td>
</tr>
<tr>
<td>B</td>
<td>hit $\Rightarrow {B_1, A_2, -3, D_4}$</td>
</tr>
<tr>
<td>E</td>
<td>miss $\Rightarrow {E_1, B_2, A_3, -4}$</td>
</tr>
<tr>
<td>A</td>
<td>hit $\Rightarrow {A_1, E_2, B_3, -4}$</td>
</tr>
</tbody>
</table>

Table 5.1: Example of cache simulation with LRU policy.

and merge this node with its predecessor (we will soon motivate why the merge is done). There exists a special case for this merge if the hit was on the first LRU position. In this case, the node is simply removed. It is correct to do this, since a worst case scenario cannot include two blocks in the first LRU position. At line 19, we put \textit{node}\textsubscript{new} into the front of the queue, which means that it gets the first LRU position (most recently used). All other blocks are implicitly pushed down one step up to the hit position (merge position). At the end of the inner loop, it can happen that the cache set is larger than the allowed associativity. Lines 20-22 handle this by truncating the queue to maximum possible length, if necessary.

The merge that is performed at line 16 is not strictly necessary, but it reduces the pessimism in a conservative way. Assume that memory blocks \textit{A\textsubscript{found}} and \textit{B} are inside \textit{node}\textsubscript{found}, which has an implicit LRU position of 3. In reality, at most one of these memory blocks can have this LRU position. If \textit{A\textsubscript{found}} actually has position 3, it implies that \textit{B} at most has position 2. After the merge is done and \textit{node}\textsubscript{new} is put in front of the queue, \textit{B} will get position 3, which is conservative. Now, assume instead that \textit{B} in reality has position 3, which implies that \textit{A\textsubscript{found}} is at most at position 2. Again, after the merge is done and \textit{node}\textsubscript{new} is put in front of the queue, \textit{B} will get position 3, which also is conservative.

Table 5.1 shows an example of cache simulation. Assume a 4-way set associative cache with LRU policy. Inside a program segment \textit{PrS}_3, the memory blocks \{\textit{A}, \textit{B}, \textit{E}, \textit{A}\} for cache set\textsubscript{1} will be accessed in order. Furthermore, assume that the incoming cache set state is \textit{in\textsubscript{set1}}[\textit{PrS}_3] = \{-1, A_2, B_2, -3, D_4\}. If the program segment simulation is performed according to Algorithm 8, the result will be \textit{out\textsubscript{set1}}[\textit{PrS}_3] = \{A_1, E_2, B_3, -4\}.
The prediction of cache misses is extracted by doing a final simulation of each program segment. Before the simulation starts, the cache is initialized to the calculated $in_{set}[PrS]$-sets. During simulation, the cache hits and misses are recorded in the order they occur. This ordered list is then passed as input to the WCET analyzer. To summarize, these steps are needed to successfully predict the cache miss and hit sequence for a set associative cache:

1. Perform data flow analysis on the control flow graph, resulting in conservative cache states $in_{set}[PrS]$ and $out_{set}[PrS]$ for each program segment and cache set, according to Algorithm 6.

2. Initialize the cache according to $in_{set}[PrS]$ and perform a final simulation of each program segment, and record the exact cache hit and miss order.

Compilers often optimize loops through a technique known as loop unrolling. This can reduce loop overhead and the number of branches taken at runtime. For similar purposes, the input CFG to our cache analysis should have all loops unrolled once. The first iteration of each loop is extracted and placed before the start of the loop structure. This technique takes care of situations where the first loop iteration loads the cache set contents, which later is reused in subsequent iterations. Thus, the results from the cache analysis can be less pessimistic.

5.3.3 Universal Set

In previous cache analysis approaches, each outgoing cache state has been initialized to the empty set. In our approach, we use the universal set. Figure 5.4(a) illustrates a CFG with $out[PrS_2]$ initialized to the empty set (for a given cache set). The simulation of $PrS_1$ starts with an empty cache, since the cache set of the predecessors to $PrS_1$ has no common data. Therefore, $out[PrS_1]$ becomes $[A,-,-,-]$. Figure 5.4(b) shows a CFG with $out[PrS_2]$ initialized to the universal set. This time, the simulation of $PrS_1$ starts with $[A,B,C,D]$, since this is the common predecessor data. Now, $out[PrS_1]$ becomes $[A,B,C,D]$, which is a much better result to proceed from in the next iteration.
Cache Analysis

There is no need for an exact order in which the program segments are traversed. This can also be contributed to the fact that we use the universal set in the initialization. If each outgoing cache state is initialized to the empty set, a certain traversal order would be necessary. Figure 5.5(a) illustrates such a CFG being traversed in the order of $PrS_1$, $PrS_2$, $PrS_3$. When $PrS_2$ has been reached, the simulation starts with the empty set, since the predecessors to $PrS_2$ has no common data. Therefore, $out[PrS_2]$ becomes $[A, -, -, -]$. The best traversal order in this case would be $PrS_1$, $PrS_3$, $PrS_2$. Figure 5.5(b) shows the CFG with $out[PrS_3]$ initialized to the universal set. This time, the traversal order is not important and the precision is kept for any order. Now, $out[PrS_2]$ becomes $[A, B, C, D]$, which is a much better result to proceed from in the next iteration.
5.3. Set Associative Cache

5.3.4 Analysis Example

This section will work through an example to see how the cache analysis works in practice. Assume a 2-way set associative cache (LRU) with 16 bytes as block size and 128 bytes as total size. This cache memory is connected to a 1024 bytes main memory. Please refer to Figure 5.6 for the input to the cache analysis. The input is a CFG with a list of memory accesses in each node. Assume that all these accesses are reads. Each access can include a possible range, as seen in column A:. The rest of the columns are calculated by the analysis and given as a convenience. Column B: represents the range of memory blocks, column S: represents the range of cache sets, and column T: represents the range of tags.

In this example, we will only show cache analysis for cache set 1. Figure 5.7(a) illustrates how the CFG is initialized. All out\textsubscript{set1}[PrS] are initialized to U\textsubscript{set}, except out\textsubscript{set1}[PrS\textsubscript{0}] which is the entry node. As previously noted, the iteration order of the nodes is undefined in the algorithm. However, the walk through order in this particular example is in ascending order of node id (2-6). The start and exit nodes are not traversed. The iterations can be summarized into:

1. The result of this iteration is found in Figure 5.7(b). During simulation, every memory access in PrS\textsubscript{2-5} results in a hit, since the meet
Figure 5.7: Cache analysis with five iterations. Each iteration shows a CFG with miss (hit) list in each program segment.

of the predecessors out-value equals to $U_{set}$. The simulation in $PrS_6$ starts with a miss. We cannot put the block in the first LRU position, since the memory access is unpredictable (spans two blocks and cache
sets). However, we still have to reserve the position for it in both cache set₁, as shown in Figure 5.7(b), and set₂. The next two memory accesses are predictable, and therefore \( \text{out}_{set₁}[PrS₆] = \{1₁, 5₂\} \).

2. The result of this iteration is found in Figure 5.7(c). Every memory access in \( PrS₂₋₄ \) results in a hit, since the meet of the predecessors out-value equals to \( \bigcup \text{set} \). \( PrS₅ \) has no instruction inside it (is a control node) and, therefore, sets the \( \text{out}_{set₁}[PrS₅] = \text{in}_{set₁}[PrS₅] = \{1₁, 5₂\} \).

3. The result of this iteration is found in Figure 5.7(d). \( PrS₂ \) starts to simulate from \( \text{in}_{set₁}[PrS₂] = \text{out}_{set₁}[PrS₅] = \{1₁, 5₂\} \). Due to this, every memory access in \( PrS₂ \) results in a hit. \( PrS₃ \) and \( PrS₄ \) also start to simulate from \( \{1₁, 5₂\} \), with results shown in the graph. \( PrS₅ \) takes the meet of the three predecessors out-value, which results in \( \text{in}_{set₁}[PrS₅] = \{1₁, 5₂\} \∧ \{1₁, 9₂\} \∧ \{1₁, 5₂\} = \{1₁, -2\} \).

4. The result of this iteration is found in Figure 5.7(e). It is only \( \text{in}_{set₁}[PrS₂] \) that gets a new value, since \( \text{out}_{set₁}[PrS₅] \) changed in the previous iteration. This value change results in one more miss inside \( PrS₂ \) than in the previous iteration. After this iteration, the algorithm detects that nothing has changed, and we therefore have a solution in this last graph.

5.3.5 Problems with FIFO and LFU

The most common replacement policies for set associative caches are LRU, FIFO and LFU. Our approach only work with the LRU replacement policy. Assume a 4-way set associative cache with FIFO policy. Inside a program segment, the memory blocks \([A,B,C,D,A,B,C,H,C]\) for a specific cache set will be accessed in order. If we are extremely pessimistic and assume an empty initial cache set state, we will from simulation results get the cache set transitions in Table 5.2, column 2. This results in a total of 5 cache misses. Furthermore, assume that the actual incoming cache set is \([G,H,B,A]\). This will after execution of the program segment result in a total of 6 cache misses. In other words, even if we make extremely
A miss ⇒ [A - - -]  hit ⇒ [G H B A]

B miss ⇒ [B A - -]  hit ⇒ [G H B A]

C miss ⇒ [C B A -]  miss ⇒ [C G H B]

D miss ⇒ [D C B A]  miss ⇒ [D C G H]

A hit ⇒ [D C B A]  miss ⇒ [A D C G]

B hit ⇒ [D C B A]  miss ⇒ [B A D C]

C hit ⇒ [D C B A]  hit ⇒ [B A D C]

H miss ⇒ [H D C B]  miss ⇒ [H B A D]

C hit ⇒ [H D C B]  miss ⇒ [C H B A]

<table>
<thead>
<tr>
<th>Block</th>
<th>Simulation</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>miss ⇒ [A - - -]</td>
<td>hit ⇒ [G H B A]</td>
</tr>
<tr>
<td>B</td>
<td>miss ⇒ [B A - -]</td>
<td>hit ⇒ [G H B A]</td>
</tr>
<tr>
<td>C</td>
<td>miss ⇒ [C B A -]</td>
<td>miss ⇒ [C G H B]</td>
</tr>
<tr>
<td>D</td>
<td>miss ⇒ [D C B A]</td>
<td>miss ⇒ [D C G H]</td>
</tr>
<tr>
<td>A</td>
<td>hit ⇒ [D C B A]</td>
<td>miss ⇒ [A D C G]</td>
</tr>
<tr>
<td>B</td>
<td>hit ⇒ [D C B A]</td>
<td>miss ⇒ [B A D C]</td>
</tr>
<tr>
<td>C</td>
<td>hit ⇒ [D C B A]</td>
<td>hit ⇒ [B A D C]</td>
</tr>
<tr>
<td>H</td>
<td>miss ⇒ [H D C B]</td>
<td>miss ⇒ [H B A D]</td>
</tr>
<tr>
<td>C</td>
<td>hit ⇒ [H D C B]</td>
<td>miss ⇒ [C H B A]</td>
</tr>
</tbody>
</table>

Table 5.2: Example (1) of replacement with FIFO policy.

pessimistic assumptions, there exist actual cache states at run-time that behaves even worse in terms of cache misses.

Once again, assume a 4-way set associative cache with FIFO policy. Inside a program segment, the memory blocks [A,B,C,D,A,B,C,H,C] for a specific cache set will be accessed in order. The difference this time is that data flow analysis (correctly) states that the worst possible incoming cache set state is \{-1, -2, -3, A_4, B_4\}, as noted in Table 5.3. In other words, we only know that A and B are in the set, and possibly also at the worst LRU position. Therefore, the first two memory block accesses must result in a hit. The next block access results in a miss. In an attempt to be very conservative we flush the cache set n = 4 times, and after this we begin to build up the cache set again, as can be viewed in the simulation column. Despite this very conservative approach, we end up in a simulated cache hit, when in reality would be a miss. Since, cache order is important on MPSoC, this approach is not safe.

Set associative caches with LFU replacement policy are even harder to predict, since each cache line in a set is associated with a frequency counter representing how many times a memory block has been referenced in the past. The memory blocks that have been referenced the most will be kept in the cache set. Since these cache states are even more dependent
5.3. Set Associative Cache

<table>
<thead>
<tr>
<th>Block</th>
<th>Simulation</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>hit $\Rightarrow {-1, -2, -3, A_4, B_4}$</td>
<td>hit $\Rightarrow [G \ H \ B \ A]$</td>
</tr>
<tr>
<td>B</td>
<td>hit $\Rightarrow {-1, -2, -3, A_4, B_4}$</td>
<td>hit $\Rightarrow [G \ H \ B \ A]$</td>
</tr>
<tr>
<td>C</td>
<td>miss $\Rightarrow [- - - -]$</td>
<td>miss $\Rightarrow [C \ G \ H \ B]$</td>
</tr>
<tr>
<td>D</td>
<td>miss $\Rightarrow [- - - -]$</td>
<td>miss $\Rightarrow [D \ C \ G \ H]$</td>
</tr>
<tr>
<td>A</td>
<td>miss $\Rightarrow [- - - -]$</td>
<td>miss $\Rightarrow [A \ D \ C \ G]$</td>
</tr>
<tr>
<td>B</td>
<td>miss $\Rightarrow [B - - -]$</td>
<td>miss $\Rightarrow [B \ A \ D \ C]$</td>
</tr>
<tr>
<td>C</td>
<td>miss $\Rightarrow [C \ B - -]$</td>
<td>hit $\Rightarrow [B \ A \ D \ C]$</td>
</tr>
<tr>
<td>H</td>
<td>miss $\Rightarrow [H \ C \ B -]$</td>
<td>miss $\Rightarrow [H \ B \ A \ D]$</td>
</tr>
<tr>
<td>C</td>
<td><strong>hit</strong> $\Rightarrow [H \ C \ B -]$</td>
<td><strong>miss</strong> $\Rightarrow [C \ H \ B \ A]$</td>
</tr>
</tbody>
</table>

Table 5.3: Example (2) of replacement with FIFO policy.

on historical execution, and not on local events inside the current program segment, it is even harder to make safe assumptions. For these reasons, our cache analysis approach does not yet support set associative caches with a FIFO or LFU replacement policy.
5.4 Verification of Cache Analysis

This section will verify that our cache analysis for direct mapped and set associative caches converges. Intuitively, the analysis for set associative caches converges, since the cache meet operator always produces the worst case of the memory blocks that both cache sets have in common. We will also verify the correctness and precision of the produced solutions.

5.4.1 Data Flow Analysis Framework

There exists a common theory basis for a large family of data flow problems, that has resulted in a data flow analysis framework [16]. This framework can provide answers to the following questions:

- The correctness of the solution obtained by an iterative algorithm.
- Whether the solution obtained by an iterative algorithm will converge or not.
- The precision of the solution obtained by an iterative algorithm.

A common theoretical framework does not just provide answers to the above important questions. It can also be used as a reusable module across several implementations. With this framework it is easy to identify what components can be reused, and what need to be treated as specific parameters for each data flow problem. This is a major advantage, since coding errors are reduced and the programmer can focus on the specific problem. A data flow analysis framework \((D, V, \land, F)\) consists of the following specific parameters:

1. A control flow graph, with two specific nodes marked as entry and exit.
2. A direction \(D\) of the data flow in the graph, which could either be forward or backward.
3. A domain of data flow values \(V\).
4. A meet operator \(\land\) defined on the data flow values.
5.4. Verification of Cache Analysis

**Input:** CFG of a task.

**Output:** \(\text{in}[PrS]\) and \(\text{out}[PrS]\), state (appropriate for the problem) reaching the entry and exit of each segment \(PrS\) in CFG.

1. \(\text{out}[\text{entry}] = \upsilon_{\text{entry}}\);
2. **foreach** program segment \(PrS\) other than entry **do**
3. \(\text{out}[PrS] = \top\);
4. **end**
5. **while** changes to any \(\text{out}\) occur **do**
6. **foreach** program segment \(PrS\) other than entry **do**
7. \(\text{in}[PrS] = \bigwedge_{P \in \text{pred}(PrS)} \text{out}[P]\);
8. \(\text{out}[PrS] = f_{PrS}(\text{in}[PrS])\);
9. **end**
10. **end**

Algorithm 9: Iterative algorithm for a forward data flow problem.

5. A family of transfer functions \(F\) from \(V\) to \(V\).

6. A constant value \(\upsilon_{\text{entry}}\) (or \(\upsilon_{\text{exit}}\)) in \(V\), representing the boundary condition for the framework.

The general iterative algorithm for a forward data flow problem is given in Algorithm 9. Both Algorithm 5 (direct mapped cache analysis) and Algorithm 6 (set associative cache analysis) are actually two instances of this generic algorithm. As \(\upsilon_{\text{entry}}\), we select the empty set \(\emptyset\). As the top element \(\top\), we select the universal cache state \(U_{\text{set}}\) (or \(U_{\text{line}}\)). As the meet operator \(\land\), we select the intersection operator for direct mapped caches and a specific meet operator for set associative caches (defined in Algorithm 7). As the transfer function for a program segment, we select the kill-gen form for direct mapped caches and the cache simulation function for set associative caches (defined in Algorithm 8).

5.4.2 Partial Order for Lattice

In Chapter 2, a semi-lattice was defined as a special po-set. Now, we will give an equivalent definition of a semi-lattice that is more appropriate to
use for data flow problems. Aho et al [16] shows why this definition is equivalent to the previous, and is motivated by their proof that the meet operator $\wedge$ (in our case, the cache meet operator) returns the greatest lower bound of any two elements $x, y \in V$. A semi-lattice is a set $V$ and a binary meet operator $\wedge$ such that for all $x, y$ and $z$ in $V$:

1. Meet is idempotent, $x \wedge x = x$.

2. Meet is commutative, $x \wedge y = y \wedge x$.

3. Meet is associative, $x \wedge (y \wedge z) = (x \wedge y) \wedge z$.

A meet semi-lattice has a top element, denoted $\top$, such that for all $x$ in $V$, $\top \wedge x = x$. Such a lattice does not need to have a bottom element $\bot$. We must also define the partial order in the domain of data flow values $V$. For all $x$ and $y$ in $V$:

$$x \leq y \iff x \wedge y = x$$

Remember that the meet operator $\wedge$ returns the greatest lower bound of any two elements $x$ and $y$. Therefore, if $x \wedge y = x$, then $x$ must be less or equal to $y$. Aho et al [16] prove that this definition gives a partial order for a semi-lattice.

Let us now look at two cache lattice diagrams. As in every meet semi-lattice, a top element exists (including an optional bottom element). Figure 5.8 illustrates a lattice for a cache line in a direct mapped cache. All values in this lattice represent the domain of the data flow values. The memory blocks $A, B, C$ and $D$ are the only blocks that can be mapped to this line. In a lattice for a cache line, the top element $\top$ is represented by the universal set $U_{line}$, and the bottom element $\bot$ is represented by $\emptyset$. The meet operator for direct mapped caches is the ordinary intersection operator.

Figure 5.9 illustrates a lattice for a cache set in a 2-way set associative cache. All values in this lattice represent the domain of the data flow values. Assume that the memory blocks $A$ and $B$ are the only blocks that can be mapped to this set. This would represent a main memory that is not larger than the cache memory, but we have to keep the example small.
5.4. Verification of Cache Analysis

Please note that data flow value \( \{A_1, B_1, -2\} \) is not valid, since both \( A \) and \( B \) cannot be in the most recently used position in the worst case. In a lattice for a cache set, the top element \( \top \) is represented by the universal set \( \mathbb{U}_{set} \), and the bottom element \( \bot \) is represented by \( \{-1, -2\} \) or \( \emptyset \). The meet operator for set associative caches is our specific cache set meet, defined in Algorithm 7.

Every element in a lattice represents an abstraction of a possible cache set state. For example, a safe abstraction for a cache set state in a 2-way set associative cache might be \( \{-1, A_2, B_2\} \). This state implies that we cannot say anything about the first position, and the memory blocks \( A \) and \( B \) are, in the worst case, at the second LRU position.
5.4.3 Further Requirements

The family $F$ of transfer functions $V \rightarrow V$ in a data flow framework needs to have the following properties to work with an iterative algorithm:

1. $F$ has an identity function $I$, such that $I(x) = x$ for all $x \in V$.

2. $F$ is closed under composition, that is, for any two functions $f, g \in F$, the function $h$ is defined by $h(x) = g(f(x)) \in F$.

A data flow problem will always converge, if the framework is monotone [16]. Intuitively, this seems plausible in our case, since the cache meet operator in every iteration, for all cache sets inside $in_{cache}[PrS]$, can only decrease the data flow values in terms of $\leq$ relationship for lattices. At some point, the data flow value for each cache set is either at the bottom element $\bot$ in the lattice, or the value has converged before reaching $\bot$.

The framework is monotone if, when we apply any transfer function $f \in F$ to $x, y \in V$, $x$ being no greater than $y$, $f(x)$ is no greater than $f(y)$.

$$x \leq y \implies f(x) \leq f(y), \quad \forall x, y \in V, \forall f \in F$$

According to Aho et al [16], the below definition of monotone frameworks is equivalent. The definition informally says that applying the meet operator early might have a bad impact on the precision.

$$f(x \land y) \leq f(x) \land f(y), \quad \forall x, y \in V, \forall f \in F$$

A stronger framework is the distributive [16] framework. The framework guarantees that it does not matter if the meet operator is applied early or late, the result will always be the same. This is illustrated in Figure 5.10 for cache analysis with program segments. A distributive framework, in general, results in more precise solutions than with a monotone framework.

$$f(x \land y) = f(x) \land f(y), \quad \forall x, y \in V, \forall f \in F$$

Finally, the semi-lattice in the framework needs to be of finite height. Obviously, a semi-lattice consisting of a finite set of data flow values will be of finite height. The height of a semi-lattice is defined as the largest
number of $<$ relations in any ascending chain. Thus, the height is one less than the number of elements in that chain.

### 5.4.4 Precision of Solution

In this section, we will reason about the precision of the solutions. The precision increases if the framework is distributive. First, we need to formulate the following:

- The Fixed Point (FP) solution is any solution that satisfies the data flow equations.

- The Maximum Fixed Point (MFP) solution is the best solution we can get with a (generic) iterative algorithm. A MFP solution has the property that in any other iterative solution, the values of $\text{in}_{\text{set}}[PrS]$ and $\text{out}_{\text{set}}[PrS]$ are $\leq$ the corresponding values of the MFP.

- The Meet Over Paths (MOP) solution is an even better solution, since the meet operator is applied as late as possible. In the analogy of cache analysis, this means that the cache meet operator is applied after all paths from entry to a given $PrS$ have been simulated. In practice, this can only be achieved for trivial CFGs. Flow graphs with loops contain an infinite number of structural paths.

\[
MOP[PrS] = \bigwedge_{P, \text{a structural path from entry to } PrS} f_{P(\nu_{\text{entry}})}
\]
The ideal solution is the best solution, since only feasible paths are considered. However, remember that the enumeration of all feasible paths in a task is an undecidable problem.

$$\text{Ideal}[PrS] = \bigwedge_{P, \text{a feasible path from entry to PrS}} f_P(v_{\text{entry}})$$

The precisions defined above are related to each other according to FP $\leq$ MFP $\leq$ MOP $\leq$ Ideal. All solutions less than the Ideal solution are conservative and safe. Monotone frameworks produce MFP solutions. If the framework also is distributive, the MFP solutions are equal to MOP solutions [16]. Thus, FP $\leq$ MFP = MOP $\leq$ Ideal. Figure 5.11 illustrates a part of a CFG with four program segments. Distributive frameworks guarantee that the value of $out_{PrS4}$ gets the same precision, regardless how early the cache meet operator is applied:

$$out_{PrS4} = f_{PrS4}(f_{PrS2}(out_{PrS1})) \land f_{PrS4}(f_{PrS3}(out_{PrS1}))$$

$$= f_{PrS4}(f_{PrS2}(out_{PrS1}) \land f_{PrS3}(out_{PrS1}))$$

5.4.5 Analysis Properties

If we can show that our cache analysis approaches obey the requirements defined in Section 5.4.2 and 5.4.3, we have shown that the below prop-
5.4. Verification of Cache Analysis

Properties are true. If the cache analysis approach is monotone, but not also distributive, point 4 below does not hold.

1. The cache analysis produces a solution to the data flow equations.

2. The cache analysis algorithm always converges, that is, it will not run in infinite amount of time.

3. The solution is the maximum fixed point (MFP) solution.

4. The solution is additionally the meet over paths (MOP) solution. A MOP solution has the nice property that we get maximum possible precision in our solutions.

5.4.6 Proof for Direct Mapped Cache

This section proves the properties that our analysis approach for direct mapped caches leads to. Conclusions are given at the end.

Basic Conditions

First, we must show that the data flow values in domain $V$ belong to a cache line lattice. This requires that the cache meet operator is idempotent, commutative and associative. We use $\cap$ as the meet operator, and this operator fulfills the requirements.

Our family of transfer functions needs to satisfy the requirements of having an identity function and being closed under composition, as previously defined. The family has an identify function $I$, when the $\text{gen}_{line}[PrS]$- and $\text{kill}_{line}[PrS]$-sets are empty. It was shown in Section 5.1.4 that composing transfer functions in the gen-kill form, the resulting function gets into the same form. Since our functions are of the gen-kill form, they are closed under composition.

We should also note that we always have a memory with a finite set of memory blocks. Therefore, the domain of data flow values is finite and this implies that any cache line lattice is of finite height.
Solution

Assuming that our cache analysis algorithm is monotone (converges), it will return a solution to the data flow equations. If the data flow equations in Algorithm 5 are not satisfied, then the condition in the while-loop notifies this, and another loop will be executed again.

Distributive

The cache analysis is distributive, if our transfer function $f$ behaves, for all cache lines, such that $f(x \cap y) = f(x) \cap f(y)$, $\forall x, y \in V$. Our data flow equations are in the form $\text{in} \_\text{line}[PrS] = \bigcap_{P \in \text{pred}(PrS)} \text{out} \_\text{line}[P]$ and $\text{out} \_\text{line}[PrS] = \text{gen} \_\text{line}[PrS] \cup (\text{in} \_\text{line}[PrS] - \text{kill} \_\text{line}[PrS])$. In a simpler notation, let the latter equation be defined as $f(x) = G \cup (x - K)$. We can now check if $f(x \cap y) = f(x) \cap f(y)$ holds:

$$G \cup ((x \cap y) - K) = (G \cup (x - K)) \cap (G \cup (y - K))$$

We can see that the values in $G$ are on both sides of the equation above. Therefore, we can eliminate $G$ completely. The resulting equation below can be verified to be true by a Venn diagram.

$$(x \cap y) - K = (x - K) \cap (y - K)$$

Monotone

Since our framework is distributive, it must also be monotone (a weaker condition).

Conclusions

We have checked and verified our cache analysis for direct mapped caches. The analysis algorithm will converge and run in a finite amount of time, since the framework is monotone. The framework is also distributive. Therefore, the precision of the solution to the data flow equations is MOP. The MOP solution is always conservative and safe, since $\text{MOP} \leq \text{Ideal}$. 
5.4. Verification of Cache Analysis

5.4.7 Proof for Set Associative Cache

This section proves the properties that our analysis approach for set associative caches leads to. Conclusions are given at the end.

Basic Conditions

First, we must show that the data flow values in domain $V$ belong to a cache set lattice. This requires that, for all $x, y$ and $z$ in $V$:

1. The cache meet operator is idempotent, $x \land x = x$. We can see that Algorithm 7 produces the same value as the input, if two input cache sets ($set_1, set_2$) have identical contents.

2. The cache meet operator is commutative, $x \land y = y \land x$. We can see that Algorithm 7 produces the same result, regardless of the pair order ($set_1, set_2$).

3. The cache meet operator is associative, $x \land (y \land z) = (x \land y) \land z$. Remember that the cache meet operator produces the worst case of the memory blocks that both cache sets have in common. The worst case will be the same, regardless of the order the cache sets are applied, if the same elements are involved.

Our family of (cache simulation) transfer functions $F : V \rightarrow V$ needs to satisfy the requirements of being an identity function and being closed under composition, as previously defined. The function $f$ becomes an identity function $I$, if the simulation is performed on a program segment with no instructions, that is $I(x) = x$ for all $x \in V$. This is illustrated in Figure 5.12(a).

Let us check if the functions $f, g \in F$ are closed under composition, that is $h(x) = g(f(x)) \in F$. We let $f_{PrS_1}$ represent our cache simulation function being called for $PrS_1$, and $f_{PrS_2}$ represent our cache simulation function for $PrS_2$. If we concatenate the instructions from $PrS_1$ and $PrS_2$ into $PrS_3$ and let $f_{PrS_3}$ represent our cache simulation function being called for $PrS_3$, we conclude that $f_{PrS_3}(x) = f_{PrS_2}(f_{PrS_1}(x))$. This is illustrated in Figure 5.12(b).
We should also note that we always have a memory with a finite set of memory blocks. Assume that \( m \) memory blocks can be mapped into a cache set with \( n \) available slots. The number of data flow values containing \( m \) memory blocks are \( m \times n + (m - 1)n + \cdots + (m - n + 1)n = n \frac{m!}{(m-n)!} \).

Sometimes, only \( i \) (\( 0 \leq i < n \)) memory blocks are contained in a data flow value. The total number of data flow values (excluding \( U_{set} \) and \( \emptyset \)) in the domain, \( |V| \), are \( \sum_{i=1}^{n} n \frac{m!}{(m-i)!} \). Therefore, the domain of data flow values is finite and this implies that any cache set lattice is of finite height.

**Solution**

Assuming that our cache analysis algorithm is monotone (converges), it will give a solution to the data flow equations. If the data flow equations in Algorithm 6 are not satisfied, then the condition in the while-loop notifies this, and another loop will be executed again.

**Monotone**

Our cache analysis is monotone, if our (cache simulation) transfer function \( f_{PrS} \) behaves, for all cache sets, such that \( x \leq y \implies f_{PrS}(x) \leq f_{PrS}(y) \), \( \forall x, y \in V \). We will check if our (cache simulation) transfer function \( f_s \) is monotone for a single instruction \( s \), which could either generate a cache miss or hit. If it is monotone, we conclude that the transfer function \( f_{PrS} \) also must be monotone for a program segment \( PrS \), since
5.4. Verification of Cache Analysis

\[ f_{P_f S} = f_{s_n} \circ f_{s_2} \circ \cdots \circ f_{s_1} \]. There are three general cache scenarios that must be taken into consideration when applying \( f_s \) to \( x, y \in V, x \leq y \):

1. Neither \( x \) nor \( y \) contain the memory block \( A \) that \( f_s \) tries to access. Putting \( A \) in the first position of both cache sets cannot influence the partial order. All other memory blocks in both \( x \) and \( y \) will increase one position and this has no impact on the partial order. For example, \( x = \{-1, C_2\}, y = \{B_1, C_2\}, f_s(x) = \{A_1, -2\}, f_s(y) = \{A_1, B_2\} \). Thus, the partial order is maintained.

2. Any of \( x \) or \( y \) (but not both) contains the memory block \( A \) that \( f_s \) tries to access. First, assume that \( x \) contains \( A \), but not \( y \). This implies that \( x \not\leq y \), so we disregard this case. Second, assume that \( y \) contains \( A \) (with hit position \( y_{hp} \)), but not \( x \). Putting \( A \) in the first position of both cache sets cannot influence the partial order. The memory blocks in both \( x \) and \( y \), up to \( y_{hp} \), will increase one position and this has no impact on the partial order. The remaining blocks in \( x \) will also increase one position, while the blocks in \( y \) after \( y_{hp} \) will retain their positions, and this has no impact on the partial order, since increasing the positions in \( x \) can only make it even more \( \leq \). For example, \( x = \{-1, A_2, -3, D_4\}, y = \{-1, A_2, C_3, D_4\}, f_s(x) = \{A_1, -2, -3, C_4\}, f_s(y) = \{A_1, -2, C_3, D_4\} \). Thus, the partial order is maintained.

3. Both of \( x \) and \( y \) contain the memory block \( A \) that \( f_s \) tries to access. We can immediately disregard the case where \( x \) contains block \( A \) in a smaller position than \( y \), since this implies that \( x \not\leq y \). If block \( A \) is in the same position in both cache sets, we move \( A \) to the first position in both cache sets. All blocks from each cache set will maintain their mutual order, and this cannot influence the partial order. For example, \( x = \{-1, A_2, -3, D_4\}, y = \{-1, A_2, C_3, D_4\}, f_s(x) = \{A_1, -2, -3, D_4\}, f_s(y) = \{A_1, -2, C_3, D_4\} \). Finally, assume that \( x \) contains \( A \) in a larger position than \( y \). Denote the hit positions \( x_{hp} \) and \( y_{hp} \), respectively. Putting \( A \) in the first position of both cache sets cannot influence the partial order. All memory block positions maintain the mutual order, except those blocks at \([y_{hp}..x_{hp}]\). However, within this interval, it is only the blocks in \( x \) that
increases the positions compared to \( y \). This can only make \( x \) even more \( \leq \). For example, \( x = \{-1, B_2, A_3, D_4\} \), \( y = \{B_1, A_2, C_3, D_4\} \), \( f_s(x) = \{A_1, -2, B_3, D_4\} \), \( f_s(y) = \{A_1, B_2, C_3, D_4\} \). Thus, the partial order is maintained.

We should also note the cases when we need to cut-off the last position in a cache set, if it has grown bigger than its intrinsic size. For example, for a 2-way set associative cache, only 2 positions inside a cache set are allowed. Assume that the last position in \( x \) is block \( A \), and in \( y \) it is \( B \). If \( A = B \), the partial order is maintained after cut-off. If \( A \neq B \), it could happen that \( A \) also belongs to \( y \). But cutting off this value could only make \( x \) even more \( \leq \). For example, assume \( x = \{-1, D_2, A_3\} \) and \( y = \{D_1, A_2\} \) before cut-off. After cut-off, we have \( x = \{-1, D_2\} \) and the partial order is maintained. It could happen that \( B \) also belongs to \( x \). But this implies that \( x \not\leq y \), so we disregard this case.

**Distributive**

Our cache analysis is distributive, if our (cache simulation) transfer function \( f_{PrS} \) behaves, for all cache sets, such that \( f_{PrS}(x \land y) = f_{PrS}(x) \land f_{PrS}(y) \), \( \forall x, y \in V \). We will check if our (cache simulation) transfer function \( f_s \) is distributive for a single instruction \( s \), which could either generate a cache miss or hit. If this is true, we conclude that the transfer function \( f_{PrS} \) also must be distributive for a program segment \( PrS \), since \( f_{PrS} = f_{s_n} \circ f_{s_2} \circ \cdots \circ f_{s_1} \).

We can, however, show that our cache analysis is not distributive. Let us assume that \( x = \{A_1, B_2, C_3, D_4\} \), \( y = \{D_1, C_2, B_3, A_4\} \) and function \( f_s \) contains an instruction that references memory block \( A \). This results in \( x \land y = \{-1, -2, B_3, C_3, A_4, D_4\} \) and \( f_s(x \land y) = \{A_1, -2, -3, B_4, C_4, D_4\} \). Furthermore, \( f_s(x) = \{A_1, B_2, C_3, D_4\} \), \( f_s(y) = \{A_1, D_2, C_3, B_4\} \) and \( f_s(x) \land f_s(y) = \{A_1, -2, C_3, B_4, D_4\} \). Thus, \( f_s(x \land y) < f_s(x) \land f_s(y) \). This is the definition of monotone (but not distributive) frameworks, as defined in Section 5.4.3.
5.4. Verification of Cache Analysis

Conclusions
We have checked and validated our cache analysis for set associative caches, together with its cache meet operator $\wedge$ and cache simulation transfer function $f_{PrS}$. The analysis algorithm will converge and run in a finite amount of time, since the framework is monotone. However, the framework is not distributive. Therefore, the precision of the solution to the data flow equations is MFP. The MFP solution is always conservative and safe, since $MFP \leq MOP \leq Ideal$. 
5.5 Comparison

This section compares our approach with two previous cache analysis approaches. The first approach is based on cache conflict graphs, while the other approach is also based on data flow analysis.

5.5.1 Cache Conflict Graph

The previous approach with CCG and CSTG graphs often returns too conservative bounds [11]. This is a result of the fact the many additional functional constraints in the CFG must be annotated by the programmer manually. These manual annotations are necessary to tighten the WCET, but are time consuming to deduce. The introduction of program segments removes much of this manual annotation work, since program segments often contain several basic blocks.

Furthermore, the complexity of the CSTG graphs gets very large for caches with higher associativity. The potential number of cache states to model for an $n$-way set associative cache, with $m$ conflicting l-blocks mapped to a set, grows according to $\sum_{i=0}^{n} \frac{m!}{(m-i)!}$ [5]. This makes it time consuming to construct the graphs as well as solve the corresponding ILP-problems.

In fact, path clustering a CFG into program segments is not even compatible with the usage of CCG and CSTG graphs. Please remember that CCG and CSTG graphs require nodes associated to l-blocks. If these l-block nodes are replaced with program segments, the analysis approach would cease to be correct, since program segments may contain loops structures where cache lines are repeatedly replaced by others. This is not allowed in CCG and CSTG graphs, since the execution of a l-block by definition only results in a maximum of one cache miss.

However, if only basic blocks (not program segments) are used on mono-processor systems, the approach with CCG and CSTG graphs could produce better results. This is due to the fact that functional constraints are supported to reduce the number of possible execution paths. The general iterative data flow algorithm does not support functional constraints. Our cache analysis, which is an instance of this general data flow algorithm, has currently the same limitation.
For MPSoC-systems, CCG and CSTG graphs are not even an option. The formulation of ILP-problems with implicit path enumeration cannot guarantee any time bounds, since bus contention is not considered. The processor producing a cache miss may need to wait an undefined amount of time, if the bus is busy serving other processors.

### 5.5.2 Hybrid Cache Analysis

For mono-processor systems, during early years in the -00s, another approach called hybrid cache analysis [11] [12] [13] [24] was presented by Wolf et al. Their solution is also based on data flow analysis. However, this solution is primarily designed and implemented for use with ILP-formulations [13], which cannot be used on MPSoC. The analysis approach is performed in three distinct steps:

1. Program segments are simulated with empty \( \text{in}_{\text{set}}[PrS] \)-sets. The \( \text{gen}_{\text{set}}[PrS] \)- and \( \text{kill}_{\text{set}}[PrS] \)-sets are computed from these simulations.


3. The first miss or misses in a program segment, due to initial empty \( \text{in}_{\text{set}}[PrS] \)-set in the first step, are now possible to remove if the calculated \( \text{in}_{\text{set}}[PrS] \)-set suggests this.

They use the same methodology as for direct mapped caches, in the sense that the \( \text{gen}_{\text{set}}[PrS] \)- and \( \text{kill}_{\text{set}}[PrS] \)-sets are independent [13] on the \( \text{in}_{\text{set}}[PrS] \)-sets. These gen- and kill-sets are invariant and are computed before the data flow analysis starts.

However, their approach will result in too pessimistic solutions, since the computed gen- and kill-sets are invariant. Assume a 2-way set associative cache with LRU-policy, a program segment \( PrS_1 \) that has a single instruction within memory block \( C \) that maps to \( \text{set}_1 \) and \( \text{in}_{\text{set}1}[PrS_1] = \{A_1, B_2\} \). In reality, we would get \( \text{out}_{\text{set}1}[PrS_1] = \{C_1, A_2\} \) after execution of \( PrS_1 \). But it is difficult for the step before global data flow analysis to construct \( \text{gen}_{\text{set}1}[PrS_1] = \{C_1, A_2\} \) and \( \text{kill}_{\text{set}1}[PrS_1] = \{A_1, B_2\} \), since it
Figure 5.13: The necessary pessimism in the hybrid cache analysis approach.

is unaware of the in-sets. Even if it would be able to correctly construct gen- and kill-sets for this particular $in_{set1}[PrS_1]$-set, it will not work for others. Figure 5.13 illustrates the necessary pessimism with this approach. To sum up, the following improvements have been made to our cache analysis approach:

1. We use a special cache meet operator, optimized for producing a safe abstraction of a cache set during conjunction of two or more execution paths. They use an ordinary intersection operator for sets. If they hold LRU positions in each cache set, only memory blocks in the same position will survive during intersection. If they hold memory blocks in each set, without LRU positions, the program segment simulation (or similar transfer function) has to start with cache sets containing the worst LRU position for these blocks. Thus, this will also produce too pessimistic results.

2. In our approach, we initialize all $out_{set}[PrS]$-sets in each program segment, except the entry, to the universal set. They do not do this, and therefore start with empty $in_{set}[PrS]$-sets. In Section 5.3.2, we motivated that the results of using the universal set leads to less pessimistic solutions, although still conservative. The universal set also reduces the need for defining a specific iteration order of program segments, which was seen in Section 5.3.3.
3. In our approach for set associative caches, we do not use the gen-kill form. As previously noted, these $gen_{set}[PrS]$- and $kill_{set}[PrS]$-sets are invariant and must be too pessimistic. We use a cache simulation function as the transfer function for program segments, where these sets are not needed.

4. We rigorously provide answers to important questions such as correctness and precision of the solutions, as well as the convergence of our algorithms.
Chapter 6

Task Generator

In Chapter 4, we reviewed the problem with estimating the WCET on MPSoC. Essentially, the problem exists due to the fact that a single bus can be busy responding to other requests during cache misses. A safe bus scheduling algorithm that can guarantee time bounds was outlined. The solution included the selection of a bus access policy and the calculated bus access schedule, which was optimized for the analyzed system tasks. In Chapter 5, we designed two cache analysis approaches to help us predict cache hit and miss sequences for these tasks. The cache analysis result is later used by the bus scheduling algorithms.

Every task has different cache miss characteristics. How these characteristics influence the bus optimization algorithms ability to find an efficient bus access schedule has currently not been investigated. Perhaps certain bus access policies are not even suitable for certain classes of miss characteristics. Therefore, the researchers at ESLAB need to find a way to automatically generate randomized tasks to evaluate their bus optimization algorithms. A randomized task is a control flow graph (CFG) with descriptive data inside each node.
6.1 Limitations

It is of high importance that the CFGs, with the requested characteristics, are generated as randomized as possible. Ideally, there should be some probability greater than zero that every kind of well-structured task can be generated. The concept of a well-structured task includes tasks that are constructed with control structures that you ordinary use in high-level languages, like C, C++, C# and Java. These high-level control constructs are created by scoped statements like if, while, switch etc. Furthermore, well-structured tasks do not include the use of explicit jump statements.

The task generator program can only construct well-structured tasks without jumps between the hierarchical nodes (defined in Section 3.7.1). These jumps are often a result of explicit goto-, break- and continue-statements, or their equivalents in programming languages. This is not a big limitation, since the absence of such constructs often are considered as a desirable program trait. Furthermore, some timing analysis tools, like SymTA/P from the University of Braunschweig [7], also put this requirement on the tasks as well. The requirement simplifies data flow analysis that checks for input data dependency [6] within tasks.

6.2 Task Structure

The task generator constructs a CFG that contains program segments and other related control structures. The control flow is generated using structural constructs, such as if, if-then, if-then-else and loop with check of condition at entry or exit. The structural blocks are put inside different hierarchical levels. Both the selection of blocks and the use of hierarchical levels are randomized.

6.2.1 Structural Blocks

The task generator is designed for use with analysis tools compatible with SymTA/P. As a consequence, the program needs to support five different node types in the generated CFG.

1. The root node, also called the start node.
2. The **sink** node, also called the exit node.

3. The **basic** node represents one possible path through a program segment. This node contains the sequence of cache misses, which is represented by the clock cycle at which each miss occurs.

4. The **control** node represents a branch condition with more than one outgoing edge.

5. The **loop** node is like a **control** node, but with the additional requirement of having an associated *loop bound* to limit the number of possible loops.

The structural blocks are the building elements of a generated task. Each structural block contains one or several nodes, and each node is represented by one of the types defined above. Almost all imperative languages contain control flow statements with equivalent semantics to the structural blocks that this program generates. Figure 6.1 illustrates the different structures that can be generated.

1. The **sequence** construct results in a path with sequential machine instructions that can generate cache misses. Figure 6.1(a) shows that the structural block 2 has been extended with sequential nodes 1 and 3. The latter two nodes are basic nodes.

2. The **if-then** construct results in a customary if-then statement with conditional branching. Figure 6.1(b) shows that the structural block 2 has been expanded with branch condition logic from nodes 1 and 3. The latter two nodes are control nodes.
3. The *if-then-else* construct results in a customary if-then-else statement with conditional branching. Figure 6.1(c) shows that the structural block 2 has been expanded with branch condition logic from nodes 1 and 4. During expansion of block 2, it is also necessary to, on demand, create a structural block 3 that represents the else-branch. Nodes 1 and 4 are control nodes.

4. The *loop* with *pre-condition* construct results in loop logic with a termination check before each execution of the loop. Figure 6.1(d) shows that block 2 has been expanded with condition logic from node 1. The latter node is a loop node with an associated bound.

5. The *loop* with *post-condition* construct results in loop logic with a termination check after each execution of the loop. Figure 6.1(e) shows that block 1 has been expanded with condition logic from node 2. The latter node is a loop node with an associated bound.

### 6.2.2 Hierarchical Levels

Each node, and its associated structural block, need to be placed somewhere among the hierarchical levels. The level grows when structural blocks are nested inside others. This is a sign of *structural complexity*. One approach would be to generate the graphs in a top-down fashion, from entry node to exit node. This approach has, however, many drawbacks and is hard to implement. Our task generator primarily constructs CFGs *inside-out*. In other words, the graphs are mostly expanded into higher hierarchical levels. Figure 6.2 illustrates how a graph can be expanded in four steps.

1. Figure 6.2(a) shows a graph that contains ordinary if-then-else control logic. Each subsequent expansion step is illustrated with additional nodes having dots inside them.

2. The graph is expanded with a pre-condition logic loop, as shown in Figure 6.2(b). The previous graph is inside the loop body.

3. The graph is further expanded with if-then control logic, as shown in Figure 6.2(c). The previous graph is inside the then-branch.
4. The graph is further expanded with two sequential nodes before and after the previous graph, as shown in Figure 6.2(d).

This approach is straightforward and easy to implement, since the task generator never needs to manage structural blocks that are not finalized. Another advantage is that it is not possible to join paths that are inside different hierarchies, which could imply that the task is not well-structured.

The CFGs should, however, not only be constructed with the help of graph expansions. In that case, the result would be extremely complex and wide graphs with little correspondence to real graphs. For more realistic graphs, the expansion model must be improved. Therefore, the task generator manages three types of hierarchies:

1. **Local hierarchy** - All structural blocks in the program are expanded around a local hierarchy, as seen in Figure 6.2. The current local hierarchy is represented by a *local CFG* inside the task generator.

2. **Global hierarchy** - This hierarchy is used as a placeholder of one or more already constructed local hierarchies. The global hierarchy is used for sequential extension (soon explained why and how). The current global hierarchy is represented by a *global CFG* inside the task generator.
3. *Task hierarchy* - This hierarchy is used as a placeholder of an already constructed hierarchy of structural blocks, which is forbidden to be used for further graph expansion. Otherwise, the maximum graph breadth can be violated. The structure inside a task hierarchy can only be placed in sequence to another structure. If the graph breadth reaches its limit, the current local CFG is sequentially copied to the global CFG, which is also sequentially copied to the task hierarchy. Afterwards, the local and global CFG are cleared. Therefore, the entire task graph will never exceed the maximum allowed graph breadth. The task hierarchy is represented by a *task CFG* inside the task generator.

As previously noted, the graph expansion is always performed around the current local hierarchy. However, before a local hierarchy can be further expanded, one of three different types of *hierarchical preparations* is carried out.

- **Use local hierarchy** - This is the easiest preparation, since nothing special needs to be prepared. The current local hierarchy remains to be used for the next graph expansion.

- **New local hierarchy** - The current local hierarchy is placed in sequence to the global hierarchy. Afterwards, a new local hierarchy is started from scratch, with just a basic node inside. Figure 6.3(a) illustrates how the entry and exit nodes are selected for the new global hierarchy.

- **Use global hierarchy** - The current local hierarchy is placed in sequence to the current global hierarchy. The difference to the previous point is that the global hierarchy is now copied back to the local hierarchy. Afterwards, the global hierarchy is cleared. The effect is that the task graph structures become more randomized, since already created sequential hierarchies together can become a child of a new hierarchy. Figure 6.3(b) illustrates how the entry and exit nodes are selected for the new local hierarchy.
6.3 Randomization Process

The task generator constructs a CFG that represents a task with a given set of characteristics concerning the distribution of cache misses and the complexity of the program flow. The invoker must be able to influence the characteristics of the generated tasks. The characteristics that can be affected are the following:

1. The size of the CFG, in terms of the number of basic nodes and their execution time (clock cycles).
2. The complexity of the CFG. In essence, this is requested by the probability of the existence of different types of structural blocks and the breadth of task graph.
3. The average distance between the cache misses inside a basic node.

Figure 6.3: Different preparation of hierarchies.
4. The standard deviation of the distance between cache misses.

It should be noted that some of the characteristics are not specified explicitly via parameters, but rather through probability tables. For example, the average distance between cache misses is constructed through a probability table associated with a random variable. When the task generator is invoked, it will calculate the average value ($\mu$) and the standard deviation ($\sigma$), according to Section 2.6, and as a convenience print them on the screen. Therefore, two invocations with the same parameters (probability functions) are highly likely to generate different tasks, but both with the same requested characteristics.

At the expense of a somewhat more unwieldy interface, the probability tables let you customize the task characteristics more fine-grained than otherwise would be possible. This is due to the fact that an infinite number of probability tables for a specific average and standard deviation exist. For example, the left probability table below generates (approximately) the same average value and standard deviation as the table to the right:

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$p_X(x_i)$</th>
<th>$x_i$</th>
<th>$p_{X_i}(x_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.25</td>
<td>1.838</td>
<td>0.25</td>
</tr>
<tr>
<td>4</td>
<td>0.25</td>
<td>5.0</td>
<td>0.5</td>
</tr>
<tr>
<td>6</td>
<td>0.25</td>
<td>8.162</td>
<td>0.25</td>
</tr>
<tr>
<td>8</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\mu=5, \sigma=2.24$ $\mu=5, \sigma=2.24$

The invoker might be interested in to just pass an average value and standard deviation and get help from the program to actually compute an instance of a probability table that matches the requested parameters. This is a feature that can be added in a future version.

### 6.3.1 Discrete Random Variables

The task generator uses the following kind of discrete random variables for randomization purposes:

1. $X_{gb}$ - The maximum breadth of the CFG.

2. $X_{sb}$ - The type of structural block to generate, for example if–then–else control logic, according to Section 6.2.1.
3. $X_{hl}$ - To which hierarchical level the selected structural block will be attached. The hierarchical level is prepared and selected according to Section 6.2.2.

4. $X_{cc}$ - The length in clock cycles of each basic node.

5. $X_{cm}$ - The number of clock cycles until next cache miss within a basic node.

6. $X_{lb}$ - The loop bound associated to every loop node.

The first three discrete random variables affect the structure of the task. The fourth and fifth variable influence the cache miss behavior. Random variables $X_{gb}$, $X_{lb}$ and $X_{cc}$ are randomized from a uniform distribution of finite set of values. The rest of them, $X_{sb}$, $X_{hl}$, and $X_{cm}$, can be randomized from a non-uniform distribution. There normally exists good support for (pseudo-)randomization from discrete uniform distributions in different programming libraries. However, the randomization from a non-uniform distribution is more seldom supported.

Random variables originating from non-uniform distributions can be implemented using the alias method [17], which is also used in the task generator. The alias method is implemented by initializing the probability function inside an alias table. Figure 6.4 shows a non-uniform discrete distribution with a corresponding alias table. A random variable is sampled by throwing “uniform darts” into this table. Basically, all that is needed for each new sample is two generate $U_1, U_2 \sim \text{unif}(0,1)$, use $U_1$ in a multiplication, and to use $U_2$ in a comparison. Consequently, to implement the alias method, it is sufficient to have support for uniform distributions. The exact algorithm of the alias method is out of the scope of this thesis, and it is enough to conclude that there exist efficient ways to generate discrete random variables associated with a probability function. Please refer to Kronmal et al [17] for algorithm details and proof.

### 6.3.2 Cache Misses

The randomization of the cache misses inside each basic node is modeled by the random variables $X_{cc}$ and $X_{cm}$. Variable $X_{cc}$ is controlled by a passed
6.3. Randomization Process

minimum and maximum clock cycle length. $X_{cm}$ is also controllable by the invoker, since the probability function of $p_{X_{cm}}(x)$ can be defined. For example,

\[
\begin{array}{c|c|c|c|}
 x & p_{X_{cm}}(x) & x & p_{X_{cm}}(x) \\
 2 & 0.10 & 7 & 0.10 \\
 3 & 0.10 & 8 & 0.10 \\
 4 & 0.10 & 9 & 0.10 \\
 5 & 0.10 & 10 & 0.10 \\
 6 & 0.10 & 11 & 0.10 \\
\end{array}
\]

defines that there is 10% probability that a cache miss will occur in each of the clock cycle values between 2 and 11. No cache miss can occur during the first clock cycle, or after more than 11 clock cycles compared to the previous one. In real cases, this probability function will be non-uniform and with more mappings from a clock cycle value to a certain probability. The generation of cache misses, for each basic node, continues until the total number of clock cycles exceed the value sampled from $X_{cc}$.

6.3.3 Control Flow Graph

The randomization of complexity in the control flow graph is modeled by the random variables $X_{gb}$, $X_{sb}$ and $X_{hl}$. Variable $X_{gb}$ is controlled by a passed minimum and maximum graph breadth. The last two variables are also controllable by the invoker, since the probability functions of these can be defined. The below probability tables are defined as default in the task generator.
6.4 Generation Algorithm

Algorithm 10 outlines how a randomized CFG of a task is constructed. The number of basic nodes \((num_{bn})\) in the CFG is passed by the invoker. The loop at line 1 continues until this number of basic nodes have been created. At lines 2-12, we prepare a local hierarchy, according to the sampled value of \(X_{hl}\). The method used for preparation of hierarchies is described in Section 6.2.2 and Figure 6.3. There exists a special case for use local hierarchy and use global hierarchy that occurs if the breadth of the current local or global CFG has reached the maximum allowed graph breadth. This forces the generator to copy the graph to the task CFG, which is forbidden to be used for further graph expansion. Afterwards, both the local and global CFG are cleared.

Remember that the task generation handles three hierarchies (local, global and task CFG), but it is only the local CFG that actually is expanded. The graph expansion is performed at lines 13-23, which has been described in Section 6.2.2 and with the help of Figure 6.1 and 6.2. The structural block to use when expanding the graph is sampled from \(X_{sb}\). At line 21, we handle the construction of if-then-else logic. Since the current local hierarchy only holds the then-branch, we also need to, on demand, construct the else-branch, before the expansion can continue. At the end of the loop, at line 24, new basic nodes are concatenated before and (or) after to the current local hierarchy. How many clock cycles a basic node represents, and the distance between each cache miss, are sampled from \(X_{cc}\) and \(X_{cm}\), respectively. At lines 26-27, the algorithm finalizes the CFG of the task by concatenating the current local CFG to the global CFG, which is concatenated to the task CFG.

<table>
<thead>
<tr>
<th>(x_{sb})</th>
<th>(p_{X_{sb}}(x))</th>
<th>(x_{hl})</th>
<th>(p_{X_{hl}}(x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>0.10</td>
<td>New local hierarchy</td>
<td>0.40</td>
</tr>
<tr>
<td>Loop (pre)</td>
<td>0.30</td>
<td>Use local hierarchy</td>
<td>0.40</td>
</tr>
<tr>
<td>Loop (post)</td>
<td>0.15</td>
<td>Use global hierarchy</td>
<td>0.20</td>
</tr>
<tr>
<td>If-then</td>
<td>0.10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>If-then-else</td>
<td>0.35</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.5 Generation Example

Figure 6.5 illustrates a task CFG that has been generated by the task generator. The generator has been invoked with the following parameters:

- The number of basic nodes are 8.
- The clock cycle length of each basic node is between 20 to 100 (uniform distribution).
- The loop bound in each loop node is between 2 to 100 (uniform distribution).
- The maximum allowed breadth of the graph is 10.
- The cache miss distribution is according to the probability table in Section 6.3.2.
- The complexity of the CFG, and the distribution of different types of structural blocks, are according to the probability table in Section 6.3.3.

The generation of nodes starts with $id:0$ and continues in ascending order. The nodes are marked with id number and type (entry, exit, basic, control or loop). Each basic node is annotated with the generated cache misses, and each loop node is annotated with the generated loop bound. The control nodes represent conditional branch logic. The first generated structural blocks are if-then-else, loop with pre-condition, if-then-else again and so on, as seen in Figure 6.5. It can also be observed that no if-then block has been sampled in this example. For each new structural block at least one basic node is concatenated.
Input: Random variables $X_{gb}, X_{sb}, X_{hl}, X_{cc}, X_{cm}, X_{lb}$ and $num_{bn}$.
Output: Control flow graph for a task.

while number of basic nodes $\leq$ $num_{bn}$ do

switch $X_{hl}$ do

<table>
<thead>
<tr>
<th>case New Local Hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenate current local hierarchy to current global hierarchy, as in Figure 6.3(a).</td>
</tr>
<tr>
<td>Start a new local hierarchy, and use $X_{cc}, X_{cm}$ to put a basic node into this hierarchy.</td>
</tr>
<tr>
<td>case Use Local Hierarchy</td>
</tr>
<tr>
<td>Do nothing - continue expanding the local hierarchy.</td>
</tr>
<tr>
<td>case Use Global Hierarchy</td>
</tr>
<tr>
<td>Concatenate current local hierarchy to current global hierarchy, as in Figure 6.3(b).</td>
</tr>
<tr>
<td>Make global hierarchy become a new local hierarchy.</td>
</tr>
</tbody>
</table>
end

switch $X_{sb}$ do

<table>
<thead>
<tr>
<th>case Pre-Test loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expand local hierarchy with pre-condition loop (use $X_{lb}$).</td>
</tr>
<tr>
<td>case Post-Test loop</td>
</tr>
<tr>
<td>Expand local hierarchy with post-condition loop (use $X_{lb}$).</td>
</tr>
<tr>
<td>case If-Then</td>
</tr>
<tr>
<td>Expand local hierarchy with if-then logic.</td>
</tr>
<tr>
<td>case If-Then-Else</td>
</tr>
<tr>
<td>Create else-branch and expand local hierarchy with if-then-else logic.</td>
</tr>
</tbody>
</table>
end

Concatenate basic nodes to local hierarchy (use $X_{cc}, X_{cm}$).

Concatenate current local hierarchy to global hierarchy.

Concatenate current global hierarchy to task hierarchy.

Algorithm 10: Generation of a randomized task.
Figure 6.5: Randomized control flow graph.
Chapter 7

Implementation and Usage

This thesis has resulted in the implementation of three tools.

- Cache Analyzer for MPSoC, also working for mono-processor systems.
- Task Generator for creating randomized tasks, which are used to evaluate bus schedule algorithms on MPSoC.
- Graph Converter for converting a proprietary format to GraphML (standard) format.

7.1 C++ and Boost

The above tools have been implemented in C++ for the following reasons:

1. The tools do not need any graphical user interface, and therefore the advantage of GUI-programming in Java is no argument. Only a simple command line interface is needed.

2. The cache analyzer needs to be integrated into SymTA/P, which is already based on C++ code. This tool can easily be compiled as a
static or shared library, usable from other analysis tools in C++ (as many of them are written in).

The implementation of the cache analysis and task generation program is dependent on Boost. Most libraries in Boost are delivered in the form of header files containing templates and inline functions. Boost is a set of free C++ libraries that can operate on many platforms and that many compilers support. The Boost Graph Library (BGL) is used in the implementations to efficiently represent and manipulate control flow graphs (CFGs). The cache analysis program needs to read (and update) graphs from files in the GraphML-format. This requires the ability to parse XML-files, which is not natively supported in Boost. Therefore, another library called Expat is also required. Some of the advantages BGL offers are:

1. The efficient adjacency list structure for graphs can be used. The cache analyzer frequently needs to find all vertices (nodes) adjacent to another vertex, which this data structure is particularly good at.

2. We can associate application-specific data to nodes and edges, for example cache misses or cache states to basic nodes.

3. Many built-in algorithms, like breadth_first_search, are provided in the library.

4. Export and import to the standardized GraphML-format are supported. This introduces a potential for other analysis tools to be able to read and understand the generated output.

### 7.2 Cache Analyzer

The cache analyzer program analyzes a CFG in GraphML-format, annotated with the following information:

1. The memory accesses performed in each basic node (or program segment). These accesses can be both data and instruction addresses.

2. The size of each memory access. This makes it possible to include unpredictable memory accesses within a certain possible range.
3. A flag that tells whether each memory access is a read or write.

When a solution to each $in_{cache}[PrS]$- and $out_{cache}[PrS]$-set for every program segment has been found, the result is written to an output file in GraphML-format. An example of the cache analysis result for node 3, from Figure 5.6 in Section 5.3.4, is given below. $key0$ and $key4$ contain the input with the memory access addresses and their range, respectively. $key3$ represents the node type, which is zero for basic nodes. $key1$ contains the loop bound, but is set to 0 since this is no loop node. $key5$ specifies whether each memory access is a read (0) or write (1). At $key2$, the cache analyzer has output the predicted cache hit and miss sequence in exact order (0 = hit, 1 = miss).

```
<node id="n3">
  <data key="key0">80 90 24</data>
  <data key="key1">0</data>
  <data key="key2">1 0 0</data>
  <data key="key3">0</data>
  <data key="key4">1 2 4</data>
  <data key="key5">0 0 0</data>
</node>
```

7.2.1 Usage

The following parameters can be passed as command line parameters to a terminal (console) program:

- The size of the main memory in bytes. (-ms)
- The size of the cache memory in bytes. (-cs)
- The set associativity of the cache memory. If the associativity is one, the cache memory is direct mapped. Therefore, no specific analysis for direct mapped caches has been implemented. (-assoc)
- The size of a memory block that can fit into a cache line. (-bs)
- The input file (in GraphML-format) containing a CFG to analyze. (-infile)
• The output file (in GraphML-format) with the results. (-outfile)

7.3 Task Generator

The task generator program generates a randomized task according to the requested task characteristics. The result can be output in the following three file formats:

1. **GraphML** format. A standard format for (any kind of) graphs inside an XML-file.

2. **vcg** or **gdl** format. A standard format for (any kind of) graphs inside a VCG specification. The **gdl** format is based on **vcg**. Many tools support visualizing graphs, and in particular CFGs, for example the **xvcg** utility and **aiSee** from AbsInt.

3. A customized format used in analysis tools at ESLAB.

7.3.1 Usage

The requested task characteristics are passed as command line parameters to a terminal (console) program:

• Total number of basic nodes in the CFG. (-nodes)

• The program segment length in unit of clock cycles, expressed as an interval. (-nl low..high)

• The loop bound value associated with generated loop structures, expressed as an interval. (-lb low..high)

• The maximum breadth of the CFG. (-breadth)

• The complexity of the CFG, represented by the probability that certain preparations of the local hierarchy are sampled. (-hier)
• A file name, whose contents define a probability table for cache misses. The file should contain two columns, $x$ and $p_X(x) = \Pr\{X = x\}$, where $x$ is the cache miss distance, and $p_X(x)$ is the probability that this distance will occur after a previous one. (-camt)

• A file name, whose contents define a probability table for different structural blocks. The file should contain two columns, $x$ and $p_X(x) = \Pr\{X = x\}$, where $x$ is an integer constant representing a structural block, and $p_X(x)$ is the probability that this particular structural block will be created in the CFG. (-comt)

• The output file to write the randomized task CFG to. (-out)

By having the probability tables defined in files, it simplifies the command line interface, as well as making it easier to reuse the probability functions for future tests. The mean and standard deviation of the probability functions are printed out at the start of each execution.

### 7.4 Graph Converter

A simple tool has been constructed that converts the internal graph format used at ESLAB to GraphML-format. This was necessary to be able to analyze already existing CFGs in the internal format. The parser is a simple recursive descent parser.

### 7.5 WCET Analysis

SymTA/P is an analysis framework designed for mono-processor systems. Figure 7.1(a) illustrates the different modules that are integrated to a single tool. The invoker passes the C source code of a task, together with input data and functional constraints to the tool. The input data must be put together such that all program segments of the task are simulated or measured. If any program segment is not included in the execution paths, the segment is marked for investigation. The functional constraints are used to further restrict the feasible execution paths.
The first performed step is the path clustering (defined in Section 3.7.1), which extends basic blocks to program segments. These program segments are passed to the cache analysis and time measurement module. The cache analysis produces cache miss counters [24] for each segment and passes these to the time measurement module. This module is now able to produce the cost of each program segment. Finally, the program segment costs, structural constraints and functional constraints are passed to an ILP solver for maximization. The returned value is the WCET of the task.

ESLAB has extended this framework to support WCET analysis on MPSoC. Figure 7.1(b) illustrates the new (shadowed) modules that are integrated into this tool. Just as before, the invoker passes the C source code, input data and functional constraints. However, the original cache analysis module is replaced with our own module supporting MPSoC. The task information passed to the cache analysis is converted into GraphML-format by the Graph Converter tool. The cache hit and miss sequence of each program segment is passed to the bus algorithms, described in Chapter 4. The bus algorithm module simultaneously selects optimized bus segment schedules and calculates the WCET. Thus, an ILP solver is no longer needed.

Figure 7.1: WCET calculation for mono- and multi-processor systems.
Chapter 8

Discussion

We started to review how static timing analysis can be performed on monoprocessor systems. However, as the hardware in the systems gets more complex, the previous approaches become invalidated. For example, the described approach with implicit path enumeration cannot be performed on MPSoC. Furthermore, cache analysis for both direct mapped and set associative caches needs to be designed with these systems in mind, where the exact cache hit and miss sequence is delivered. This approach was constructed and verified in Chapter 5. A comparison to previous approaches was finally conducted, and some improvements were pointed out. We also showed why set associative caches with replacement policy FIFO and LFU are much harder to model safely. Therefore, only LRU is currently supported. The cache analysis has been implemented as an independent module, with the possibility to be used in different timing analysis tools.

A tool that generates randomized tasks has also been designed and implemented. This tool supports parameters that influence the complexity of the task, as well as the cache miss behavior. The output from the tool will be used in future studies to evaluate the efficiency of different bus access policies, introduced in Chapter 4. These studies may result in a need to introduce new bus access policies optimized for certain tasks.
8.1 Future Work

The following improvements of the implemented programs are natural:

- In chapter 5, we constructed cache analysis for direct mapped caches and set associative caches with LRU as replacement policy. In general, LRU is an efficient replacement policy in terms of cache hit ratio. As previously noted, no solution for set associative caches with FIFO and LFU was provided. However, the usage of such cache memories is common. Therefore, future work needs to look further into safe cache analysis for these memories.

- The cache analysis is primarily designed for memory hierarchies with two levels. Processor architectures with two and three levels of cache are getting more common nowadays. For these architectures, the designed algorithms need to be extended.

- For some random variables, the task generator uses an associated probability table to randomize characteristics. However, the invoker may not have time and interest to define a probability table, but finds it sufficient to pass the average value and standard deviation of some random variable. Future versions could simplify the usage by automatically constructing an instance of a probability table for a certain average and standard deviation.

The task generator program has been constructed to evaluate how cache misses affect the WCET of a real-time application on MPSoC.

- In Chapter 4, an algorithm for generating bus access schedules was introduced. The algorithm assumes that the nodes in the task graph already have been mapped to the different processors in the system. Task mapping is usually performed without taking the cache miss characteristics into consideration. Intelligent algorithms for the mapping process need to be designed.

- It would be interesting to investigate how tasks with identical cache miss distance, but varying standard deviation, will influence WCET with the current bus access algorithms, introduced in Chapter 4.
• It would be interesting to find out some typical classes of task characteristics. Therefore, the cache miss characteristics of already existing tasks need to be extracted and analyzed.

Static timing analysis for hard real-time systems has many issues to solve in the future. The problems listed below are related to how the WCET can be further tightened.

• As already noted in Chapter 4 and Section 7.5, the ILP-formulations have been abandoned for MPSoC-systems. The reason is that the bus algorithm module must handle explicit execution paths and their cache hit and miss sequences, since the bus contention at each memory access must be taken into consideration. Functional constraints that can be specified together with ILP-problems are currently not supported. For example, it is not possible any longer to specify the execution count for a branch in terms of other branches. Future enhancements should include this possibility to be able to produce more tight WCETs.

• As noted in Section 3.7, a safety margin between each basic block or program segment needs to be added. This safety margin might be possible to further reduce with the help of data flow analysis considering the possible pipeline states.

• In Section 3.7, we showed how basic blocks were extended to program segments and why the safety margin was no longer needed between each basic block. For this to work without problems, the CPU can only use very simple branch predictions, for example taken and not taken. More advanced branch predictions make use of the execution history and are much harder to accurately model during program segment simulation.
Bibliography


Appendix A

Compilation of Programs

This appendix describes how to set up Boost and Expat, to be able to compile the implemented tools. Please note that the instructions are just recommendations that have worked for us. The programs have been compiled with Boost 1.35 and Expat 2.0.1. They have also been tested on both Linux and Windows.

A.1 Boost Graph Library

Both the cache analysis and task generator program need to associate arrays to each node in a control flow graph. To parse this graph data has proved to be difficult and resulted in exceptions. To fix this problem, please do the following:

1. Open `%BOOST_BUILD_PATH%/boost/lexical_cast.hpp`.

2. Lookup all occurrences of `stream.unsetf(std::ios::skipws)` and comment them out. This will allow space as a separator between array elements.
A.1.1 Fix on Windows

The programs need to be able to interpret data files with GraphML contents. Therefore, the Boost Graph Library function `read_graphml` needs to be invoked. Unfortunately, this function was not exported during builds of the graph library into the destination DLL on our Windows machine. The solution was to mimic how the function `read_graphviz` is defined. To fix the problem, please do the following:

1. Open `%BOOST_BUILD_PATH%\libs\graph\src\graphml.cpp`.

2. Insert the following define before the first inclusions:
   
   ```
   #define BOOST_GRAPH_SOURCE
   ```

3. Replace the following lines:
   ```
   void
   read_graphml(std::istream& in, mutate_graph& g)
   ```
   with:
   ```
   BOOST_GRAPH_DECL void
   read_graphml(std::istream& in, mutate_graph& g)
   ```

4. Open `%BOOST_BUILD_PATH%\boost\graph\graphml.hpp`.

5. Insert the contents below after the first inclusions.

   ```
   #ifdef BOOST_HAS_DECLSPEC
   # if defined(BOOST_ALL_DYN_LINK) ||
       defined(BOOST_GRAPH_DYN_LINK)
   # ifdef BOOST_GRAPH_SOURCE
   # define BOOST_GRAPH_DECL __declspec(dllexport)
   # else
   # define BOOST_GRAPH_DECL __declspec(dllimport)
   # endif // BOOST_GRAPH_SOURCE
   # endif // DYN_LINK
   #endif // BOOST_HAS_DECLSPEC
   ```
A.2 Build on Linux

We start to configure Expat. Go to http://expat.sourceforge.net/ and follow the download links. After download completion, extract the package into $HOME/expat-2.0.1. Then, continue with the following commands in a terminal:

```
cd ~/expat-2.0.1
mkdir ~/boost_1_35_0/expat
./configure --prefix=$HOME/boost_1_35_0/expat
make install
```

This puts the header and library files in $HOME/boost_1_35_0/expat. Let us move on to Boost. Go to www.boost.org and follow the download links for Linux. After download completion, extract the package into $HOME/boost_1_35_0. Then, continue with the following commands in a terminal:

```
BOOST_ROOT=$HOME/boost_1_35_0
EXPAT_INCLUDE=$HOME/boost_1_35_0/expat/include
EXPAT_LIBPATH=$HOME/boost_1_35_0/expat/lib
export BOOST_ROOT
export EXPAT_INCLUDE
export EXPAT_LIBPATH
cd $BOOST_ROOT/tools/jam/src
sh ./build.sh
cd $BOOST_ROOT/libs/graph/build
$BOOST_ROOT/tools/jam/src/bin.linuxx86/bjam
```

Finally, the programs can be compiled by just invoking make in the directory containing the source code for the implementations. If the compile fails, please check the paths to the libraries inside the makefile.

A.3 Build on Windows

We start to configure Expat. Go to http://expat.sourceforge.net/ and follow the download links. After download completion, run the setup and
install library into C:\Program Files\Expat 2.0.1. Then, continue with the following actions:

1. Open C:\Program Files\Expat 2.0.1\Source\expat.dsw in your Visual Studio environment.

2. Build the solution.

This creates the static and/or dynamic libraries that are needed for XML-parsing. Let us move on to Boost. Go to www.boost.org and follow the download links for Windows. After download completion, run the setup and install the library into C:\Program Files\boost_1_35_0. Then, continue with the following commands in a command prompt:

```bash
set BOOST_BUILD_PATH=C:\Program Files\boost_1_35_0
set EXPAT_ROOT=C:\Program Files\Expat 2.0.1
set INCLUDE=%EXPAT_ROOT%\Source\lib;%INCLUDE%
set EXPAT_INCLUDE=%EXPAT_ROOT%\Source\lib
set EXPAT_LIBPATH=%EXPAT_ROOT%\Source\win32\bin\Debug
set INCLUDE=%EXPAT_ROOT%\Source\lib;%INCLUDE%
cd "%BOOST_BUILD_PATH%\tools\jam\src"
build.bat
cd "$BOOST_BUILD_PATH\libs\graph\build"
"BOOST_BUILD_PATH%\tools\jam\src\bin.ntx86\bjam"
```

Finally, the programs can be compiled with Visual Studio. If the compile fails, please check the paths to the libraries within the projects.
Copyright

Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtid ansättare - under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera åktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnad som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådan sammanhang som är krävande för upphovsmannens litterära eller konstnärliga anseende eller egenart. För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Carl-Fredrik Neikter