FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Time-predictable Memory Hierarchy

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht von

Bekim Chilku
Matrikelnummer 0928443

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner

Diese Dissertation haben begutachtet:

(Prof.Dr. Björn Lisper) (Ao.Prof.Dr. Martin Schöberl)

Wien, 20.08.2018

(Bekim Chilku)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Time-predictable Memory Hierarchy

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Bekim Chilku
Registration Number 0928443

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner

The dissertation has been reviewed by:

(Prof.Dr. Björn Lisper) (Ao.Prof.Dr. Martin Schöberl)

Wien, 20.08.2018

(Bekim Chilku)

Erklärung zur Verfassung der Arbeit

Bekim Chilku
Schottenfeldgasse 1/20, 1070 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Acknowledgements

This thesis is the result of my academic research at the Institute for Computer Engineering, Department for Real-Time System, at Vienna University of Technology.

At first I would like to express my gratitude to my supervisor Peter Puschner, who gave me the opportunity to do my research in the area of Real-Time Systems. The discussions and suggestions from him have been very encouraging and supportive during my research.

Next, I would like to thank Martin Schöberl and his team for the insightful discussions and the friendly work environment I received during my stay at Technical University of Denmark. Their constructive comments incented me to further widen my research.

Furthermore, I would like to thank my fellow colleagues from the Institute of Computer Engineering for the support, collaboration and the positive work atmosphere they provided during my stay at the Institute. I also want to thank Daniel Prokesch, who implemented the single-path code analysis as part of the toolchain.

And last but not least, I want to praise my family and friends for their support during my years of studies in Vienna. Especially, I would like thank my parents for the emotional and spiritual support throughout writing this thesis.

Abstract

Computing the Worst-Case Execution Time (WCET) of a task becomes mandatory when timing guarantees on task completion deadlines have to be given. Unfortunately WCET computation is a complex undertaking, especially for systems that use caches, out-of-order pipelines, and control speculation. The state-of-the-art WCET tools are avoiding the complexity problem by substituting the real hardware with abstracted models. However, abstraction leads to lots of unclassified model states, which in turn results in overly pessimistic WCET bound and poor processor utilization.

Single-path code is another alternative to eliminate complexity on timing analysis by transforming the conventional code into a code that has single execution trace. The approach converts all input-dependent alternatives of the code into pieces of sequential code as well as loops with input-dependent termination condition into loops with constant execution count, thus eliminating all control-flow induced variations in execution time. For obtaining the information about the timing of the code, it is sufficient to run the code once and measure the time. The major drawback of the single-path approach is its potential to end up with a quite long execution time.

In this work we address the problem of long execution time of single-path code. In particular, we focus on narrowing the speed gap between processor and main memory, by proposing a new prefetcher that brings instructions into the cache before they are required. The prefetcher exploits the time-predictable properties of single-path code to accurately predict the target of each prefetch request without polluting the cache at any moment. Another advantage of the prefetcher is its efficiency by issuing prefetch request for every possible cache miss that can occur during the runtime of the code.

In order to make the system design composable and compositional, we propose a new memory hierarchy that provides stable timing through the execution of the code. Although single-path code always runs through the same sequence of instructions, the timing of instructions can vary due to dependencies on the memory hardware states. In order to achieve stability on execution time we need to have repeatability on the history of the hardware states on each layer of the memory hierarchy. Therefore, we have defined a new memory-hierarchy organization that forces the sequence of the hardware states in the memory hierarchy to be repeatable for any iteration of the code. The memory hierarchy is also adapted to allow fetching and prefetching processes to work in parallel without interfering with each other in order to reach the best performance.

To demonstrate the applicability of the new concept, we have implemented the architecture of the system with the new memory hierarchy on an FPGA board and run experimental evaluation. The results prove the benefits that can be achieved in timing performance for single-path code.

Kurzfassung

Die Berechnung der maximalen Ausführungszeit (WCET) eines Tasks ist unumgänlich, wenn Zeitgarantien für die Beendigung von Tasks gegeben werden müssen. Leider stellt eine WCET-Berechnung ein komplexes Unterfangen dar, insbesondere für Systeme die Caches, Out-of-order-Pipelines und spekulative Ausführung nutzen. Die neuesten WCET-Tools vermeiden das Komplexitätsproblem indem sie die Hardware durch abstrakte Modelle ersetzen. Allerdings führt Abstraktion zu vielen nicht klassifizierten Modellzuständen, die wiederum überpessimistische WCET-Schranken und eine schlechte Ausnutzung des Prozessors ergeben.

Single-Path-Code ist eine Alternative, die die Komplexität der Zeitanalyse beseitigt, indem konventioneller Code in einen Code mit einem einzigen Ausführungspfad umgewandelt wird. Diese Herangehensweise wandelt alle eingabeabhängigen Alternativen des Codes in Teile eines sequentiellen Codes um und transformiert alle Schleifen mit eingabeabhängigen Abbruchbedingungen in Schleifen mit konstanter Anzahl von Ausführungen, so dass folglich alle von Kontrollflüssen bedingten Schwankungen in der Ausführungszeit eliminiert werde. Um Informationen über die Laufzeit des Codes zu erhalten, genügt eine einmalige Ausführung und Messung der Zeit. Der große Nachteil des Single-Path-Ansatyes besteht in einer potenziell sehr langen Ausführungszeit des resultierenden Codes.

Diese Arbeit beschäftigt sich mit dem Problem der langen Ausführungszeit von Single-Path-Code. Insbesondere wird ein neuer Prefetcher vorgestellt, der Instruktionen in den Cache lädt bevor diese gebraucht werden, um den Geschwindigkeitsunterschied zwischen Prozessor und Hauptspeicher zu verringern. Der Prefetcher nutzt die vorhersagenden Eigenschaften von Single-Path-Code, um das Ziel der nächsten Prefetch-Anfrage genau vorherzubestimmen, ohne den Cache zu korrumpieren. Ein weiterer Vorteil des Prefetchers ist die effiziente Verhinderung jedes möglichen Cache-Miss der während der Laufzeit des Codes auftreten kann.

Um das Systemdesign zusammensetzbar und kompositionell zu gestalten wird eine neue Speicherhierarchie vorgeschlagen, die stabiles Zeitverhalten während der Codeausführung liefert. Obwohl Single-Path-Code immer dieselbe Anweisungssequenz ausführen, kann die Laufzeit aufgrund der Abhängigkeit vom Zustand der Speicher schwanken. Um eine stabile Ausführungszeit zu erhalten, benötigt man die Wiederholbarkeit in der Folge der Hardwarezustände auf jeder Schicht der Speicherhierarchie. Deshalb wird in dieser Arbeit auch eine neue Speicherhierarchie definiert, die erzwingt, dass die Sequenzen der Hardwarezustände in der Speicherhierarchie für jede Iteration des Codes wiederholbar sind. Die Speicherhierarchie ist auch so adaptiert, dass sie die parallele Arbeit der Fetching und Prefetching Prozesse ohne gegenseitigen Beeinträchtigung erlaubt, um die beste Performance zu erzielen.

Zur Demonstration der Anwendbarkeit des neuen Konzeptes wird die Architektur eines Systems mit der neuen Speicherhierarchie auf einen FPGA-Board implementiert und experimentelle Evaluierungen ausgeführt. Die Ergebnisse beweisen die erzielbaren Vorteile in der Ausführungsperformance für Single-Path-Code.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

In this chapter we introduce embedded systems and describe how these systems deal with timing when they are employed for safety-critical applications. The chapter starts with an introduction on embedded systems and then continues by reasoning about the need for WCET analysis when time is a critical asset of the system. Next, the chapter enumerates the issues that emerge when WCET bounds of the tasks need to be estimated and how these issues can be overcome with the use of single-path approach. In this chapter, we also discuss the main motivation of the thesis for building a time-predictable memory hierarchy which provides predictability and performance improvement for a system that runs single-path code. The list of contributions that has been achieved during this work is discussed as well. At the end, the chapter gives a brief description on the structure of the thesis.

1.1 Introduction to Hard Real-Time Systems

By definition, any self-contained information processing system that is embedded into an enclosing product is called *embedded system* [74]. The price, size and efficiency that embedded systems have achieved made them the widest spread class of computers, with a range of use from home appliances and multimedia systems to nuclear plants and space mission systems [50].

Embedded systems are composed of a processing core, memory component and interfaces used for communication with the external environment. Usually, embedded systems are hidden from the end user and accessed only through interfaces which are acting as remote controller. The software that runs on embedded systems is called *embedded software*. Compared to general-purpose systems, embedded systems differ in many aspects. General purpose systems are designed to run a wide variety of applications where each one has different performance requirements from the rest, while embedded systems are running a single application or a set of applications related to a single function during their whole lifetime. Such an advantage of knowing in advance the purpose of the system gives embedded system designers the opportunity to build systems that are substantially optimized. Depending on the usage, the optimization

can be on power consumption, cost, reliability, etc. Systems of larger scale can even integrate a few embedded systems within, where each one has a particular function. In some cases they can be interconnected in order to interact with each other, but this is not always required. For instance, the modern car is a case of a system that integrates several embedded systems where each one has a particular task, like one for monitoring and controlling the airbags, one for anti-lock brakes, another one for fuel injections and so forth [62].

In several application domains, embedded systems are required to perform in timely manner. In such a system the correctness of the outputs depends not only on the logical results of the computation, but also on the physical instant at which the results are produced [58]. These systems are called *real-time systems*. The time instants when the results must be produced are called *deadlines*. However, whether the deadlines are met or not is not always crucial. There are embedded systems where the miss of the deadline will affect only the system performance but not the functionality of the system itself. These types of systems are known as *soft real-time systems*. For example, video streaming is a real-time system with soft deadlines, because missing of a few frames will only reduce the quality of service, but the service is still useful. Conversely, there are systems where the miss of a single deadline can lead the whole system to catastrophic failure. These systems are called *hard real-time systems*. An example of a system with hard timing deadlines is the fly-by-wire controller which is part of the aircraft flying system that moves the aircraft surfaces for achieving the desired flight path [13]. The fly-by-wire controller consists of a set of tasks that are running in a predetermined sequential order. Their purpose is to periodically scan the inputs from the pilot and the surface sensors, to calculate the actual position of the surface parts and, based on these estimations, to release new outputs to the actuators for their new position. Considering that the fly-by-wire controller has a crucial safety function in preventing the aircraft from going outside of its safe operating range, it is important that in such a system each task must respond within a certain timing period that is given in the aircraft specification. Failure to do so can cause the aircraft to go off-course at the best or crash at the worst. Guarantees for task deadlines can be given only if *timing analysis* of the tasks is performed. However, when a task is executed, its execution time can take various values across the range of times due to dependencies on input data and the initial hardware state. For hard real-time systems, validation process has to consider the maximum length of the execution time of that task which is known as *worst case execution time* (WCET), while the analysis that estimates the WCET value is called *WCET analysis* [120]. The WCET analysis of the task is performed under the assumption that the task is not interrupted during its execution. In this thesis we will consider only the group of real-time systems whose deadline is hard.

1.2 Motivation

Performing WCET analysis on safety-critical embedded systems is a challenging issue, especially for systems that use caches, pipelines, control speculation and out-of-order execution. The chase for better performance makes the presence of these features almost inevitable for the used computer systems, caches aim to bridge the speed gap between processor and main memory, the pipeline has the goal of overlapping the execution of instructions, the branch predictor speculates on the outcome of branch instructions to keep the pipeline full with correct instruc-

tions and out-of-order execution minimizes the stall of the pipeline [102]. On the other hand, these features are the main source of complexity when the WCET of the system needs to be assessed. The complexity emerges due to execution-history dependency that these features impose on the execution time of the instructions [49, 98]. For instance, the cache can vary the execution time of an instruction from one to hundred of clock cycles, depending if the cache access for that instruction results in a hit or miss [119]. Additionally, these features are also interdependent, which means that the analysis gets even more complex when a timing effect that occurs between these features as a result of interference has to be considered as well [47]. For example, a wrong speculation on a branch outcome will entitle the processor to fetch wrong instructions which also affects the state of the cache. Another example is the interdependency between cache and pipeline, where a cache miss stalls the pipeline and affects the timing of the other ongoing pipelined instructions. Thus, for a software that runs on such architectures, a high-quality WCET estimation can be performed only if the analysis covers all the possible system states that can emerge during the runtime. Such an approach of exploring all the possible hardware states of the system will lead the WCET analysis quickly into an unmanageable state-space explosion.

The state-of-the-art WCET tools avoid the problem of complexity by substituting the real hardware model with an abstracted one [120]. Abstract interpretation is a static program analysis method that executes an abstract version of the program on a set of abstract values [24]. In case of WCET analysis, abstract domain and abstract transition functions are defined to reduce the complexity of hardware modeling and with that to reduce the set of states that need to be analyzed. The abstraction is considered successful if the set of abstract states can represent compactly the real hardware states at any program point. However, abstraction leads to information loss, which in turn results in lots of unclassified model states and with that in a pessimistic computation of the WCET value. This affects the task CPU-time reservation, since pessimistic estimates will lead to poor processor utilization and overly pessimistic results on schedulability tests.

A technique to eliminate the complexity problem of WCET analysis is the use of *single-path conversion* [88]. The approach transforms the conventional code into a code with single execution trace called *single-path code*. This is achieved by serializing all input-dependent alternatives of the code into code with sequential segments. The approach also converts loops with input-data dependent termination conditions into loops with constant iteration counts. Thus, the newly generated code, whose execution is fully independent from the input data, forces the execution to always follow the same sequence of instructions for any set of inputs. The WCET analysis process for such a code becomes trivial since the whole procedure is turned into a simple single execution time measurement. Another advantage of single-path conversion is the property of composability and scalability that the software gains with the use of single-path code, which also simplifies the process of system design [89]. However, the single-path code has one major drawback because the conversion may generate code which can end up with a quite long execution time, especially if the original code has many input-data dependent control decisions.

The main motivation of this thesis is to present a solution that improves the performance of the system running single-path code. It is well known that over the past decades the speed of processors has been rapidly increased, while the development of main memory was more focused

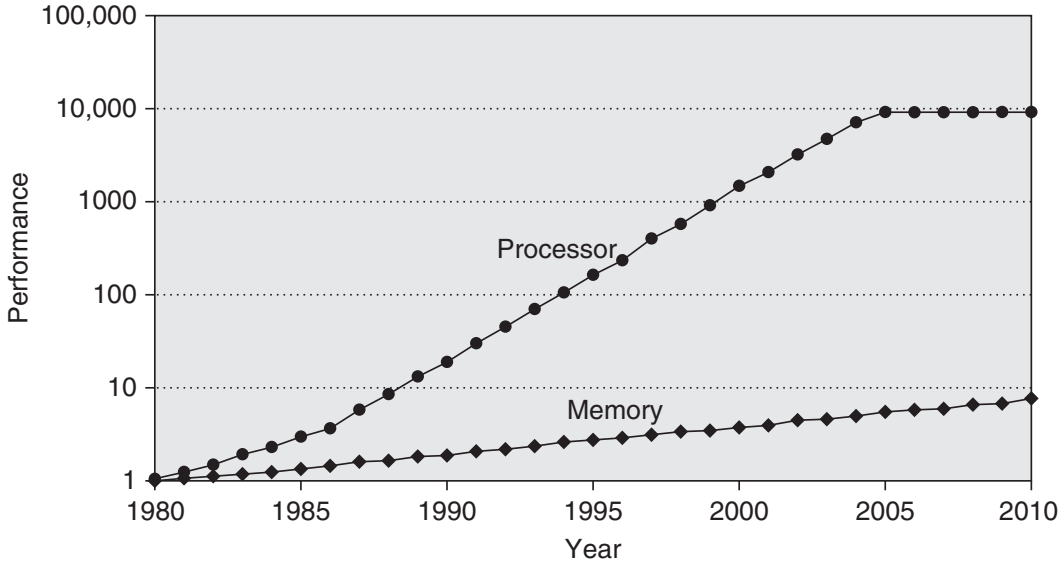


Figure 1.1: Performance gap between processor and DRAM memory over the years [50]

on increments of the storage density. The speed gap between these two units (Figure 1.1) became one of the major factors in limiting system performance [122]. Although the employment of the cache is crucial in bridging this gap, its presence as part of the memory hierarchy cannot be considered as a complete solution yet. Caches improve the memory performance by holding a copy of code fractions near to the processor. Thus, whenever a fetch request results into cache hit, it will be serviced immediately, without any delay. However, this solution reduces only the frequency of main memory accesses, but not the time that is required for main memory access. If a cache miss occurs, the processor still has to be stalled and wait for the missed instruction to be brought from the main memory into the cache [50]. In such cases, prefetching has been shown to be an effective solution. Its objective is to mask the large latency of memory accesses by bringing memory blocks into caches before they are referenced [104]. The scheme can be implemented as a software or hardware solution and the prefetched blocks can contains instruction or data. However, employment of a prefetcher into the memory hierarchy does not mean success by default. Prefetching requires prediction of the future stream of execution, which is not trivial in most cases. Thus, the presence of a prefetcher as part of the hierarchy can interfere with the normal cache operation by keeping the ports of the cache, the memory bus and the main memory itself busy with useless traffic [111]. In such cases, the prefetcher will do more harm on performance than providing improvements. A full utilization of prefetcher benefits can be achieved only if the prefetch requests are issued at the right moment and have the correct target address. Any deviation from these two parameters will not only underuse the capacity of the prefetcher but also reduce the system performance.

Furthermore, a system that runs single-path code should be composable and compositional by providing stable execution times. To achieve that, the memory hierarchy must force repeatability through all the layers of the hierarchy, which means that the sequence of memory states

through the execution of the code should be the same for any iteration of the code. However, this is not trivial to achieve with conventional memory hierarchy solutions. First, not every cache organization guarantees that the sequence of cache states will be repeatable through different iterations, even though the sequence of instructions is the same. Second, the interference that occurs between instruction and data path when main memory is accessed can change the timing of instructions in an unpredictable way. Third, the asynchronism between DRAM refresh and memory access can impose unpredictable timing variation on instruction level. All these occurrences are affecting the execution time of instructions by imposing jitter on the execution time.

1.3 Contribution

The major contribution of this work is a new memory hierarchy that makes systems with single-path code become useful and competitive for applications with real-time properties. To achieve that, the memory has to provide performance improvements by reducing the execution timing of single-path code as well as predictability on temporal behavior through all layers of the hierarchy. In this thesis, we have build and evaluated a memory design that exploits the pre-runtime knowledge about the execution traces of single-path code to prefetch instructions into the cache before they are required. Such properties enable the prefetcher to behave predictably and work efficiently through the whole runtime of the single-path code without polluting the cache at any moment. To reach the best performance on the whole memory, other levels of the memory needs to be adapted as well. Therefore, we propose a modified cache design that allows regular instruction fetching and prefetching to work in parallel without interference. Moreover, we also have proposed a new organization of the hierarchy in order to eliminate execution jitter and provide stable timing. Such an approach enables the system designer to build a time-predictable system that has better performance and still preserve the properties of simplicity and scalability gained for the single-path code. The contribution in this work is mainly focused on the instruction path of the memory hierarchy with small changes in the data path.

In the following we emphasize the major contributions accomplished with the implementation of the new memory hierarchy:

- **Accuracy** - The calculation of every prefetch target is always accurate, which also eliminates the possibility for cache pollution and useless memory traffic.
- **Redundancy** - All issued prefetch requests are sent to cache and not directly to the main memory in order to avoid redundancy to prefetch instructions that are already in the cache.
- **Miss penalty time reduction** - Prefetching is performed in parallel with execution by utilizing the free bus cycles in order to reduce the timing of cache-miss penalties.
- **Miss rate reduction** - In cases when loops fit in the cache, the prefetcher continues with filling the cache with more than one cache line without interfering with the loop content. This eliminates the cache misses of the instructions following after the loop exit and reduces the cache miss rate.

- **Cache conflict avoidance** - The prefetcher preserves temporal locality of the cache by avoiding to prefetch instructions that are mapped in the actual cache lines accessed by the fetch stage. This is active only when the cache is organized as direct-mapped cache.
- **Optimized reference table** - The use of an indexed table for pointing to the next prefetch target eliminates the need to have a fully associative table. This simplifies the hardware and reduces the table access time.
- **Independent from other components** - Having a hardware solution makes the prefetcher fully autonomous, without requiring any changes on the CPU itself or the compiler.
- **Repeatability** - All layers of the memory hierarchy are organized to provide repeatability of instruction timing through any iteration of the code.
- **Stability** - The execution of the single-path code has the same time for any iteration which means it is jitter free.

1.4 Structure of the Thesis

The thesis has the following structure:

Chapter 2 - compares the advantages and disadvantages of the single-path code approach with techniques that are nowadays used for WCET tools. The chapter introduces the state-of-the-art techniques for WCET estimation, describes the issues that occur due to complexity when these techniques are employed and then continues with a description of single-path transformation rules and shows how this approach overcomes the problems of complexity.

Chapter 3 - gives a background on all levels of memory hierarchy by showing memory elements of each level of the hierarchy, how they can be organized and what type of technology is used for their implementation. The chapter starts by describing the concept of hierarchical memory and then explains in detail each component starting from cache, scrachpad and main memory. This chapter also describes locking and prefetching as cache techniques for performance improvement.

Chapter 4 - presents the time-predictable instruction prefetcher, by enumerating the requirements that a prefetcher should have to be efficient and time-predictable and how these properties are accomplished in the new single-path code prefetcher. The presentation of the time-predictable prefetcher starts with a description of the algorithm and then continues with its architecture. A part of this chapter is also the generation and organization of the table that guides the prefetcher through the execution.

Chapter 5 - goes through all layers of the memory hierarchy and shows the configuration that each component needs to have in order to achieve repeatability of instruction timing for any iteration of the code and with that stability on code execution timing. This chapter shows also the modifications that are required to be implemented in the cache in order to enable the fetching and prefetching process to be overlapped without interfering each other.

Chapter 6 - presents hardware implementation algorithms for prefetcher and cache memory, as well as their integration in the Patmos processor. The chapter describes the evaluation process

and the benchmarks that are used for that purpose. At the end, the results of the evaluation are presented.

Chapter 7 - presents the related work. It starts by describing the state-of-the-art approach for cache analysis and then continues with techniques used to improve predictability of the on-chip memories. Next, time-predictable on-chip memory with prefetcher are also described.

Chapter 8 - summarizes the thesis with a conclusion and potential future work that can be done in this direction.

CHAPTER 2

Worst Case Execution Time Analysis vs. Single-path Conversion

In this chapter, we compare the process of estimating the worst-case execution time (WCET) bound on conventional code and on single-path code. The chapter starts with a brief description of the state-of-the-art strategies that are used nowadays in timing analysis, followed by a list of issues and obstacles that these techniques have when they are employed in systems with modern hardware. The chapter continues with a description of the single-path strategy and how this approach overpasses all these problems. At the end, a summary is given on the advantages and benefits brought by the code with a single execution path when employed in real-time systems.

2.1 Worst-Case Execution Time Analysis

Any structured method or tool that obtains information about the execution time of a code or part of it can be considered for *execution-time analysis* [62]. In general, performing execution-time analysis of a piece code is not possible due to the halting problem. To make the problem tractable, the program is restricted with code that is free of recursive structures and has loops whose number of iterations is boundable in order to guarantee that the program always terminate. [120]. Despite these restrictions, the execution time analysis process is not trivial. Dependency on input data and initial hardware states force the execution to vary through a range of values. The smallest value within that range is called *best-case execution time* (BCET), the largest one *worst-case execution time* (WCET). For a hard real-time system the WCET is the only subject of interest when timing guarantees have to be provided. However, to estimate a precise WCET value, the analysis needs to go through all the possible execution states, which in practice is almost infeasible due to the amount of memory and the computation time that is required for such an approach.

The state-of-the-art WCET analysis tools are designed to overcome this problem by deriving an estimated $WCET_{est}$ value, which in fact represents a conservative approximation of the real

$WCET_{real}$. In order to be valid, the estimated $WCET_{est}$ should be *safe* and *tight* [70]. A WCET bound is considered safe if the estimated value is greater or equal to the real bound $WCET_{est} \geqslant WCET_{real}$. Overestimation guarantees that the $WCET_{est}$ has covered all the possible WCET cases. For the second requirement, the $WCET_{est}$ bound is considered tight if the margin of uncertainty ε of the estimated value is within an acceptable range $WCET_{est} \in [WCET_{real} - \varepsilon, WCET_{real} + \varepsilon]$. For instance, if the $WCET_{est}$ bound is highly overestimated, it will lead to a system with underutilized resources. Tools that are used nowadays for WCET analysis are mainly divided into three major groups: *measurement-based*, *static analysis* and *hybrid approaches*.

The measurement-based approach is the most common way used in industry. It derives the WCET bound by executing the task on real hardware or a simulator and measuring end-to-end the execution time of the program. The estimation is derived either by augmenting the program with an additional code that would read the hardware timer at predefined points, or through additional hardware that observes the signals of the relevant pins [109]. Since the code consists of many execution paths, it is required for the approach to perform many measurements of the same code but with different input vectors in order to trigger different paths. The outcome from all measurements is compared and the one with the longest execution time is selected for $WCET_{est}$. The set of input vectors that are used to trigger different paths can be generated randomly or with the help of an evolutionary algorithm [114]. In practice, the initial input vector is generated randomly or as a set of carefully selected inputs that are supposed to trigger a case with quite long execution time and then through the use of algorithms the input vector evolves to trigger a path with long execution time. The process is repeated until the maximal predefined number of iterations is reached or all available resources are consumed. However, finding an input vector that would trigger the $WCET_{real}$ is very difficult. To ensure that such a vector has been found, measurements for all possible combinations of input vectors for that program must be performed, which in practice is infeasible. Thus, the values derived with measurement-based approach are always considered to be underestimated compared with the real WCET. This is one of the main disadvantages of the measurement-based WCET approach. The second problem is instrumentation of the code for measurement purposes. The augmented code is useful only during the validation process, but the validation restriction considers the code for valid only if it remains unchanged. This means that the system in use has to use the same code, including the instrumented parts, although they are not required for further use. The usual practice in industry with measurement-based method is to do few measurements, determine the longest measured execution time from that set and then to add a safety margin to that value. However, such a solution is not always safe since in many cases it may generate still an underestimated bound and in some cases it may convert the bound from underestimated to highly overestimated.

The static analysis is the only solution when strong evidence on WCET are required. Unlike the measurement method, static analysis does not rely on the execution of the code, but combines the information extracted from the code with mathematical model of the system architecture to derive the WCET bound [70]. The whole analysis process is divided into three phases [120]: *control-flow analysis*, *processor-behavior analysis*, and *estimate calculation*.

- **Control-flow analysis** collects information on possible execution paths of the code. Real-time programs by construction have to guarantee termination of the program for any input

data. This means that this type of programs have always a finite number of possible execution paths. However, finding the exact set of paths is in general undecidable. Therefore, control-flow analysis defines a superset of paths which is considered as a safe approximation of the real set. On the other hand, the superset should be as narrow as possible in order to have small overestimations. The analysis firstly builds the *control-flow graph*, which reflects the program structure. Next, it derives information on the bound of the loops, functions that are called, dependencies between conditions and so on. The bound on maximal number of iterations that a loop can have is determined through *loop bound analysis* [34, 48]. To narrow the set of possible paths, and with that to reduce the pessimism of the estimated WCET bound, the analysis tries to identify and exclude *infeasible paths*. As infeasible are considered those paths which according to the control-flow graph structure are executable, but not feasible when the semantics of the code and the set of possible input data are taken into account [34]. To increase the accuracy of loop bounds and infeasible paths, the analysis also performs *value analysis*, which determines the range of possible values that the registers can have at any point in the program. All this flow information can be extracted from the source level of the code or from its executable binary. Extracting information from the source code is easier but not always practical since not all changes that happen during the compile time can be mapped into the graph. On the other hand, extracting information from executable binary is more difficult but the outcome includes all optimization changes which are performed through compiling. Furthermore, flow information can be provided manually by the programmer or through automatic flow analysis. Manually extracted flow information can be error-prone, while the use of automatic approach is not always possible.

- **Processor-behavior analysis** determines execution time bounds of instructions or basic blocks on a given architecture. For systems with simple hardware, this is an easy and straightforward process since the execution time of each instruction is static. A problem emerges when the program executes on a processor that has cache, pipeline and branch predictor, where the effects generated from these features impose variability on the execution time of instructions. In such a system, the execution time of a single instruction is dependent on the state of the hardware, while the information about the hardware state at that moment is itself strongly dependent on the history of the previous hardware states. Therefore, for precise timing estimation, the processor-behavior analysis has to consider all the possible hardware states that lead the execution to that instruction [119]. To avoid the problem of state space explosion, most of the hardware analyses employ timing models that are based on abstract interpretation [62]. The benefit of abstraction is that it simplifies the analysis by substituting the real hardware model with a simplified abstract model, which reduces the number of states that need to be considered and the calculation efforts. However, the abstract model can run into a situation where estimation on timing for a particular state cannot be derived. In this case the model uses conservative approximation under the assumption that such an approach is safe to be used. For instance, if the analysis cannot determine if the cache reference of a particular instruction will result into a hit or a miss, then a safe assumption is considered to classify that instruction as miss in any cache context. In contrast to control-flow analysis, which can be performed in source or binary

code, the processor-behavior analysis requires access to binary code, since it analyses the time on instruction level.

- **Estimate calculation** is the last stage of static analysis which estimates the WCET upper bound of the whole task, by combining the program flow information derived from control-flow analysis with the outcome from processor-behavior analysis. There are three possible methods to perform the calculation phase: *tree-based calculation*, *path-based calculation* and *implicit path-enumeration technique* [120]. Tree-based (structure-based) estimation generates the WCET bound by performing bottom-up traversal through the syntax tree of the program [9, 22]. As it traverses through the tree, the analysis merges the nodes to a single one and at the same time it also estimates the time for that node. All transformation are done in accordance with the rules for tree-based transformations. However, the problem of this approach is that not every control flow can be easily expressed within a syntax tree. The second technique, path-based calculation, estimates the upper bound by representing the possible execution paths of the task explicitly and then search through that set for paths to find the one with longest execution time, which is also considered as the WCET of that task [108]. The approach is straightforward for codes with single loops, but becomes complex when loops are nested. The complexity is also related exponentially with the number of paths that need to be examined, which makes the approach not so suitable for codes that have a large number of paths. The last estimate calculation technique, called IPET, transforms the WCET estimation into an integer linear programing problem where the structure of the program and the execution-flow information are represented in form of constraints [67, 92]. The WCET bound of the task is obtained by maximizing the objective function $WCET = \sum_{n=1}^{N} B_i \times C_i$, where B_i is the longest execution time that the basic block i can have and C_i is the execution frequency of that block. Unlike the previous two methods, IPET is able to handle different types of flow information. The only problem with IPET is that flow information are represented in form of constraints, while the size of complexity of estimation grows with the number of constraints. Even though ILP is not the most suitable technique to estimate the WCET bound it still remains as the most used approach because of its power compared to tree-based and path-based methods.

Hybrid approaches combine measurement with static complementing each other. The goal is to substitute the complex low-level analysis used in static analysis with a simple measurement-based solution. The approach uses firstly the static analysis to construct the model of the program and next it partitions the code into small segments. In most cases the size of the partitioned segments is the size of the basic blocks. When the set of all possible segments is defined, the tool performs measurements by executing the segments on real hardware or in a cycle-accurate simulator. If the segment contains data-input branches, then the input data for complete coverage of that segment should also be given. The measured values are then used as input to static analysis to produce the WCET bound. Although the hybrid approach includes all the possible paths, it still cannot be considered as a solution that derives a safe WCET bound since determining the worst-case initial state for each segment is difficult or even impossible in some cases [120]. The

context problem is usually attacked by running more measurements of the same segment with different initial states in order to decrease the uncertainty.

To summarize, all three methods have their pros and cons. Static analysis has in favor the fact that it covers all the possible context dependencies and gives guarantees, and that the estimated WCET bound is always safe. In favor of static analysis is also the use of abstraction, which simplifies complex hardware models and eliminates the need for real code execution. However, abstraction causes loss of information, which is also reflected in the diminished WCET bound accuracy. So far, static analysis still misses a general abstraction model that could be employed on any hardware configuration. Abstracted models, which are used nowadays, are valid only for small set of hardware configurations. Even for these configurations, if the hardware experiences a small change it makes the analysis not valid anymore. Adapting the model for the new hardware configuration is also a difficult process. In contrast to static analysis, the measurement-based approach is much simpler to apply since no timing model for the hardware is needed. However, the main disadvantage of this approach is the scarcity of input vectors that would guarantee WCET bounds. Even if somehow the input vector that triggers the real WCET is found, this does not mean that the same input vector will trigger the WCET of the same program on other hardware platforms [7]. Hybrid analysis on one side tries to simplify the complexity of hardware by replacing the abstract processor analysis with measurements, but on the other side gives less precise WCET estimation by integrating the measured values into the static analysis. The main advantage of this approach is that it requires less effort when a new hardware is used. It still cannot give guarantees on the estimated bound.

2.2 WCET Analysis Issues on Modern Hardware

In theory, the estimation of WCET bounds is a decidable problem, because the software and hardware that are used for real-time systems are by design restricted to have a countable state space. However, in practice this is not the case. The complexity imposed from the software and the hardware of the modern system expands the state space to the scale that is tedious or in some cases even infeasible to be analyzed. The WCET research community tries to diminish this problem by proposing timing models that would derive approximated WCET bounds. The efficiency of these models is strongly related to the accuracy of the estimated WCET bound and the computation effort that is needed to calculate these bounds [56]. Hence, to be acceptable the WCET analysis should provide timing models and analysis methods that are affordable, tractable and derive precise results [121]. Despite the advances that have been made during the last decades in improving the efficiency and the correctness of the models for WCET analysis, there are still issues that the analysis has to deal with. In the following we describe some of these issues that the WCET analysis has to deal with.

On the software level, the analysis faces the issue of complexity when information on dynamic behavior of the code needs to be extracted. Although there are a number of techniques that can derive loop bounds [35, 48, 57] and infeasible paths [46, 110] automatically, they are not always successful. When they fail, flow information needs to be provided manually by the programmer, which is a tedious and error-prone process [55].

The situation gets more aggravated with the low-level analysis. The demand for powerful processing forces the embedded systems to adopt techniques that were developed for general purpose systems like cache, pipeline, branch predictor and out-of-order execution [61]. On the other hand, these features are the main source of complexity when the analysis has to be done and not all of them have adequate timing models that can be used for estimation of the WCET bound [98, 121]. In the following, we describe the current problems that the static analysis has to deal with when features like these are employed on the system.

Cache analysis has the goal of classifying each memory access as a hit or miss. Running exhaustive exploration through all the possible states that the cache can have during program execution is not feasible due to the enormous size that the cache state space can have. Abstract interpretation is considered a well established approach that simplifies the complexity of the cache analysis by aggregating states of different paths into a single abstracted state. However, the precision of the results derived from the abstracted models are strongly dependent on the architecture of the cache and its replacement policy [49]. For example, the abstracted model for LRU replacement policy achieves better predictability compared to the FIFO or PLRU policy, since the updates of the abstract states for caches with LRU behave in more regular fashion than those with non-LRU [73]. Cache with non-LRU policies are also more sensitive to the initial states, which means that the abstract model requires a longer sequence of memory accesses to evict all unknown cache states from analysis [94]. The uncertainty of cache analysis gets even higher when subject of analysis are data caches. Before classifying the cache access, cache analysis needs to know in advance the reference address of that memory access. Unlike instructions where each address is often available during the compile stage, the addresses for dynamic data structures can only be known at the run-time. Memory access with unknown address destroy the history trace of the cache states up to that moment and with that also the predictability of the analysis [100]. For some cases, value analysis can be used to reduce the set of possible target addresses which are part of indirect addressing [73]. The same problem remains for unified caches, since the uncertainty on data accesses will destroy the predictability not only for data accesses but also for instructions. All of this shows that the current abstract model used for cache analysis cannot be considered a mature solution yet.

Pipeline analysis models the behavior of a program through the processors pipeline. For an ideal pipeline, modeling the transition of instructions would be easy, since the execution time latency of instructions would be equal with the length of the pipeline stage. Unfortunately, with real pipelines the execution suffers from data, control, or structural hazard due to instruction dependencies on data or shared resources [50]. When such hazards occur, the pipeline is stalled until the dependency conflict is solved. However, even if most of the pipelined processors have forwarding techniques implemented, they still are not able to fully eliminate the stall penalty time. Interdependency between pipelined instructions prevents the pipeline analysis to perform individual analysis at the level of single instructions. Instead, the analysis considers all current pipelined instructions collectively in order to identify all potential pipeline stalls that can occur and include them into the timing model. Dependencies within a basic block can be modeled easily. The problem emerges when an instruction of one basic block causes delays to instructions of the other basic blocks which are not immediate successors [33]. Such dependencies are called *long timing effects*. They are the main reason for preventing the pipeline analysis to be performed

on the level of isolated basic blocks.

Branch prediction analysis has the purpose to analyze the behavior of the branch predictor. The easiest way to deal with this feature is to disable it or to assume that all branches are mispredicted. However, such an assumption would result in a highly pessimistic bound considering that mispredicted branches can be very expensive for architectures with deep pipelines. Another reason is that the rate of prediction of modern dynamic branch predictors is quite high. Therefore, inclusion of branch-prediction analysis is important when tight WCET bounds are required. The branch-prediction analysis integrates estimated information about branch misses into the global WCET analysis by adding penalty cost for mispredicted branches [23] or by bounding the number of mispredicted branches through ILP constraints as part of IPET approach [66]. The first solution uses tree-based computation and has implementation only for local branch predictor, while the second one is implemented for global dynamic branch predictor and is more precise since it considers not only the number of miss-predicted branches but also the effect of the branch predictor on the cache states. Although the last approach achieves a high precision, its employment can increase the complexity very fast to a level that makes the analysis to become not affordable anymore [14].

Powerful out-of-order processors are suffering from *timing anomalies*. This phenomenon invalidates the intuitive assumption that a local pessimistic assumption will lead to the global WCET. Anomalies can emerge when the processor speculates with execution on wrong direction (*speculation-caused anomalies*), or when a sequence of instructions are scheduled differently on the hardware resources (*scheduling anomalies*) [72]. Speculative execution is used to avoid pipeline stalls caused from branch instructions, but if the speculation is incorrect the processor will prefetch wrong instructions, which will also change the cache content. Changes done from wrong speculation sometimes can cost more to undo than the cost of a cache miss. Hence, the local assumption of a cache miss can lead to a globally shorter time. The second type of anomaly can occur in multi-issue out-of-order processor when a cache hit can take longer than cache miss due to a dependency of instructions and the way how these instructions are allocated to the resources. In addition, it has been demonstrated that timing anomalies can also occur in multi-issue in-order processors [118].

2.3 Single-path Approach

The single-path approach uses a code transformation strategy that converts conventional code with multiple-execution paths to a code that has a single execution trace [88]. The idea behind the strategy is the elimination of all input-data dependences of the code by serializing input-dependent alternatives into sequential code segments. The outcome from this transformation is a new generated code that has preserved the same semantic as the original code, but its execution always follows the same sequence of instructions regardless of the program input values. For a code with such a behavior, the WCET analysis becomes a simple process. To determine the WCET, it is sufficient to run the code and measure the execution time for only one execution.

The whole concept of single-path transformation is based on so called *if-conversion* which converts control dependences of the code to data dependences [4]. This technique was firstly applied only on the body of the most inner loops in order to convert them into non-branching

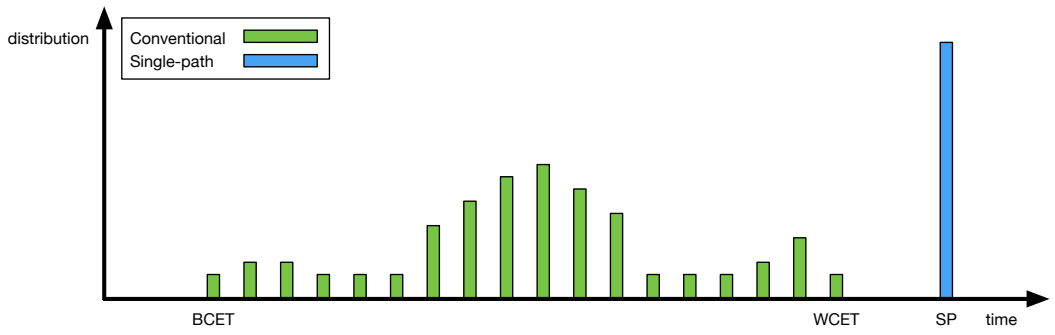


Figure 2.1: Distribution of the execution time for conventional and converted single-path code

code and with that to avoid the stall of the pipeline, especially for processors with deep pipelines. The single-path approach extends this strategy through the whole code by covering all input-dependent control flows, loops and functions of the code. The strategy can be applied on any conventional code whose WCET can be bounded.

To achieve input-data independent execution, the transformation needs to transform all input dependent control flow instructions. Identification of those instructions can be done automatically by data-flow analysis. On the other hand, control flow branches that are input data independent can be preserved as they are or they can also be subject to if-conversion. In both cases the code will be single-path and will have the same semantic. The difference is only in the amount of computation time that is required for their execution. If input independent branches are preserved, then the code will contain mutual alternatives whose patterns of execution will be always the same regardless of the input data. If they are converted it means that the alternatives will be serialized and with that also the execution stream of instructions.

Figure 2.1 shows the distribution of the execution time for conventional and single-path code. While a conventional code has different times for different input data, the single-path converted code of the same program has the same execution time for any input value. However, the benefits to have a jitter-free code comes at the cost of the execution time. As can be seen from the figure, the execution time of single-path code can take much longer than the WCET of conventional one due to its behavior to force the execution through all instructions of the code.

In the following we describe predicated instruction as a requirement that the system should support in order to perform single-path execution, the set of rules that are applied on conventional code for its conversion into a single-path one and the beneficial timing properties that are gained from the conversion.

Hardware Requirements for Execution of Single-path Code

The execution of single-path code requires from the hardware to support *predicated instruction* in order to preserve the semantic of the original code. Predicated instructions are instructions whose semantics are controlled by a predicate (guard), which can be implemented through a specific predicate flag or a register in the processor [79]. Instructions whose predicate value is evaluated as “true” are executed normally, while those with “false” predicates are nullified

in order to prevent the processor from modifying its state. Predicates of instructions should be calculated before entering the code segment and the value of the predicate should be kept unchanged through the whole segment. Since the single-path conversion is spread through the entire code, the compiler has to take into account interdependencies of the predicates within nested code structures. In such cases the predicate of the outer structures is passed to the inner one and then using the boolean algebra they are combined with the inner predicates. For instance, if a piece of code has two nested loops where L_1 is the outer and L_2 is the inner loop, and if their exit conditions are associated with predicates p_1 and p_2 then the body in the inner loop is also associated with the outcome of $p_1 \wedge p_2$.

Single-path Conversion Rules

Single-path conversion is performed on the executable binary after optimization transformations have been conducted by compiler [84]. However, to be more lucid with the concept of the single-path transformation, we demonstrate the transformation rules as they would have been performed in the source-level of the code. The rules are applied on those parts of the code which generate variability in execution time. Code structures that are transformed are input-dependent conditional branches, loops and function calls [90].

Input-dependent branches of *if* or *case* structures are conditional statements whose semantic consists of two or more mutually exclusive alternatives where only one of them can be executed. The decision about which particular alternative will be executed depends on the outcome of the branch condition that precedes those alternatives. The single-path conversion serializes all the alternatives of the branch into sequential code and with that forces the execution to pass through the whole branch [87]. The responsibility of predicates is to control which instructions can do register-state changes and which will act as null. In cases when conversion is nested, the predicates of all nested conditions are combined into a conditional assignment. Figure 2.2 illustrates an example on how the if-else conversion eliminates the input data dependent branch instruction. Variant (*a*) is a conventional code with an *if* branch instruction, where only one statement is executed. The decision on which one will be executed depends on the outcome of the condition *cond*. Variant (*b*) executes both statements, but only one of them will change the hardware states and this depends on the value of the predicate *pred*.

```
     if (cond) goto L2
L1:  statement_1
     goto L3
L2:  statement_2
L3:  ...
```

(a) Branch instruction.

```
          pred<-cond
(!pred)   statement_1
( pred)   statement_2
            ...
            ...
```

(b) Predicated instructions.

Figure 2.2: Conversion of branch code into serial code with predicated instructions.

A loop is a code structure that is continually repeated until a certain exit condition has been reached. If the exit condition is input dependent then the loop can exit after any iteration, depend-

ing on the current program input. Such a exit condition generates variability in execution time of the loop for different loop iterations. The single-path approach transforms input-dependent loops into loops with constant iteration count where each iteration has a constant execution time [86]. The conversion sets the iteration count of the new generated loop to the value of the loop bound derived form the original loop, while the exit condition of the original loop is used to calculate the value of the predicate related to the body of the new loop. Figure 2.3 illustrates the transformation of a *while* loop. As can be seen from the figure, the loop is transformed into a *for* loop structure with constant iteration count. The body of the new loop includes predicated instructions. The predicate is responsible in preventing hardware states to be changed after the exit condition from the original loop has been satisfied. The value of the predicate is set at the beginning, before entering the loop, and then is controlled with *statement_cond*. Once this condition is fulfilled, the value of predicate is changed to *false*. The loop will iterate until it reaches *max*. Thus, although the number of loop iterations is constant on each invocation, the semantic of the transformed loop will still be the same as the original one.

```
while (cond) do
  statement_1
  statement_2
  ...
  statement_cond
```

(a) Conventional loop.

```
pred<-cond
for 1 to max
  (pred) statement_1
  (pred) statement_2
         ...
  (pred) pred<-statement_cond
```

(b) Loop with predicated instructions.

Figure 2.3: Conversion of conventional loop into a loop with stable time.

A function is a reusable block of code that is executed whenever it is called. Input-dependent calls invoke the function in dependence to the program input. Single-path conversion transforms the code in a way that each function of the code is called unconditionally, and with that also a predicate related to the function call is passed to the called function. Hence, if the predicate is evaluated as *false*, even though the execution will pass to the function, there will be no changes in the hardware states. The function predicate that is passed to the function represents the initial precondition for all statements within the function.

Predictable Properties of Single-path Code

Single-path converted code, except that it is free of input data-dependent control condition, it also inherits beneficial properties from transformation that help the process of hard real-time system design to be simpler and easier. In the following we enumerate the main properties that the code gains after its transformation [21, 90, 91]:

- *Simplicity* - One of the most important properties of single-path code is its simplicity. By making the execution independent from input data, the single-path conversion also simplifies the process of WCET analysis. Instead of performing complex WCET static

analysis, for single-path code it is sufficient to do a single end-to-end measurement of the execution time of the code and with that to determine the exact WCET value. The measured WCET is safe and valid for any context of the code.

- *Stability* - From a software point of view, single-path code has no variability in execution time for any input data. Such a property eliminates the possibility for overallocation of the hardware resources. The only source for execution-jitter in a system with single-path code can emerge from hardware features due to their temporal behavior that are based on execution history. However, in a well designed system this execution-jitter can also be eliminated.

- *Composability* - Having a single execution trace allows the programmer to add or remove segments from the code without affecting the timing of the other code segments. Hence, each software component can be developed autonomously with respect to the timing and then added to the main code. This makes the single-path converted code timing composable.

- *Compositionality* - The timing of code composites can be derived from the timing of the components with a simple formula. This allows the construction of the system to be performed hierarchically by keeping the design and development of complex composites to be simple. In such an approach the timing of the whole system can be calculated straightforward from its components.

- *Predictability* - Although there is no precise definition on what predictability is, the proposed ones [44,56] classify the single-path code as time-predictable since the timing analysis for such a code calculates the WCET with full accuracy.

All these properties support the design process of a real-time system. The use of hierarchical development allows the designer to dismantle the complexity of the whole system from system level to the level of sub-component and then treat each sub-component as a smaller sub-problem. After building all sub-components, their integration into a whole system is performed. This step is quite simple due to composable and compositional properties that the system components have.

The only drawback for single-path code is that transformation can produce codes with long execution time, since every alternative of the code is executed. This is especially pressing if the code consists of lots of input-data dependent control flows.

2.4 Chapter Summary

Complexity in timing analysis emerges as the result of highly-integrated analyses that state-of-the-art analysis approaches are using. The analyses simultaneously keep track of all hardware features whose performance enhancement are based on the program execution history. Each of these features increases drastically the hardware state space that needs to be analyzed. Additionally, most of these features are also highly interdependent and this effect on timing analysis must be included in WCET analysis as well. Abstraction is proposed as an option to mitigate the

complexity of the analysis by reducing the state space. However, the approach provides acceptable solution only for a small set of features, while for the rest an appropriate model still needs to be found.

With the single-path approach, the process of timing analysis gets simple and easy. The strategy eliminates the complexity of the analysis by generating code that has a single execution path. In a well designed system, the timing of instructions through all hardware features is the same whenever it is executed, since the state transitions of the hardware features are always the same and with that also the path to reach that instruction. Hence, an end-to-end measurement of the execution is sufficient for WCET estimation. Furthermore, the single-path property of stable time gives the opportunity for building composable systems whose timing can be easily derived.

CHAPTER 3

Background on Memory Hierarchy

Since the invention of the microprocessor the density of transistors per chip area has been doubling roughly every year. Whereas microprocessors have used this advantage to increase the rate of executed instructions, memory technology was focused on increasing the capacity. Such divergent development has led to a wide performance gap between processor and memory. To overcome this issue, the concept of *memory hierarchy* has been proposed. A good understanding of the memory hierarchy architecture and the memory technologies is necessary for designing and analyzing real-time systems.

In this chapter we describe the architectural concept of the memory hierarchy, memory hierarchy components and the technologies that are used in nowadays systems to bridge the speed gap between processor and main memory.

3.1 The Concept of Memory Hierarchy

Memory is an addressable storage where programs and data are stored. Since fast memories are expensive and modern programs require large amount of storage, it is important for memory designers to build memory systems that satisfies cost-performance requirements. This can be achieved by combining various storage technologies in a well organized structure that as a whole performs fast and has a low cost per bit. Such memory structures are known as *memory hierarchies*.

Figure 3.1 illustrates, in form of a pyramid, the main components that are used for the construction of a memory hierarchy in embedded systems. Each level of the pyramid provides temporary repository for recently accessed memory blocks, but they are distinct to each other by their size, speed and cost. The components located on the higher layers have better performance than those on the lowers one, but the cost per bit is in the reverse order. At the top are *registers*. These are the smallest and the fastest components of the hierarchy. Usually, registers are organized in a set of so-called *register files*, as an integrated as part of the central processing unit (CPU) in order to keep the actual instructions and data near to the CPU. The second layer

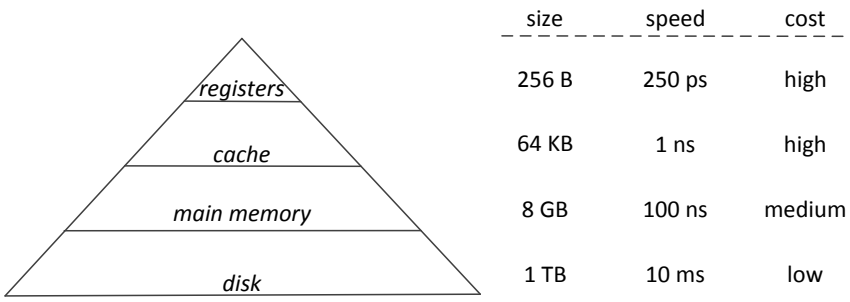


Figure 3.1: Levels of memory hierarchy in embedded system

covers *caches* and *scratchpad* memories for bridging the speed gap between CPU and the main memory. The difference between these two types of memories is that a cache holds content that is duplicated from the lower layers and is transparent to the CPU, while the scratchpads has unique content and occupies part of the memory address space, which means that it can be directly accessed. Another difference is that the cache contents is managed through an autonomous hardware, while scratchpad needs special software for that. The storage of cache and scratchpad components is mainly based on static random access memory (SRAM) technology. The third layer, called *main memory*, is an off-chip memory that is mostly based on dynamic random-access memory DRAM technology. When an embedded system employs non-volatile storage for applications and the main memory is used only for keeping temporal data, then an SRAM chip can be considered for main memory as well. *Disk* is the last and the only layer of the hierarchy that provides non-volatile storage. It has a large capacity, but it is slow on read and write. For embedded systems, this layer is usually used to store executable images and system configuration parameters.

Functionality of the entire concept of the memory hierarchy is based on the *principle of locality*. Locality is an observed code attribute that results from the tendency of computer programs to access the same or nearby memory locations frequently and repeatedly. Mainly, there are two different types of locality:

- *Temporal locality* - means the code executed at the moment is likely to be referenced again in the near future. On the instruction stream, this type of program behavior is exhibited when a loop is executed, since the instructions that are building the loop body are referenced repeatedly on each iteration. On the data stream, temporal locality occurs when the same program variables are accessed frequently.

- *Spatial locality* - refers to the following accesses of nearby memory locations that are close to the actual reference. On the instructions stream, this type of locality occurs when instructions allocated sequentially in the memory are also executed in same order. Within the data reference stream, the spatial locality occurs due to the tendency of compilers to cluster related variables together in the memory space. A classic example for such locality

is the access to elements of arrays, where elements that are belonging to the same array are located adjacently in the memory space.

A memory hierarchy exploits the advantage of locality by keeping the entire executable code in the main memory, while the upper layers of the hierarchy keeps only copies of code fractions. Therefore, the main memory can be considered as the operating storage, cache and registers as the fast storages, while the disk as the permanent storage.

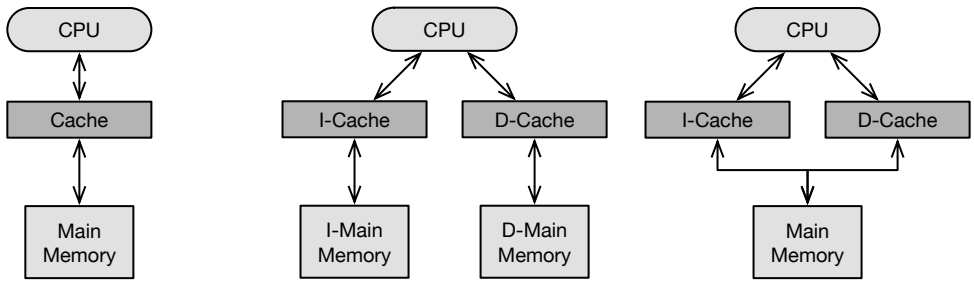
Memory Access Paths

There are two memory-design alternatives for accessing and transferring instructions and data through the layers of the memory hierarchy. These alternatives are known as Von Neumann and Harvard architectures [25].

The Von Neumann architecture, depicted in Figure 3.2a, is a design with a single unified cache that stores both instructions and data on the same storage space. The address space for such a cache can be divided between data and instructions in a way that each type has its own sub-space of addresses, or it can be an intermixed address space that allows both instructions and data to be located anywhere within the cache. On Von Neumann architectures all CPU generated memory accesses are transfered through a single shared bus. Therefore, when the processor is pipelined the transfer of instruction and data should be scheduled, since the fetch and load/store stage can not access the memory at the same time.

In the Harvard architecture (Figure 3.2b), the program memory and data memory are disjoint and each part is accessed through a dedicated bus. The major advantage of such a separated approach is the overlapping of the instruction and data accesses in order to increase the performance of the pipelined processors. In contrast to the Von Neumann architecture, having a separate cache storage for instructions and data does not only reduce the utilization of the cache size but also increase the cost of the system.

Modern CPU combine both memory designs (Figure 3.2c). They use separated on-chip buses for instruction and data cache, while the off-chip communication is done through a common single bus. Such kind of hybrid architecture can be classified internally as Harvard but externally as Von Neumann solution.



(a) Von Neumann architecture (b) Harvard architecture (c) Combined architecture

Figure 3.2: Memory access paths

3.2 Memory technologies

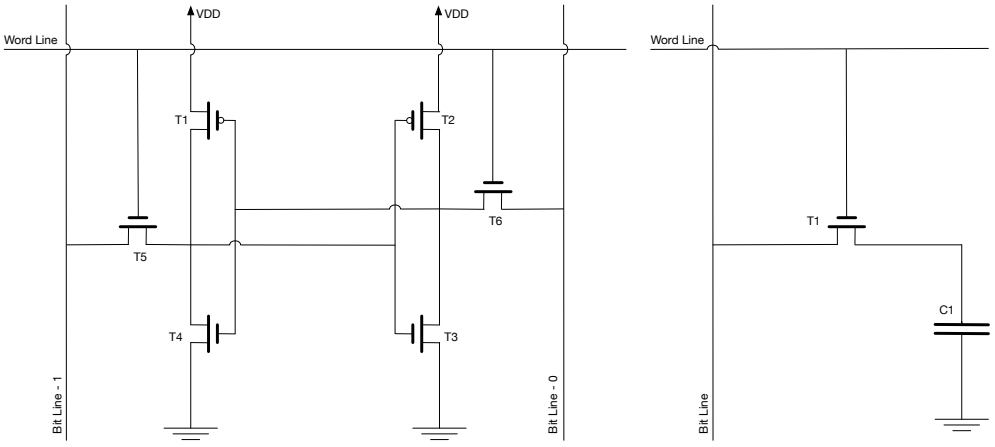
Technologies used for memory components of embedded systems are mainly divided into three main categories [117]:

- SRAM - Static Random Access Memory,
- DRAM - Dynamic Random Access Memory, and
- Flash Memory.

The first two, SRAM and DRAM, are *volatile* types of memories, which means that the content retains on them as long as electrical power is applied on them. Once the power is cut off, all data is lost. In contrast to that, flash memory is *non-volatile* memory, because it keeps the data safe even after power supply is off. In the following we describe the main characteristics of these three types of technologies.

SRAM technology

SRAM is an integrated memory that is organized in form of an array where the cell is its fundamental storage for storing a single bit. Figure 3.3a shows an SRAM cell consisting of six transistors, where transistors T1-T4 are used for keeping the cell value, while T5 and T6 control the access to that cell. The activation of the T5 and T6 transistors is done through the word line, while the value of the cell is read/write through the bit line. Although for normal operation of the SRAM memory one bit line for each cell is enough, in practice the SRAM cells are connected to two bit lines with opposite charging in order to detect faster the voltage difference between these two lines and with that the value of the cell as well [52].



(a) SRAM cell (b) DRAM cell

Figure 3.3: Memory cell technologies

Despite the fact that SRAM memories occupy a relatively large area per bit, they are mainly integrated as part of the processor die since they use the same fabrication process. Such an approach keeps the manufacturing process simple and reduces the cost [117]. Furthermore, this technology provides fast and simple access to the data. The access time to each SRAM cell is fix and can take one or two clock cycles depending on the speed of the CPU. This property makes this technology suitable to be used for cache memories.

DRAM technology

DRAM is an array memory, which unlike SRAM has a much simpler cell structure. Figure 3.3b shows the DRAM cell consists of one transistor-capacitor pair. The charge of the capacitor determines the cell stored value while the transistor controls the access to that bit. The selection of the cell is done by applying voltage to the word line while the cell value is read/write through the bit line. The use of only one transistor makes the DRAM much denser and cheaper per bit than SRAM. However, the DRAM-cell capacitors are not perfect, because they need to be recharged periodically in order to retain the stored information. This is also the reason why this type of memory has the prefix *dynamic* in its name.

DRAM memory is mainly employed as an off-chip memory solution which implies longer latencies and higher power consumption than the on-chip counterparts [102]. For these reasons DRAM is used at the lower level of the memory hierarchy.

Flash Memory

Flash memory is a non-volatile type of memory that stores data permanently but also much slower than SRAM and DRAM. Most embedded system use this type of technology to keep boot loader, operating system or applications that do not change often [10]. Because of their limited performance, the contents of the flash memory is copied into the RAM at the start of the system and is then accessed from there. Writing to the flash memory is possible only in blocks which is firstly preceded by an erasing action. Concerning their internal organization, flash memories can be of type NOR or NAND flash. In contrast to NOR, NAND is cheaper but the reading is much slower since the page has to be re-read if the elements accessed are not continuous, while NOR offers an acceptable random read access due to its parallel internal structure. Furthermore, flash memories of type NOR are suitable for so-called eXecute-In-Place (XIP), which means that the instructions can be read directly from the flash memory without copying the code into the working memory, but this is not always suitable because they are much slower compared to DRAM [117].

3.3 Cache Memory

Caches are small high-speed memories which are located between the processor and main memory. Their purpose is to bridge the speed gap between the processor and main memory by holding copies of code fragments that are frequently accessed. Hence, when the CPU references an address that has been recently accessed, the reference will be fetched from the cache. Instructions are loaded from main memory into the cache with a granularity of one memory block that hold

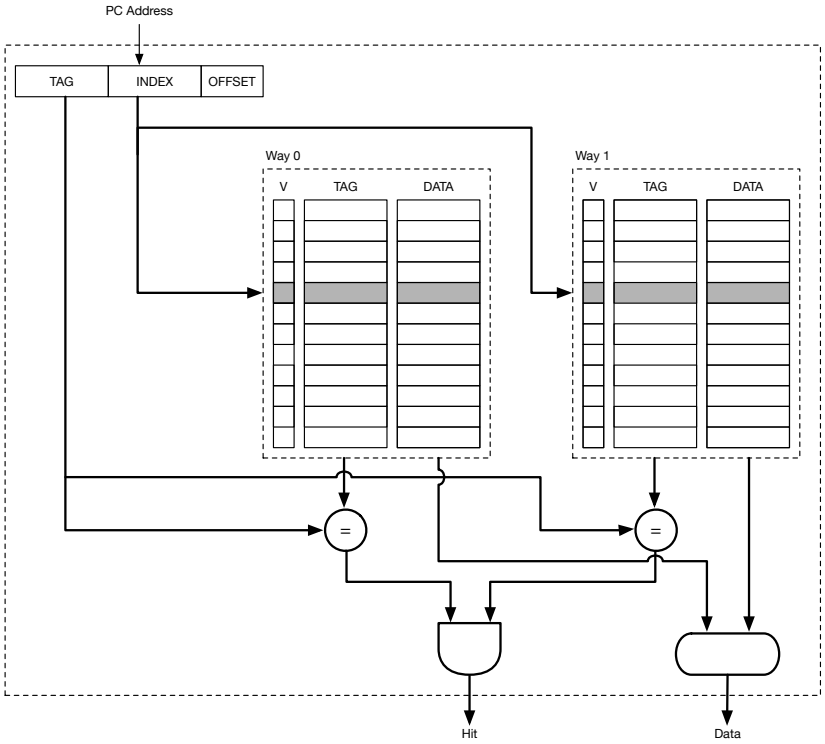


Figure 3.4: Logical organization of a two-way set-associative cache memory

more than one instruction. Such an approach improves the access time when the next accessed instruction belongs to the same cache line.

Cache organization

When the processor requests an instruction, the cache is the first level of the memory hierarchy that encounters the request. Figure 3.4 illustrates an example of a two-way set-associative instruction cache design consisting of lookup tables (tag entries), valid bits (V) and data storages (instruction entries). Since the cache holds only portion of code, its first action is to determine if the requested instruction is in the cache or not. In case of *hit*, the instruction is immediately sent to the processor and the execution continues. Cache hit occurs when *tag* part of the requested address matches with the entry from the cache tag table. The tag table contains the upper portion of the address which is used to identify instructions that are held at that row. A cache can also hold instructions that may not be valid even though the tags match. This can happen when the processor starts up and the content of the cache is not flushed. To indicate whether the cached instructions are valid or not, a *valid bit* of each cache line should be set. If the valid bit is zero then all instructions belonging to that line are considered not valid.

In case of a cache *miss*, the request is forwarded to the next lower level of the memory hierarchy in order to fetch the required block. Such an occurrence affects the processor by stalling its pipeline until the required instruction becomes available. Once the instruction arrives

into the cache, the cache controller will set the valid bit of that line and send the instruction to the fetch stage. The causes for a cache miss can be categorized into three groups [50]:

- *Compulsory* - The very first reference to a block always generates a cache miss, since the block has not yet been loaded into the cache.
- *Capacity* - When the cache is full, some instructions from the cache must be evicted to make space for the incoming fetched instructions. A capacity cache miss occurs when the evicted instructions are referenced again from the processor.
- *Conflict* - If the retrieved block is restricted to be placed only in one location in cache and evicts instructions that are later referenced again then a conflict miss will occur.

The time needed to determine if the requested instruction is in the cache or not is called *hit time*, while the time for retrieving the instructions from the lower memory level (when a cache miss occur) is called *miss penalty time*. Since the performance is the major reason for organizing the memory in form of a hierarchy, the time for servicing hits and misses is important. In the following we describe the impact of the cache-line size, cache size, placement and replacement policies on cache performance.

Cache Line

The main memory is logically partitioned into a set of *cache lines* with size of b bytes. A cache line (cache block) is a fixed-size data that is transfered from main memory to the cache. Selecting the appropriate line size is an important decision of the cache-design process. Small line sizes have several advantages since their transmission time from the main memory is short. With small line size fewer unneeded instructions will be brought to the cache along with the requested one. On the other hand, large line sizes have their advantage in fetching more data at once, increasing the exploitation of spatial locality, and reducing the complexity of the lookup circuit by decreasing the number of lines in the cache. The criteria for choosing the optimal cache-line size are a trade-off between short and long cache lines in order to reduce the number of cache misses and the miss penalty time.

Cache size

Increasing the cache size increases the exploitation of codes temporal locality and with that the cache hit rates. However, the size of the cache cannot be too large for several reasons [104]. Firstly, caches are expensive and usually are on-chip memory, where the part of the die that can be reserved for cache is limited. Secondly, smaller caches have a smaller lookup table, which means less searching hardware and also shorter access times. Hence, the size of the cache is again a trade-off between the price and expected performance.

Cache placement policies

Caches are designed to be transparent to the processor in order to eliminate the control overhead. Such an approach is achieved by employing well-defined placement algorithms that automatically map memory blocks form the main memory to the cache lines.

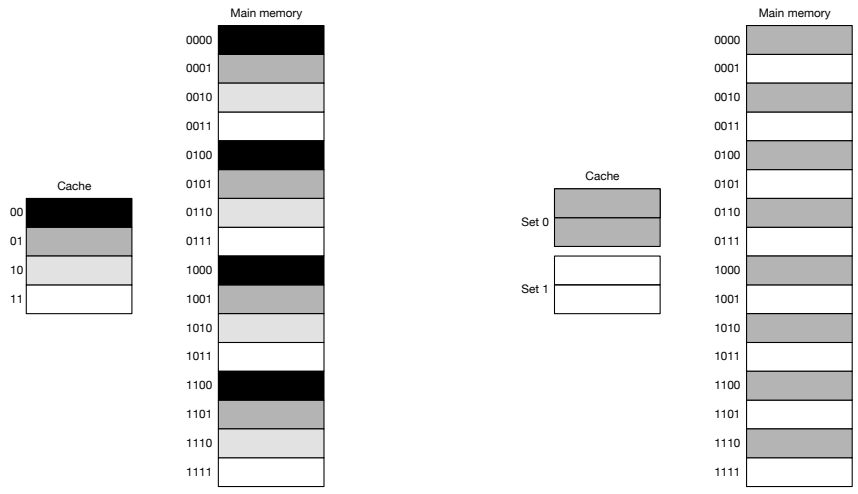
The simplest form of assigning a memory block to the cache is the *direct-mapped* placement policy where each memory block is assigned to a fixed single cache line. The approach forces mapping of type many-to-one, by restricting the placement to only one possible cache location. This location is determined by using n bits from the address of the memory block as a direct index for one of the 2^n possible locations in the cache. Figure 3.5a illustrates the mapping of the memory blocks into the cache using direct-mapping. The equation for calculating the cache-line index is:

$$(\textit{main memory address}) \text{ MOD } (\textit{number of cache lines}) \tag{3.1}$$

The second strategy, called *set-associative*, is more flexible because it allows the memory block to be mapped at any location within a set of cache lines. This policy forces many-to-few mapping. In contrast to direct-mapped, set-associative placement uses part of the address to index the sets. Figure 3.5b illustrates a two-way set-associative cache and how the memory blocks can be located at any of the two cache lines within the assigned set. The mapping between addresses and the sets is done through the following equation:

$$(\textit{main memory address}) \text{ MOD } (\textit{number of sets}) \tag{3.2}$$

The last policy is the most flexible one, since it allows placing any memory block at any location in the cache (any-to-any mapping). This form of placement policy is called *fully associative*. In such an approach no index bits are used from the memory address for determining the storage location.



(a) direct-mapped (b) set-associative

Figure 3.5: Address mapping of direct-mapped and two-way set associative cache

The direct-mapped implementation has the simplest hardware overhead and achieves the shortest access cycle, but due to its restriction on placement the conflict rate between memory blocks placed in the cache is quite high. On the other hand, fully associative is an expensive and slow solution but the conflict rate is zero. In order to balance between these properties different n-way set-associative solutions have been proposed.

Replacement policies

When the cache is full and a new miss occurs, the cache is forced to evict the content from one of the currently occupied cache lines and make space for the currently referenced one. The candidate to be evicted is selected based on the predefined replacement policy. For cache with direct-mapped placement this is a trivial problem, since there is only one cache line where the memory block can be placed. However, associative organized cache needs to choose which line form the set to replace in order to place the upcoming memory block. For a set associative cache, the selection is restricted only within the set, while the selection of the set is chosen based on the address index. Fully associative caches have a wider freedom, since any cache line can be a potential candidate for eviction. The replacement algorithm should be a hardware implementation in order to perform quickly and have no negative impact on the cycle length.

There are five common policies that can be found in modern caches: *random replacement* (RR), *first-in first-out* (FIFO), *least recently used* (LRU), *pseudo least recently used* (P-LRU) and *most recently used* (MRU).

- The random replacement policy selects the candidate for eviction randomly. The advantage of this algorithm is that it does not need to keep a record of the previous accesses. However, implementation of a pure random policy in practice is very difficult. Therefore, caches that require randomness on replacement usually employ some reasonable pseudo-random approximation to select the block for eviction [102].

- The first-in first-out replacement policy keeps track of the cache misses and evicts the entry that has resided in the cache for the longest time. Such an algorithm can be implemented as a simple circular counter where the counter points to the oldest cache line. Every time a new block arrives in the cache, the value of counter is incremented. The counter is updated only when cache misses occur. In case when the maximal count value is reached the counter is set to zero. A fully associative cache has only one counter, while a set associative cache requires one counter for each set.

- The least recently used policy keeps tracks on references of each block. Every time a cache line is referenced, it is positioned on the top of the order list while the others are shifted down. The least referenced cache line, which is at the bottom of the list, is the first candidate for eviction. In contrast to FIFO, LRU is updated on each cache access (miss or hits).

- The pseudo least recently used strategy is an approximation of LRU that tries to reduce the cost of implementation. It guarantees that the candidate for eviction is not the most recently used cache line, but that candidate is not always the least recently used one. The

implementation is based on binary decision tree consisted of $n-1$ nodes, where n is the number of ways in cache associativity. The bit in nodes directs the path towards the cache line that will be replaced. For instance, the policy can be set to point the cache line for replacement by follow the path of zeros. In case of a hit, the bits that directs to the cache line will be inverse in order to protect the last accessed line from eviction.

- The most recently used policy protects the most recently referenced cache line form eviction. Its implementation is simple. Each cache line has one bit that is set to one when the cache line is accessed. In case where all bits become one and only one is zero, the policy resets all the bits to zero except for the most recent cache line that was zero, which is switched to one.

Although the LRU policy is the most successful in reducing the cache miss rate, the other policies are more widely used in practice because the implementation of LRU is challenging do to additional hardware and storage that requires to update the records on replacement.

3.4 Scratchpad Memories

Scratchpad is another on-chip memory alternative. It is a small and fast SRAM memory that can be accessed in one or two processor cycles. In contrast to cache, scratchpad has no additional hardware logic for managing its content, which makes it small by occupying a smalles area of the die and also more energy efficient. However, the content of the scratchpad must be managed explicitly through software provided from the programmer or compiler [117]. Such an approach allows the user to have full control on the scratchpad content, since nothing can be stored in it unless the software explicitly puts it there. Another distinction from cache is that scratchpads are not transparent to the processor but they are part of the memory address space with their own assigned address range. Thus, any access to the scratchpad must be performed through an explicit address. Embedded systems can have instruction, data or unified scratchpads.

The allocation of the program portion into the scratchpad can be done *statically* or *dynamically*. A static approach loads the selected code fractions into the scratchpad before the start of the program and keeps its content unchanged during the whole execution. On the other side, dynamic allocation swaps the content between main memory and scratchpad during the execution based on the flow of the execution. This approach can be considered as more scalable, especially for large applications. The detailed knowledge about the code that is gained during the compile stage can be used to determine which fractions of the code should be located into the scratchpad. The selection criteria depend on the goal that is aimed to be achieved since scratchpad memory has been shown to be successful in improving system performance, energy consumption and timing predictability [115].

Despite their beneficial properties, scratchpads have also disadvantages [5]. First, the compiler has to do additional work to determine fractions of the code that have to be allocated into scratchpad based on some predefined algorithms. Second, compiler must take into account the target of control-flow instructions, since the scrachpad has its own address space. The target address of each control-flow instruction that points to a destination that is allocated to scratchpad needs to be changed. Third, the transfer time of the code fractions between scratchpad and

main memory generates additional overhead. Fourth, the scratchpad requires additional control instructions inserted into the code which increases the code size.

3.5 Main Memory

Main memory is the largest operational storage of the embedded system. It stands in a separated die and communicates with the processor through the bus. Main memory is composed of a DRAM storage device and memory controller, where the DRAM storage keeps the code, while the memory controller intermediates between processor and DRAM by handling the requests from the processor to the DRAM device. In many cases, when the embedded system requires fast but not large operational memory, a chip of SRAM technology can also be used.

DRAM Chip Organization

A DRAM chip has a set of DRAM cells organized in an array with rectangular form where the access to each cell is performed as an intersection of *rows* and *columns*. A row is a group of parallel cells that are activated through a single row line, while a column is a fraction of data from that row that is transferred to the memory controller. DRAM memories with more than one array can be organized in different ways. If the memory arrays are considered as a single unit, then they operate in unison where the width of the column gives the number of arrays that are accessed. For example, a x4 DRAM organization means that the die has four DRAM arrays acting as one and each access provides four bits of data (one bit from each column). The collection of arrays that work as a single unit and respond to the same commands are called *banks*. The other solution is when the arrays are operating independently form each other, where each array is a bank in itself. Independently means that each bank can be activated, precharged, read or write independently from the activities of the other banks that belongs to the same die. Modern DRAM devices contain multiple banks consisting of multiple arrays in order to allow pipelining of operational commands. In practice, interleaving many independent memory banks has been shown as a successful approach for achieving higher memory bandwidth. In such case, the bus uses higher frequency and switches back and forth between banks. A set of DRAM chips can be further organized into *ranks*, which act as a unison of DRAM devices where internally each device can have one ore more independent banks [52].

Figure 3.6 illustrates an example of main memory consisting of two DRAM chips where each one has a single array. In this case each array is also a bank in itself. The selection of chips is done through *chip-select* signal that is part of *control signals*, while the address lines are divided between row and column to select the required cell. In fact, the DRAM selects the whole row and transfers the data of that row to a row buffer. The column part of the address selects a fraction of data from the buffer and sends them to the bus. This is the smallest unit of transaction performed between DRAM and memory controller.

Communication between memory controller and DRAM storage is done through three different types of memory buses: data, address and control bus. The data bus is usually 64 bit wide and is used for transfer of data to and out of the DRAM storage. The address bus carries the address of the data location. Its width depends on the size of the DRAM chip. In order to keep

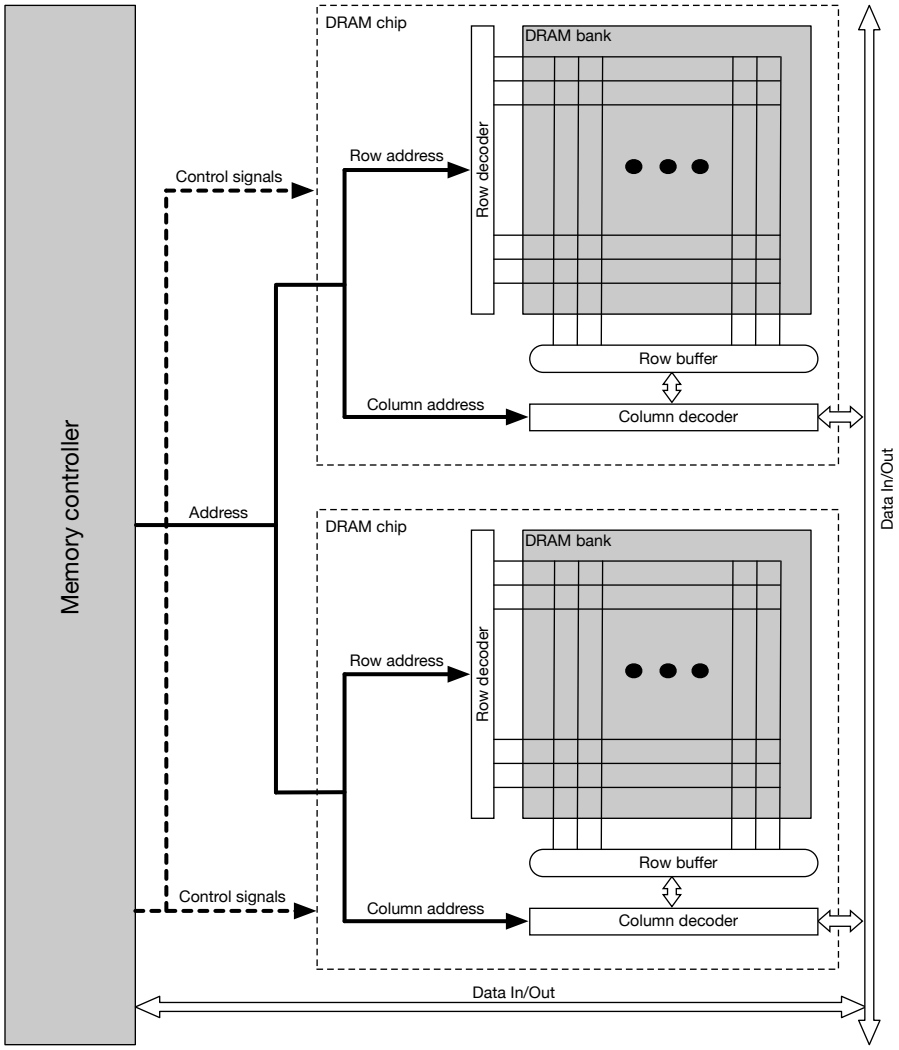


Figure 3.6: DRAM memory organization

the price of the DRAM low, the row and column access are multiplexed using the same address pins. The control bus is composed of strobe signal, output enable, clock, clock enable, chip selection, and all other signals that are used to manage the data transfer between the memory controller and the DRAM chip. In terms of communication, address and control buses perform unidirectional communication while the data bus is bidirectional.

Memory Controller

The memory controller serves as communication interface between the processor and DRAM die in order to hide the complex details from the processor when DRAM memory is accessed. This is achieved through a well defined protocol that moves the data in and out of the DRAM

device in accordance with timing access constraints defined in the DRAM-memory specification. The part of the memory controller that services the requests form the processor is known as the *front-end* of the controller, while the other part that deals with the DRAM device is called the *back-end* [3].

The front-end accepts the incoming requests from the processor, performs arbitration if there is more than one source of request, and queues them to the back-end. When the requested data is brought from the DRAM device to the memory controller, the front-end informs the processor through control signals that the data is ready for transfer and releases it into the bus.

On the other hand, the back-end reads the requests queued from the front-end, performs scheduling if there is more than one queue, and translates them into a set of commands which are adequate for the DRAM chip. Since the DRAM is organized as a set of arrays, the back-end firstly decomposes the requested address into a command for identifying the rank, bank, row and column within the memory chip. Once the data is localized within the DRAM, the cells are activated and the information is transferred to the memory controller [52]. The back-end is also responsible for the refreshing period by releasing periodically the refresh command to recharge the cell capacitor.

Memory Access Protocol

A memory access protocol defines the sequence of commands to access data in the DRAM device and the timing constraints that these commands should meet when sequential accesses are performed. Different DRAM memory systems have slightly different access protocols, but the set of fundamental commands is similar for all of them.

The protocol itself can be observed as a three step process. During the first step the location of the row that contains the required data within the DRAM chip is determined. At this stage the address of the required data and the control signals to access that row are sent to the address and command bus. The second step covers the process of transferring the data from the selected row to the sense amplifier. Since the voltage of the storage capacitor is extremely small, the DRAM uses the sense amplifier to detect faster the values of the row and also to recharge them. Once the data are brought to the sense amplifier they can remain there until the next row access is triggered, thus making the sense amplifier to act as a row buffer. During the third step the protocol moves the data from the row buffer through the data bus to the memory controller if the request is for read, or in the other direction if it is a write request.

All read and write operations on the DRAM are performed through a set of commands. The first command that the memory controller issues for any type of access is the *row access command* or also known as row activation command, which selects the requested row, moves the data from its array to the sense amplifier and then restores the cells values of that row. The time that the row access command needs to move the data from the row to the sense amplifier is called *Row-Column Delay* (t_{RCD}), while the time need to restore the values of the cells is called *Row Access Strobe* (t_{RAS}). Next is the *column read command* which moves the data from the sense amplifier through the data bus to the memory controller. The time required to access the column from the row buffer is known as *Column Access Strobe* (t_{CAS}), while the time needed for the burst transfer through the data bus is (t_{BURST}). Modern DRAM memories move the data internally in short and continuous bursts as well. The duration of this process is labeled as

(t_{CCD}), which also represents the minimum timing between column to column commands. In contrast to column read, the *column write command* moves the data in reverse direction from memory controller to the selected array. The timing between assertion of the command and placement of the data into the data bus is called *Column Write Delay* (t_{CWD}), while the time to propagate the data to the array is *Write Recovery* (t_{WR}).

After each row access, the sense amplifier and the bit line needs to be reset in order to be prepared for the next row access. This is known as *precharging*, which brings the voltage of the bit lines and the sense amplifier to the reference value. The timing parameter associated with the precharge command is t_{RP}. t_{RP} defines the time duration of this process. In modern system, precharging can be configured to be *open-page* or *close-page*. The open-page policy triggers the precharging process only when the read command has to switch to a new row line. This is especially beneficial for codes with high spatial locality, since it avoids the row access latency if the next access is done on the same row. In contrast to that, the close-page policy activates a precharge command after every read command and prepares the array for the next access. This solution shows better performance for a code that jumps more frequently through different memory locations.

The last is the *refresh* command, which is issued to restore the values of the DRAM cells. Triggering this command means reading the value of the cells and with that restoring their voltage to full level. The command should be issued regularly before the level of voltage in the capacitors falls below the threshold. However, the drawback of this process is that refreshing temporary blocks all read or write processes to the banks that are subject of refreshing. The process can be performed as *burst* or *distributed*. Burst means that all rows of all DRAM banks are refreshed one after the other and during that period no other action in the DRAM device can be performed. On the other hand, distributed refreshing spreads the refreshing process through wider timing interval where each issued refresh command affects only one row from all banks. There are few different ways to refresh an array [12]. The first option is called *RAS-Only* refresh, where the RAS signal is activated and a row address is issued on the address bus. It is the duty of the memory controller to provide the correct address and make sure that all rows are refreshed in turn. The second option is *CAS-before-RAS* refresh, where the CAS signal is activated and then the row selected. The difference between these two options is that with RAS-Only the memory controller has to provide the refresh address, whereas CAS-before-RAS keeps track on the refresh address using an internal counter. The third option is called *Hidden* refresh which is the same as CAS-before-RAS except that with hidden refresh the data from the previous read or write process are still available in the bus. All these refreshing techniques have been used for asynchronous DRAM. The evaluation of asynchronous DRAM to SDRAM modified the refreshing technique as well. SDRAM uses *auto-refresh* and *self-refresh* techniques for bank refreshing. When auto-refresh is triggered, both CAS and RAS are asserted and the internal counter releases the address of the rows to be refreshed. The memory controller only activates the refreshing process, while the internal counter is responsible for the address of rows that should be refreshed. In contrast to auto-refresh, self-refresh is performed when the system is idle in order to save power, where the DRAM device internally generates refresh pulses using an internal timer.

3.6 Prefetching

Prefetching is a technique that aims to enhance memory-system performance. Its job is to anticipate upcoming cache misses and issue prefetch requests a few cycles in advance in order to bring the memory blocks into the cache before they are requested [104]. The process is performed in parallel with the normal CPU operations by utilizing free bus cycles. Through overlapping, the prefetcher achieves to hide from the CPU the long latency of memory accesses. In the following we illustrate a simple example of how the system benefits from prefetching. When the CPU fetches the memory block with address *X*, the prefetcher assumes that the execution will continue through the next consecutive memory block as well and for that reason it issues a prefetch request for the following sequential block. If the cache line is of size four instructions and the current address is *X* then the address of the next memory block is *X+4*. Hence, while the execution progresses through instructions with addresses *X*, *X+1*, *X+2*, and *X+3*, the prefetcher will issue the request and prefetch the memory block with address *X+4*. Memory blocks that are fetched before they are referenced are called *prefetched* blocks.

Metrics and Terminologies for Prefetching

The main metrics used to evaluate the effectiveness of prefetching are: *coverage*, *accuracy* and *timeliness* [32]. Coverage represents the fraction of the original cache misses that are covered with prefetching. If the guess on a prefetch target is correct then the prefetch request is qualified as *good prefetching* and if a wrong item is loaded then the request is qualified as *bad prefetching*. The second metric, accuracy, represents the fraction of prefetched cache lines that are good prefetching. If the code has in total *M* misses and the number of good prefetches is marked with *G* and bad prefetches with *B*, then the coverage and the accuracy of prefetcher can be calculated as:

$$Coverage = \frac{G}{M} \tag{3.3}$$

$$Accuracy = \frac{G}{G + B} \tag{3.4}$$

Timeliness is the time distance *D* between the moment when the prefetch request is issued and the moment when the corresponding address is accessed. As the distance increases from zero, the amount of hidden latency is also increased. The miss latency can be hidden completely if the timeliness distance is wide enough, but not so wide because the prefetched line can be evicted from cache before it is accessed. The metric for representing the timeliness distance can be expressed through the number of clock cycles, the number of branches, the number of instructions, or the number of misses that occur between the moment of issuing the prefetch and its corresponding fetch request. Each of these measure is valid and conveys unique information.

However, in a real implementation, these metrics can work against each another. For example, increasing the coverage means employing an algorithm that performs more aggressively, but on the other hand reduces the rate of correct guesses and with that also the accuracy. Similarly, improvement on prefetch distance could increase the chances for eviction of memory blocks that are still useful. This also affects the cache miss rate by generating new cache misses that would not occur if the prefetching were not active [93].

Prefetching Techniques

Prefetching can be implemented either as software or hardware solution. Software prefetching employs explicit prefetch instructions that are inserted into the code and executed when prefetch requests need to be issued [15]. The position of prefetch instructions in the code defines the triggering moment of the prefetch request, while the content of the prefetch instruction dictates the address of the target that should be prefetched. The process of placing the prefetch instructions into the code is called *prefetch scheduling* and this can be done manually, by the programmer, or automatically through an intelligent compiler. Software prefetching generates a separate prefetch instruction for each prefetch action. Having explicit control on the prefetching moment and prefetch target of each prefetching request makes the software prefetcher to achieve high accuracy even for codes with complex structures. However, this type of prefetching has also a number of disadvantages. First, inserting prefetch instructions into the code for each prefetching action drastically increases the size of the code and with that also its execution time. Second, prefetch instructions require additional hardware modification so the CPU could support their execution. Third, scheduling the prefetch instructions within the code is not an easy task due to the difficulty of finding appropriate locations that would achieve optimal prefetch performance. This is especially tricky for references that have more than one predecessor path where the prefetch instruction can be placed. In such cases, the scheduler should be careful with the location so that the prefetch instruction would not generate useless memory traffic or cache pollution. Last, software prefetching rely exclusively on compile-time analysis and during this stage not all referenced addresses are available. For instance, prefetch instructions for indirectly addressed targets cannot be generated, although they could be good candidates for prefetching.

In contrast to software prefetching, hardware solutions are implemented through additional hardware logic that are part of the memory hierarchy and perform prefetching without any support from the compiler [104]. The job of the hardware prefetcher is to monitor the behavior of the code during its runtime and based on that, to issue the prefetch requests. The benefit of such an approach is that it eliminates the need for explicit prefetch instructions and the additional execution-time overhead. However, hardware solutions consume additional hardware resources, where the size and complexity depends on the type of behavior patterns that they try to anticipate. A prefetcher that employs simple prefetching algorithm, like anticipating and prefetching the next sequential block, has a small size on the die, but has limited success regarding its coverage and accuracy rate. On the other hand, a prefetching algorithm that tries to anticipate complex code behavior with higher coverage and accuracy needs a much larger and complex hardware design. In some cases, a complex hardware solution uses additional memory storage, which is organized in form of a table for keeping the history of the previously executed states and based on that anticipates the targets for the upcoming prefetch requests. Usually, the period of filling the table is considered as a training time during which the prefetcher is not effective. Although hardware prefetching does not generate any additional prefetch instructions, it often issues more unnecessary prefetch requests than the software solution. This is due to the lack of compile-time information that the software prefetching has. Such a behavior consumes higher memory bandwidth and in some cases it can result in cache pollution.

To overcome the limitations of the software and hardware solutions, a combined scheme of these two approaches has also been proposed [43, 71]. The idea is to use a hardware prefetcher

for targeting simple code structures, while the software prefetching would insert prefetch instruction for code structures which are more complex. Thus, through cooperation between these two solutions a new prefetching scheme will result, which trades off between the number of prefetch instructions and the consumed hardware size.

Prefetching Algorithms

Prefetching algorithms are distinguished from each other on how they anticipate on upcoming cache miss, but in general they can be classified into two main categories: those which are targeting sequential (continuous) misses and those which are targeting non-sequential (discontinuous) ones [106]. In this section we cover only instruction prefetching algorithms, considering that the instruction path of the memory is the focus of this thesis.

Sequential prefetch algorithms are the simplest form of prefetching, and also one of the most dominant solutions on modern processors [50]. With this scheme, the address of the prefetch target is calculated simply by incrementing the address of the current cache block [103]. The tendency of the compiler to generate code layout that is sequential makes this kind of approach quite efficient.

Always prefetching is the first and the most primitive sequential prefetching algorithm which initiates prefetching of the next cache line whenever a new cache line is fetched. The following extension of always prefetching is *Next-N-line prefetcher*, which attempts to reduce the memory access overhead by prefetching N consecutive cache lines after the one that is currently fetched. However, increasing the value of N increases the likelihood for prefetching useless memory blocks and generating cache pollution. To mitigate such an effect, *adaptive prefetching* has been proposed [26, 51]. While in fixed N sequential prefetcher the number of prefetched cache blocks is constant, an adaptive prefetcher tunes the degree of prefetching based on a dynamic measurement of prefetching efficiency. To do this, a *prefetch-efficency* metric is periodically calculated as indication for the spatial locality of the code. The metric is the ratio between useful prefetches and total prefetched blocks. Another solution to avoid cache pollution is adding a *stream buffer* where all prefetched lines are placed [54]. When a miss occurs, the stream buffer begins prefetching successive lines after the one that triggers the miss. Hence, subsequent accesses to the cache will also compare their address against the entries in the stream buffer. If the reference results in a miss in cache but a hit in the buffer then the cache is reloaded in a single cycle from the stream buffer. This avoids polluting the cache with prefetches that may never be referred.

To control the frequency of useless prefetches, variations of sequential prefetching schemes called *prefetch on miss* and *tagged prefetching* have also been proposed [104]. *Prefetch on miss* is similar to the previous sequential algorithms, except that the prefetching of memory block $X+1$ is performed only if the reference to the block X is a miss. *Tagged prefetching* associates each cache line with single *tag* bit used to detect when the line is demand-fetched or prefetched. The bit is set to zero when the line is prefetched from memory and changed to one when the same line is referenced from the CPU. Whenever the *tag* bit is switched from zero to one, a prefetch request for the next cache line is issued. This is similar to always prefetching, except that it avoids issuing repetitive requests for a cache line which has been prefetched and later replaced without having been referenced.

In contrast to the sequential prefetcher, non-sequential ones strive to capture the target of control-flow instructions and with that to overcome limitations of sequential schemes. This group of prefetchers is especially efficient when the program is composed of small functions and the transfer of control happens very frequently. Based on the style of how non-sequential prefetchers anticipate their prefetch requests, they are categorized as history-based schemes or execution-based schemes [106]. History-based schemes have a table which is maintained and consulted through the execution in order to determine the address of prefetch targets. Execution-based schemes use additional hardware, like branch predictors, to run ahead and explore the target of the upcoming control-flow instructions and bring them into the cache before they are demanded by the fetch unit.

A *target-prefetcher* is one of the earliest non-sequential prefetch implementations consisting of a small table that retains information about the behavior of control-flow instructions, and based on that history determines the prefetch target [105]. The table is organized in form of a list of pairs, where each pair is composed of a current line address and target line address. Whenever the program counter changes from one address to another, the prefetcher uses this value and searches through the table for a possible match. If a match is found, then the target address from the table becomes a prefetch-candidate address. The content of the table is dynamic and updated whenever there is a line change. If the missed line is the next consecutive line, the table is updated for that too. Later, the same approach was augmented with a filter that checks the queue with prefetch requests against the most recent demand fetches in order to eliminate possible useless prefetch requests that are already served as fetch demand [106]. A similar scheme is *wrong-path-instruction-prefetching* as well [81]. The major difference from the previous two schemes is that the last scheme does not save any target address. Instead, it prefetches the fall-through and then immediately the branch target after the branch instruction is decoded. Thus, if the execution returns to the branch, the target has already been brought to the cache.

Markov prefetching uses a Markov transition diagram to model the miss address stream and based on that anticipates the featured memory references [53]. Nodes of the model are representing the missed references while transitions hold the probabilities that the current node will be followed from the other one. However, this model can become arbitrary large since the execution can have different reference pattern for different iterations. In practice such a model is implemented through dedicated hardware that is organized in form of a table with limited size. This constrains the number of states that can be recorded and with that also the accuracy and the coverage of the prefetcher. Each state of the model occupies one row of the table and each row has the addresses of a few transitions that can be made from that state. The decision about which transition will be taken depends on their associated priority.

Branch-prediction-based prefetching is an aggressive algorithm, where the prediction of the prefetch target address for control-flow instructions is driven by the branch predictor [17]. The whole concept is based on a separated pseudo-program counter called Look Ahead Program Counter (LA-PC) which runs ahead of the regular program counter (PC) [18]. The prefetch unit is an autonomous state machine, that fetches the cache line from the cache and runs ahead through instructions till a branch instruction is encountered. In the same cycle, the prefetcher predicts the direction of the branch by using a two-level branch predictor. However, if the actual

outcome of the branch is different from the prediction, then the PC value of the prefetcher is reset to the real PC and also the content of the branch history register is reset to that of the execution unit. When the examination of the cache line reaches the end of the line and no branch instruction is encountered, then the prefetcher will continue with prefetching the next n sequential cache lines.

Fetch-directed-instruction-prefetching (FDIP) performs non-sequential prefetching [95] by using requests generated from the branch predictor. This solution decouples the branch predictor from the instruction cache in order to allow the branch predictor to proceed independently through the code when the instruction cache gets stalled. The asynchronism between these two components is bridged through a buffer called Fetch Target Queue (FTQ). The entries from FTQ are used as input for the prefetch unit. To be more effective, the prefetcher implements filtering as well to avoid the prefetching of blocks that are already in the cache.

Temporal-instruction-fetch-streaming prefetching algorithm records the sequence of cache misses and then uses the trace to predict and prefetch cache misses when the same stream recurs [41]. The success of the approach depends on stability and repetitiveness of the instruction reference. However, the uncertainty imposed from data dependency branches and branch predictor reduces the ability of the algorithm to predict the sequence of accesses and with that its effectiveness. Later, the algorithm was modified into *Proactive-instruction-fetch* algorithm [40], where instead of cache misses the sequence of instruction references was recorded. The correctness of the stream is achieved by recording retire-order instruction sequences, that are not affected by speculative execution, branch predictor or hardware interrupts. When the same recorded address sequence recurs, the prefetcher simply starts with prefetching based on that sequence.

3.7 Chapter Summary

A memory hierarchy is a stack of layers consisting of on-chip and off-chip memories. On-chip memories are small and fast memories located near the processor and keep only a fraction of the code and data. Off-chip memories are larger and can be of type SRAM or DRAM, depending on the size of the application. Their communication with the processor is done through a memory controller, while the connection can be organized as Von Neumann or Harvard architecture.

A prefetcher is an additional component of the memory hierarchy that brings memory blocks into the cache before they are referenced. Simple prefetchers are easy to build but may bring useless cache lines which can interfere with useful lines, by evicting them before they are used and also generating useless traffic. This results from the inability of these prefetchers to guess all the prefetch addresses due to the simplicity of the implemented algorithms. On the other hand, a complex implementation prefetcher with higher accuracy demands a complex solution with huge metadata and latency overhead. Storing the data off-chip will need frequent and costly communication, while storing it on-chip consumes precious chip size.

CHAPTER 4

Time-predictable Instruction Prefetching

In this chapter, we describe a time-predictable instruction prefetcher for single-path code. The chapter starts with the requirements that a prefetcher needs to posses in order to be effective and time-predictable. Next, the chapter continues with a presentation of a new prefetching algorithm for single-path code. Details on how the proposed prefetching algorithm anticipates the prefetch target address for different single-path code structures and how all of this is performed in an effective and time predictable way is also described. After that, the chapter presents the architecture of the prefetcher, firstly by showing the architecture on a higher level as a set of well organized modules and then by continuing with the details of each module. The chapter ends with a description of the process that generates the static information of the code, which the prefetcher uses as control input.

4.1 Towards Effective and Time-predictable Prefetching

For a long time prefetching has been known as an approach for hiding the long latency of memory accesses [104]. However, its presence in a system does not mean improvement of performance by default [111]. There are cases when a prefetcher behaves contrary to its purpose. For example, it can happen that the prefetcher guesses a wrong target and brings into the cache an incorrect memory block. Another failure that can happen with a prefetcher is when the correct target is brought into the cache but the prefetching process is triggered at a wrong moment. In both cases prefetching is useless, since it keeps the memory bus busy with unnecessary memory traffic and brings into the cache memory blocks that will not be referenced by the processor. The situation gets even worse when the uselessly prefetched memory blocks evict memory blocks from the cache that will be referenced in a near future. In such a case, the prefetcher becomes even harmful because it degrades system performance by generating new cache misses that

would not occur if prefetching had not been performed. The process of evicting useful cache contents with a useless prefetched memory block is known as *cache pollution*.

To achieve effective and time-predictable prefetch behavior the prefetch algorithm must consider the following criteria [38, 112]:

- Target addresses prediction - The algorithm should be able to guess with full accuracy the address of every cache block that is a candidate for prefetching. In other words, this criterion should answer correctly the question of *what to prefetch*;
- Prefetch lookahead distance - The algorithm should calculate precisely the release moment of each prefetch request. In other words, this criterion should answer the question of *when to prefetch*;
- Placing prefetched block - The algorithm should choose the right location in the cache to place the prefetched block without destroying any useful residing block. This should answer the question of *where to place new content*;

A deviation from any of these three criteria during the design stage of the prefetcher would result in an algorithm that would interfere with the normal cache-processor operations and generate cache pollution. In the following, we describe the main issues that a prefetcher has to deal with in order to fulfill the above requirements.

Target address prediction

The duty of the instruction prefetcher is to foresee the execution stream of instructions and based on that to anticipate the addresses of the following memory blocks. However, prediction of the upcoming memory blocks is a difficult process because the execution can run sequentially through consecutive memory addresses or it can be transferred to some remote memory location and continue from there. When the execution flows through sequential segments of the code, the anticipation of the upcoming target address is an easy and straightforward process since the block that should be prefetched resides in the next consecutive memory address of the currently active block. The difficulty emerges when the prefetching algorithm has to deal with non-sequential structures. Transfer of control happens as a result of control-flow instruction execution. However, the decision on transferring the execution may be mandatory or optional, depending if the transfer-of-control instruction is condition-dependent or not. Condition-dependent control flow instructions change the direction of execution only when the outcome of their condition is *true*, otherwise the execution will just fall-through. Hence, for such code structures the non-sequential prefetcher must firstly predict the condition outcome of the control flow instruction and after that calculate the target address. While the target address of direct control-flow instructions can be known during compile time, the outcome for indirect control-flow instructions, procedure (function) returns and indirect jumps, can be known only when the code is executed. Considering that the prefetcher should make the prediction of the target addresses ahead of time, it means that for these types of instructions it is almost impossible for the prefetcher to be accurate on target address prediction.

Another issue related to the *what to prefetch* question is the level of aggressiveness that the prefetching algorithm should realize. A conservative approach would issue fewer prefetching requests but would have a higher rate of accuracy on the target addresses. Although such a solution achieves better predictability on prefetched blocks, the level of efficiency on performance improvement can be insignificant due to the low number of the requests that are issued. On the other hand, an aggressive approach achieves higher coverage, but has smaller accuracy because a large number of the issued prefetch requests will result with erroneous target addresses [16].

The last parameter that the *what to prefetch* question has to consider is the number of blocks that a prefetcher should prefetch when a single prefetch request is issued. Each prefetch request can be configured to bring into the cache one or N continuous blocks. When a request prefetches a single memory block it reduces the possibility for cache pollution, since it makes a guess for only one memory block. But when each memory block is prefetched individually, a separate request for each prefetch candidate needs to be issued which will generate additional memory access latency. Prefetching N consecutive memory blocks, on the other hand, has the advantage of shrinking the memory access latency by eliminating the need for more prefetch requests to be sent to the memory, but this solution increases the probability to bring into the cache memory blocks that are then not referenced by the CPU.

Prefetch lookahead distance

Prefetch algorithms are devised to issue prefetch requests a few cycles before the target is referenced from the CPU in order to provide enough time distance for the prefetcher to be able to mask the prefetching process. However, the moment of issuing the request is quite critical. If the request is issued too early then the chances for the prefetched block to remain in the cache before being referenced are very low, because it can be evicted form the cache by a replacement operation. From timeliness point of view, this type of request can be called *early request*. On the other hand, if the request is issued too late then the prefetched block may not arrive in the cache before it is referenced from the CPU. In this case the processor will still be exposed to a stall, but the stalling period is shorter than the one which happens with conventional cache miss [112]. This type of request can be called *late request*. Therefore, a timely prefetch algorithm should issue the request at the right moment, which is early enough to hide the memory access latency but not that early as to be evicted form the cache before it is referenced. From the timeliness point of view, this type of request is called *timely request*.

Placing prefetched block

A prefetching process can be harmful even when the prefetch prediction is correct and the request is issued at the right moment, if the prefetched block is placed at a wrong location within the cache. This is because the prefetched block competes for cache location with the other resident memory blocks in the cache. Eviction of any residing memory block that is useful will generate a new cache miss. To avoid such a problem, the prefetcher should be designed to place the prefetched block in locations that do not affect any of the blocks which are potential candidates to be referenced in the near future.

The proposed solution for eliminating this problem is the use of a *stream buffer* as an augmenting part of the memory hierarchy, and placing all the prefetched block there [54]. When the CPU issues a fetch request, the same reference is also compared with the entries of the stream buffer. If the search in the cache results in a miss and a hit in the buffer then the block from the buffer will be immediately reloaded into the cache. This option protects the useful memory block in the cache to be victim of early prefetched blocks or from prefetch requests with a wrong target, but on the other hand it increases the complexity of the look-up hardware and the size of the on-chip memory.

4.2 Prefetching Algorithm for Single-path Code

The prefetching algorithm for single-path code is devised to be effective and time-predictable. Predictability derives from the property of the single-path code of having a single execution trace that stays fixed for any set of input data. Such a property enables the prefetcher to extract statically the whole knowledge on code behavior and later to use this for accurate anticipation of the target address of each issued request.

For the sake of efficiency, the algorithm is designed to perform aggressively as well, by issuing prefetch requests for every possible memory block that is part of the program. This is achieved by triggering the prefetcher and releasing a new prefetch request whenever the execution switches to a new cache line. In this way, the prefetcher always prefetches the next upcoming memory block and is at least one step ahead from the on-demand fetching process. The target of the issued requests depends on the flow of the execution, which can be the next sequential memory block or the block of some remote memory location from where the execution will continue. The decision is taken based on the knowledge that is collected form the statical analysis of the single-path code. Once the target is determined, the request is issued and the prefetching process is started. Hence, a combination of these two properties enables the proposed prefetching algorithm, in predictable and efficient fashion, to cover all the possible cache misses that can occur during the runtime of the code.

The granularity of the prefetching algorithm is determined to be on the level of memory blocks in order to minimize the possibility for cache pollution. Since the algorithm performs with full accuracy through the whole execution, the only pollution that can happen with single-path code prefetching algorithm is when the memory block, brought with prefetching, contains control-flow instructions which can dislocate the execution. However, this form of pollution is unavoidable due to restriction in the transfer between the cache and the main memory to a minimum of one cache line.

In the following we describe how the prefetching algorithm copes with sequential and non-sequential structures of the single-path code, what type of information are required to be derived by static analysis to achieve accurate anticipation of the prefetch targets, how the prefetched memory blocks are managed within the cache without generating cache pollution and what are the performance limitations of the proposed algorithm. In this section we also present a new prefetching scheme called *bulk* prefetching and describe how this form of prefetching is integrated together with sequential and non-sequential prefetching into a single and well synchronized prefetching algorithm for single-path code.

Sequential prefetching

Over half of the memory accesses of conventional code are performed on program segments with sequential addresses [38]. This number becomes even higher when a conventional code is converted into a single-path due to its fundamental transformation rule to serialize all input data-dependent alternatives of the code. With a structure of that form, the single-path converted code becomes even more suitable for sequential prefetching.

For our single-path prefetching solution we use the simplest sequential prefetching algorithm called *next-line always* prefetching [104]. Whenever a cache line is fetched from the CPU, the *next-line always* prefetcher immediately issues a request for the next sequential memory block. The address of the new prefetching target is calculated by incrementing the address of the currently active memory block. However, within the single-path code prefetcher this scheme of prefetching is active only when the execution runs through sequential segments of the code, thus preventing the sequential prefetching component to perform useless prefetch actions or to pollute the cache when the execution has to run through non-sequential structures of the code.

Non-sequential prefetching

Single-path conversion transforms all dynamic control-flow decisions of the code into static ones, thus making the outcome and the target of each control-flow instruction statically available. However, organizing and passing all this information to the prefetcher is a great challenge. The format and the amount of information that needs to be transfered to the prefetcher for accurate address prediction varies for different non-sequential code structures. In the following we describe the types of discontinuances that single-path code can have and the format of information that needs to be transfered to the prefetcher for its correct behavior.

Based on the type of discontinuance, control-flow decisions of single-path code can be categorized into: *if-branches*, *loop-branches*, *call-branches* and *return-branches*. Figure 4.1 depicts a fragment of a control-flow graph of a single-path code that includes different types of branches. The stream of execution for this example is A, B, D, E, A, C, F, I, C, D and E, where nodes F and I are part of another function. Node A contains an *if-branch*, node E a *loop-branch*, node C a *call-branch* and node I a *return-branch*.

If-branches of converted single-path code are input-independent, which means that the outcome of these branch instructions can be statically derived. To predict accurately the issuing moment and the address of the target memory block for *If-branches*, the prefetch algorithm needs to have information on the position of the branch in the code, the outcome of the branch condition and the address of the target if the branch is taken. Passing all this information to the prefetcher in form of a string can be expensive and space consuming. The use of a lossless compression algorithm [96] can be consider as an approach for reducing the size of the outcome string of if-condition. The algorithm needs to be lossless because the generated reconstruction from the compressed representation should be identical with to the original string. However, the pattern of the if-condition through the execution of the code can be irregular and quite long. This makes difficult to devise a general compression model that would be efficient for all strings of if-conditions. Furthermore, the presence of input-independent if-branches in code is very rare considering that most of the if-structures are input-data dependent and they are all serialized af-

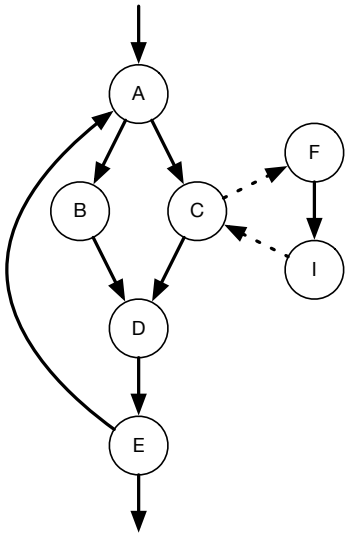


Figure 4.1: Control flow graph of single-path code consisted of *if-branch*, *loop-branch*, *call-branch* and *return-branch*

ter transformation. Therefore, for if-branches we propose to inform the single-path prefetching algorithm only on the position of the if-branches in the code and not about their behavior. Thus, whenever an if-branch is encountered, the prefetcher will not perform any prefetch action at that moment but will only follow the execution. Once the branch is executed, the prefetcher will observe if the branch is taken or not and based on that it will calculate the position in the code from where the prefetcher should continue with its actions. Such an approach prevents the cache state to be interfered with useless memory blocks by not allowing the prefetcher to make any wrong decisions on branch target and pollute the cache. Input-independent *if-branches* are the only structures of the single-path code for which the single-path prefetching algorithm does not issue a prefetch request.

The next code structure encountered in single-path code are loops. In our approach converted single-path loops have a single exit node and constant number of iterations, which means that the execution frequency of the loop is also a constant. Furthermore, the *loop-branch* decision for exit is always located at the end of the loop [84]. Therefore, for this type of discontinuity it is sufficient to inform the prefetcher only on the position of the loop-branch within the code, the loop header and the number of iterations. Hence, whenever the prefetcher is triggered for a loop-branch prefetching, the back edge target of the loop will be prefetched. Once the iteration reaches the value zero, the prefetcher will dismiss the loop-prefetching option and will continue to prefetch the next memory block through sequential prefetching. However, there are cases when loop conversion is not required and the loop still generates a single trace. Figure 4.2a shows an example of a nested loop where the iteration of the outer loop is converted to its loop bound while the iteration of the inner loop remains dependent from the iteration of the outer one. Although the iteration of the inner loop changes due to dependency on the outer loop, the sequence of iterations of the inner loop will remain the same. This mean that the execution of the

```
for i = 1 to max_a
    for j = 1 to i
        statement
            ...
```

(a) Inner loop is set to i.

```
for i = 1 to max_a
    for j = 1 to max_b
        statement
            ...
```

(b) Inner loop is set to max.

Figure 4.2: Conversion of nested loops.

inner loop can be considered a single-path and its behavior can be statically derived. To predict the behavior of such a loop, the prefetcher would require additional information for each loop iteration and this can be expensive. One solution is to restrict the compiler to convert the inner loop to a loop with a constant number of iterations as shown in Figure 4.2b. Another approach is to inform the prefetcher for the *loop-branch* of the inner loop as it would be an *if-branches*. In this case the prefetcher will not perform any prefetching whenever the *loop-branch* of the inner loop is encountered.

In single-path code, functions (procedures) are always called and executed, but the value of the predicate related to the function determines if the execution will affect the hardware states or not. Therefore, the prefetcher will always prefetch the target of the call instructions when this type of discontinuity is encountered. Information that are passed to the prefetcher for this kind of code structure are only about the position of *call-branches* within the code and their target addresses.

Unlike the other types of single-path branches that have always the same target, *return-branches* can have different ones and this depends on the position in the code from where the function is called. Although the string of targets for each return instruction can be statically derived, transferring this to the prefetcher will mean a huge amount of information is passed. Therefore, for this type of branch, we inform the prefetcher only on the position of the *call-branches* within the code without giving any further detail on its behavior. As the code runs, whenever a call instruction is executed the address of the instruction that follows that call is stored in a stack. Once the prefetcher arrives at the end of the function and the return target needs to be prefetched, the value of the target is read directly from the stack. Such an approach gives enough time to the prefetcher to read from the stack the information about the target address and issue the request when a *return-branch* needs to be prefetched.

Bulk prefetching

In addition to sequential and non-sequential prefetching, the single-path code algorithm also performs *bulk prefetching*. This form of prefetching is active only when the execution iterates through a loop which is smaller than the cache size. In such a case, all CPU fetch accesses will result in cache hits, since the entire loop will be found in the cache, thus leaving the memory bus free. The prefetching algorithm uses this opportunity and engages the bus to bring more memory blocks into the cache that will be accessed after the loop execution will be finished. For this form of prefetching, the prefetcher requires information about the address of the last memory block

of the loop in order to determine the moment when this type of prefetching should be triggered and the number of blocks that can be prefetched when bulk prefetching is active. The number of memory blocks that can be prefetched with *bulk prefetching* is determined statically through analyzing the execution trace of the code.

Placement of the prefeched block within the cache memory

The single-path prefetching algorithm brings prefetched memory blocks directly into the cache without affecting the temporal property of the cache at any moment. The main purpose is to not generate any additional cache miss compared to the execution without a prefetcher. For sequential and non-sequential scheme this advantage emerges by design since these two schemes have a prefetch granularity of one memory block and both of them anticipate the prefetch targets with full accuracy. Thus, the cache line that is replaced with the prefetched block is the same one that would have been replaced if there were only ”on-demand” fetching, except that now with single-path code prefetching the process of eviction begins a few cycles earlier.

The situation is different when bulk prefetching is performed. Since few memory blocks in a row are prefetched with this scheme, the prefetcher needs to be careful about the cache lines that will be evicted from the cache in order to prevent temporal property of the cache and with that to maximize the benefit gained from temporal locality of the code. Bulk prefetching is active only when loop smaller than the cache size is executed, therefore the attention should be on the cache lines that hold the loop and avoid any sort of cache conflict between them and the prefetched block. The single-path prefetching algorithm prevents this from the stage of code analysis when information about the possible free cache line is generated. During this process the analyzer takes into consideration the size of the cache, its replacement algorithm, the size of the loop, its position within the cache and based on that it determines the number of free cache lines that can be replaced without affecting the temporal properties of the cache. It is important to note that in cases when a loop body contains call instructions then the analyzer considers for loop size the space that is required for loop memory blocks and the blocks which belong to functions that are called from that loop.

Limitations of the Single-path Prefetching Algorithm

Fundamental for a prefetcher to be effective is to run in parallel to the other CPU operation. This condition happens only when the CPU fetch request results in cache hit and the free cycles of the bus are used for prefetching. The single-path prefetching algorithm is devised to maximize the utilization of the free bus cycles, but its success in overlapping the fetch with prefetch process gets limited when the memory bus becomes the bottleneck of the system. Figure 4.3 illustrates this limitation by showing the length of overlapping that can be achieved when a sequence of a few memory blocks is fetched and executed. The blue quadrilaterals in the figure mark bus transfer time of the memory blocks, while the red ones their execution time. The first part of the figure (fetching) shows the execution time of the blocks without prefetching. In this case we have on-demand fetch, where the cache transfers the memory blocks from the main memory only when they are requested by the CPU. The CPU stalls during the transfer time and proceeds with the execution when the whole memory block is present in the cache. The second part of the

figure (prefetching) demonstrates the benefit on execution time when the prefetcher is present. As can be seen from the figure, the prefetcher starts immediately with prefetching once the bus becomes free. Although the prefetcher utilizes every possible free bus cycle, the access latency still can not be completely hidden due to the memory-wall problem [122]. For such a scenario, the prefetcher hides only part of the *cache miss latency* and with that partially mitigates the CPU stall time.

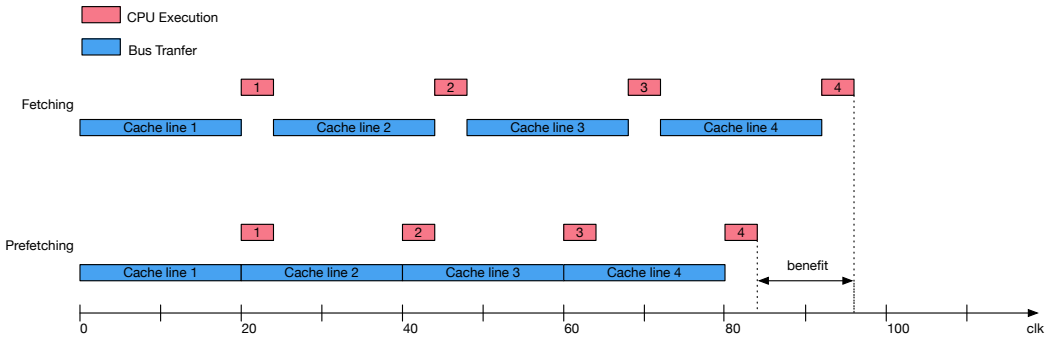


Figure 4.3: Reducing the cache miss latency through prefetching

If we assume that the program executed in Figure 4.3 consists of a sequence of n memory blocks $(l_1, l_2, ..., l_n)$, where the transfer time for each block takes t cycles and the execution of the block is constant and takes e cycles, then for a sequence of n blocks the execution time for an *on-demand* memory architecture will be:

$$time_{execution} = (t + e) * n \tag{4.1}$$

while for the architecture that includes a prefetcher the execution time of the sequence will be:

$$time_{execution} = (t + e) + t * (n - 1) \tag{4.2}$$

Under the assumption that each memory block is entirely executed, the maximal benefit that can be gained from prefetching is:

$$benefit_{max} = e * (n - 1) \tag{4.3}$$

The single-path prefetcher can fully hide the cache miss latency only when bulk prefetching is performed. Figure 4.4 illustrates the benefit of the prefetcher in such a scenario, where a few memory blocks are prefetched while the loop is iterating. Unlike the first case where the presence of the prefetcher reduces only the *cache miss latency*, here the prefetcher reduces the *cache miss rate*.

If we consider the same assumptions as in the above calculation then the number of possible memory blocks whose latency can be fully hidden with prefetching is:

$$N_{max} = r * (b * e) \; mod \; t \tag{4.4}$$

where r is the number of iterations of the loop, b is the number of memory blocks belonging to the loop, e is the execution time of one memory block, and t time needed to transfer one memory block from the main memory to the cache. The outcome is the maximal number of memory blocks that can be masked with prefetching during the execution of the loop.

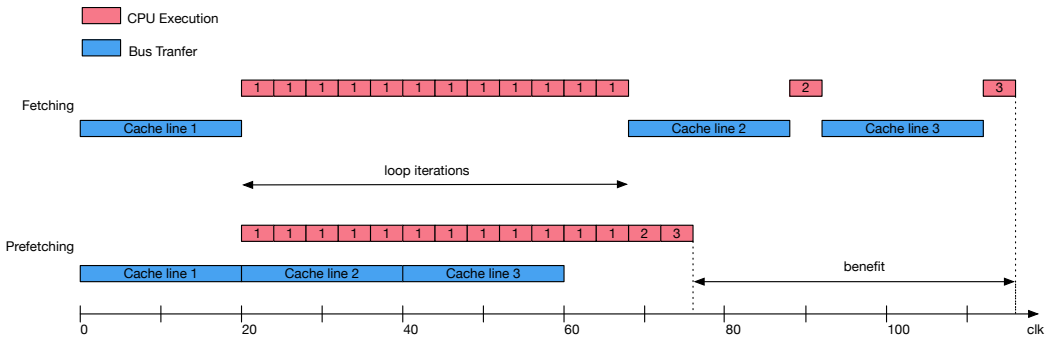


Figure 4.4: Reducing the cache miss rate through prefetching

4.3 Architecture Model of the Single-path Code Prefetcher

The single-path code prefetcher has been built as a separated hardware unit in order to enable its design to be developed independently from the processor and compiler. Another reason for having a hardware solution instead of a software one, is that this type of prefetcher does not require additional prefetch instructions in the code that would increase the execution-time overhead. However, in contrast to the other conventional hardware prefetching solutions where the knowledge on code behavior is attained during the runtime [40, 41, 107], for the proposed single-path code prefetcher this knowledge is statically derived and available before the execution starts. Thus, having all the required information on code behavior ahead of time allows the prefetcher to start with prefetching immediately, once the execution is triggered, and with that to cover not only capacity and conflict cache misses but also compulsory ones.

The architecture of the single-path code prefetcher is organized as a set of modules where each module performs one type of prefetching. This design makes the prefetcher scalable, because if a new prefetching scheme needs to be added it would be just another module. Figure 4.5 illustrates the high-level model of the architecture consisting of *If*, *Sequential*, *Call*, *Return*, *Loop* and *Bulk* module. The activation/deactivation of each module is orchestrated through a separate module called *Controller*, while the information about the behavior of the single-path code, which have been statically derived, are placed in the *Reference Prediction Table* (RPT). Part of the prefetcher is also the *Return Address Stack* (RAS), which is accessed only from the *Call* and *Return* modules. Although one return instruction can have multiple targets, for single-path code all target addresses as well as the order of their occurrence can be statically derived. However, such a solution would impose having multiple entries for the same return instruction in the RPT due to multiple-target addresses that the same return instruction can have. The employment of *Return Address Stack* (RAS) eliminates this issue and at the same time reduces the size of the table. Whenever the prefetcher encounters a function return, it reads the target from the RAS which has been pushed when the function was called. The modules and the table are connected with each other through control and data signals. In the figure, dotted lines represent control signal connections, whereas the data paths are shown with full lines.

From Figure 4.5 it can be seen that each address reference issued by the program counter (PC) is also forwarded to the prefetcher, which at any moment makes the prefetcher aware of the

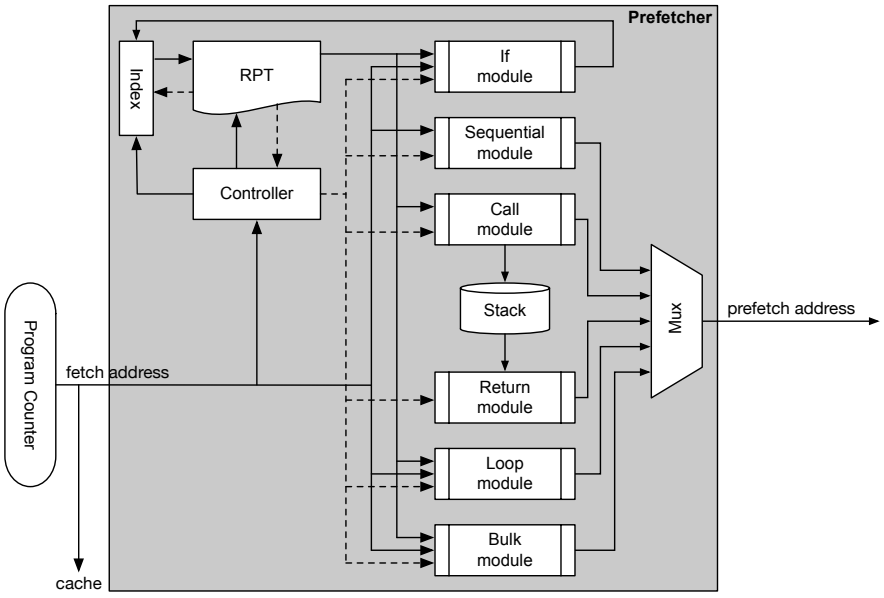


Figure 4.5: Architecture model of the single-path code prefetcher

position of the execution within the code. The prefetcher also knows the length of the cache line. Thus, whenever a new PC reference is received, the prefetcher firstly checks if the new reference is in the same cache line with the previous PC reference or the execution has to switch to a new cache line. The detection of cache line switching is based on the comparison of the cache-line bits of the current reference address with the cache-line bits of the previous one. If they have the same value, it means that the execution is still on the same cache line. If they differ then the execution has changed the cache line. Once the execution enters a new cache line, the prefetcher immediately triggers the algorithm and releases a new prefetch request for the following memory block. The calculation of the prefetch target address begins by forwarding the PC reference address to the RPT to check for a possible match. If the search in the table results in a hit, it means that the upcoming cache line is non-sequential and the calculation of the target should be done by one of the non-sequential modules. Which non-sequential module will be selected depends on the entries in the table. On the other hand, if the search in the table results in a miss, then the sequential module is activated. Once the target has been calculated the request is generated and released for prefetching.

Reference Prediction Table

The Reference Prediction Table (RPT) is a small memory organized in form of a table that holds code-related information for guiding the prefecher through non-sequential prefetching. In other words, the RPT is a projection of the control-flow structure of the code. Execution of the single-path code always follows the same execution trace, which means that the entries of the RPT will be accessed in the same order as they are produced and this order is preserved through any iteration of the code. This property eliminates the need to search through the whole RPT for a

possible match. Whenever a PC reference needs to be compared there is only one entry from the table that has to be checked and the position of that entry is pointed to by the *Index* register, illustrated in Figure 4.5. The value of the *Index* register is changed whenever the comparison results in a hit, in order to point to the next upcoming non-sequential prefetching entry. This solution simplifies the size and complexity of the hardware required for RPT implementation. Another advantage of having this form of table organization is that the RPT allows multi-targeting, even though the encoding granularity of the table is in the level of cache lines. For instance, if there are two or more control-flow instructions within one cache line, then the same cache line will be represented in the table as many times as it contains control-flow instructions and each entry will be encoded with a different prefetch target. The position of these targets in the table will determine their order of prefetching.

The type of entries that the RPT can have are separated in eight groups, where each group has a different column. In the following we describe in detail each of these columns.

- **Index** - Each row of the table holds entries for one non-sequential prefetch action. This column indexes these rows.
- **Trigger** - Holds the triggering address for non-sequential prefetching. If the fetch address matches with the trigger address then prefetching with entries from this row will be performed.
- **Type** - Determines the type of prefetch module that should be activated to work on the prefetch request. The type of entries that this column can have are *if*, *call*, *return*, *loop* or *bulk*, which are the same as the names of the modules that should trigger.
- **Destination** - Holds the target address for non-sequential prefetching, which is statically calculated and released as a prefetch request.
- **Iteration** - Determines the number of loop iterations. For loops in single-path code this number is constant due to the property of the single-path loops that have constant execution time.
- **Next** - Points to the next non-sequential entry by holding the next value of the *Index* register.
- **Count** - Determines the number of cache lines that can be prefetched when *bulk* prefetching is performed.
- **Depth** - Distinguishes the loop-nesting level for nested loops in order to prevent the iteration value of the counters from the outer loops to be jeopardized by the inner ones.

Table 4.1 illustrates an example of an RPT that holds all possible types of entries that this type of table can have. The first row shows the entries required by an *If*-module to perform prefetching when an *If* type of code structure is encountered. As can be seen from the table, this row has data in *Trigger* column which is the trigger address, in *Type* column to tell to the prefetcher the type of module that should be activated, in *Destination* column which is the

Table 4.1: Reference Prediction Table

Index	Trigger	Type	Destination	Iteration	Next	Count	Depth
0	8014	if	8020	-	3	-	-
1	8310	call	8305	-	1	-	-
2	8350	return	-	-	-	-	-
3	8780	loop	8205	24	2	-	1
4	9215	bulk	-	-	-	15	-

address of the destination if the branch is taken and in *Next* column which represents the new value of the *Index* pointer in case the branch is taken. Entries in *Trigger* column are necessary information for each type of non-sequential prefetching and are part of each row. The entries in the second row of the table are for *Call* type of prefetching. This row has data in the *Trigger*, *Type*, *Destination* and *Next* columns. For this type of prefetching the data in *Destination* column is the address of the target that should be prefetched, which in this case is the first memory block of the function. The entry in *Next* column is used by the *Index* register to point the position in the RPT of the next non-sequential prefetch action. The third row has entries for *Return* type of prefetching, which in contrast to *Call* has no data in the *Destination* and *Next* columns, because the target of return instructions and the next value of the *Index* register is read from the *Return Address Stack*. Entries for *Loop* type of prefetching are shown in the fourth row. Apart from data in the *Trigger* and *Type* columns, for this type of prefetching the table provides information about the address of the loop header in *Destination* column, how many times the execution will jump to the loop header in *Iteration* column, where the *Index* pointer should be located when the execution goes back at the beginning of the loop in *Next* column and the depth of the loop to distinguish if the loop is nested within some other loop in *Depth* column. The last row of the RPT example is about *Bulk* prefetching which determines the number of sequential memory blocks that should be prefetched while the execution iterates through some cached loop. This entry is placed in the *Count* column.

By default the prefetcher activates the sequential components and switches to non-sequential only when the search in the RPT results in a hit. The idea behind this solution is based on the tendency of single-path code to have longer sequential segments due to its main transformation rule to serializing all input dependent alternatives. Having this, single-path code has fewer control flow instructions compared to conventional code and with that also needs a smaller RPT.

Controller

The *Controller* is the module that coordinates all the actions within the prefetcher. Its job is to watch the stream of execution, to initiate prefetching when the execution switches to a new cache line, and to select the appropriate prefetch component for calculation and generation of the prefetch target address. In our solution, the *Controller* is modeled as a state machine consisting of states A, B and C as shown in Figure 4.6. State A is the initial state, which marks the readiness of the *Controller* to perform a prefetch action. When prefetching is triggered, and the actual prefetching is performed by the *Sequential*, *Call*, *Return* or *Loop* module then the state of the state machine does not change but remains in A since the calculation of the target addresses

for these types of prefetching is done within one cycle. This tells to the *Controller* that the actual prefetch action is finished and the prefetcher is ready to perform the next upcoming prefetching. The state machine transits to state B only when the prefetcher encounters an *if*-branch structure. For this type of prefetching, the prefetcher has the information about the position of the branch in the code but no information about its outcome. Therefore, the state machine transits to state B and stays there until the outcome of the *if*-branch is solved. At the same time, when the state machine transits to state B, the *If* module of the prefetcher is also triggered. In state B the prefetcher does not perform any prefetch action in order to preserve temporal properties of the cache and avoid any possibility for cache pollution. Once the outcome of the branch is resolved, the state machine switches back to state A and waits for the next prefetching action to be triggered. The *Controller* transits to state C when the *bulk* prefetching has to be performed. In this state, the *Controller* monitors the execution of the loop and the prefetching process. When the *bulk* prefetching has finished with the prefetching of memory blocks or the execution has exited the loop, the state machine will transit to state A immediately.

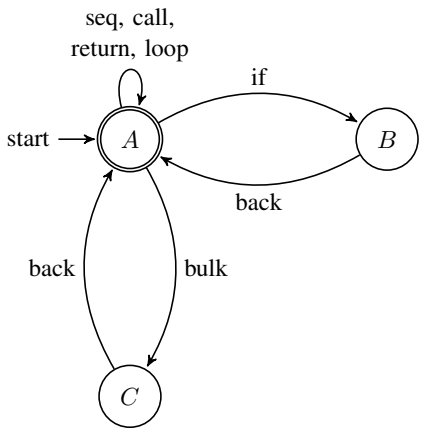


Figure 4.6: State machine diagram of the prefetch controller

Sequential Module

The *Sequential*-module, illustrated in Figure 4.7, generates the prefetch address of the following memory block and is triggered whenever the search in the RPT results in a miss. Its architecture has the simplest design of all other modules. It is comprised of one adder and one sign extend component. Whenever a prefetch request for the next sequential memory block needs to be issued, the cache-line address of the current PC fetch address is passed as an input through the adder, is incremented, and released as the cache-line address of the next memory block. The sign extension unit is used to extended the width of the cache-line address to the length of the full memory address so the request is in the same format as the address issued from the PC because the prefetcher works with cache-line addresses. This whole process is performed within one cycle.

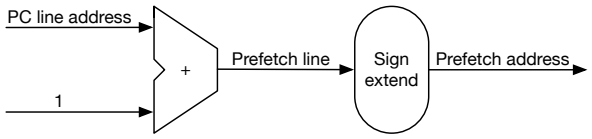


Figure 4.7: Sequential module

If Module

The *If*-module is activated when prefetching encounters an *if* code structure. This is the only module that does not issue any prefetch request, but just sets the prefetcher with the right settings and prepares it for the following prefetch action. The module calculates the next value of the *Index* register in order to point to the right position in the RPT once the if-branch has been solved. The architecture of the *If*-module, shown in Figure 4.8, is comprised of three registers (Index Reg1, Index Reg2, and Dest. Reg), one multiplexer, one comparator and one adder. They are connected in form of two parallel paths, where one path is for calculation of the next sequential index (index 1) and the other for reading the index from the RPT table (index 2). The next sequential index is just an increment of the current index value and is valid only when the if-branch is not taken. In case the branch is taken, the value of the index is read from the RPT table, which is statically calculated and stored ahead of time when the RPT table is filled. The decision which of these two indexes will be chosen depends on the outcome of the comparator, where the cache-line bits (PC line address) of the current PC reference are compared with the value of the *Destination* register, which are the cache-line bits of the destination where the execution will be dislocated if the branch is taken. The value of the *Destination* register is also statically calculated and stored in the RPT table. If the comparison results in a hit, it means that the branch is taken and the next index value is the one read from the RPT, otherwise the index value is just incremented.

The whole process of next-index calculation is performed in two steps. In the first step, when the *If*-module is triggered, the values of both indexes are generated and saved in their related registers, while in the second step the outcome of the branch is compared and the right index value is sent to the main *Index* register.

Call module

The *Call* module generates the prefetch address of the first memory block of the called function. As shown in Figure 4.9, this module does not perform any calculation, but only releases the prefetch request with target read from the RPT table and replaces the content of the *Index* register with the value read also from the table. Both entries, the prefetch target and the *Index* register value are calculated statically.

However, this module prepares the prefetcher for return prefetching when the end of the function is reached. For that reason, the architecture in addition includes two adders which are employed to increment the current PC address and the value of the *Index* register. Both incremented values are pushed on the *Return Address Stack* (RAS) located within the prefetcher, between *Call* and *Return* module. The whole operation is performed within one cycle.

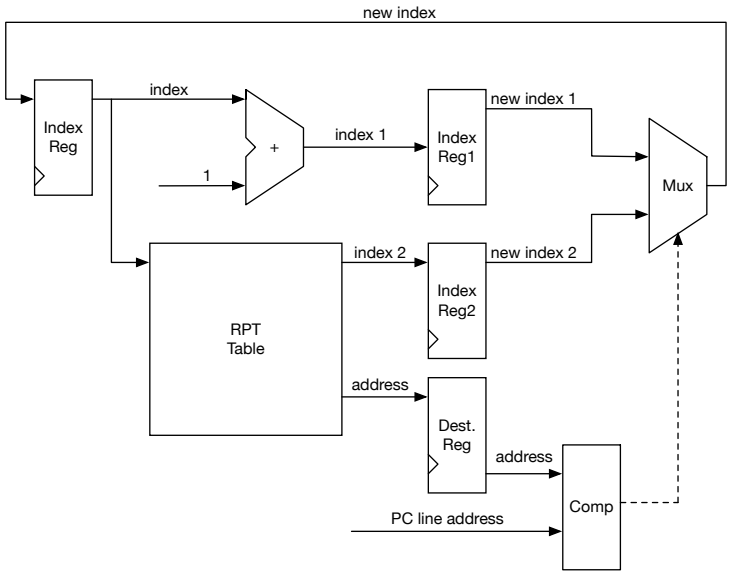


Figure 4.8: *If*-module

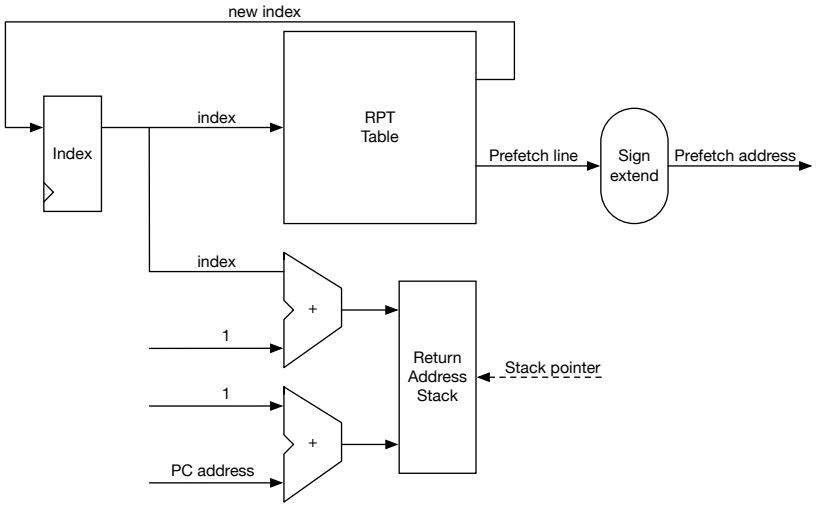


Figure 4.9: *Call*-module

Return module

The architecture of the *Return* module is shown in Figure 4.10. Its job is to issue prefetch requests for the cache line of the code from where the function was called. Whenever this module is triggered, it immediately pulls the pair *Destination* and *Index* from the RAS and decrements the value of the stack pointer. The first value of the RAS pair is used as prefetch target, which is the cache-line address from where the function was called. The cache-line bits

are extended through the sign extend and released from the prefetcher as a prefetch request. The second value read from RAS is the new value of the *Index* register. The whole process is performed within one cycle.

Currently, the RPT holds only a single entry for each return in the code by telling to the prefetcher only the moment during the execution when this module should be triggered. All other required information for operation of the *Return*-module are stored on the RAS. The presence of the RAS reduces the size of the RPT table because the RAS consumes less space due to the dynamic to change its pointer and to keep information only for the actually called functions.

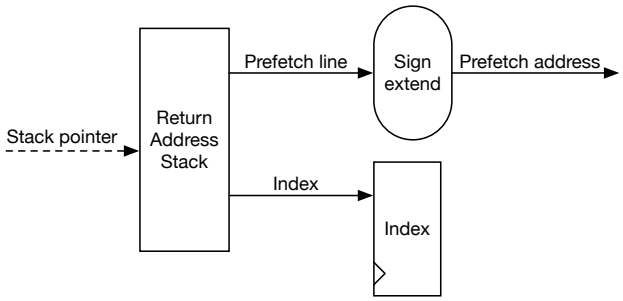
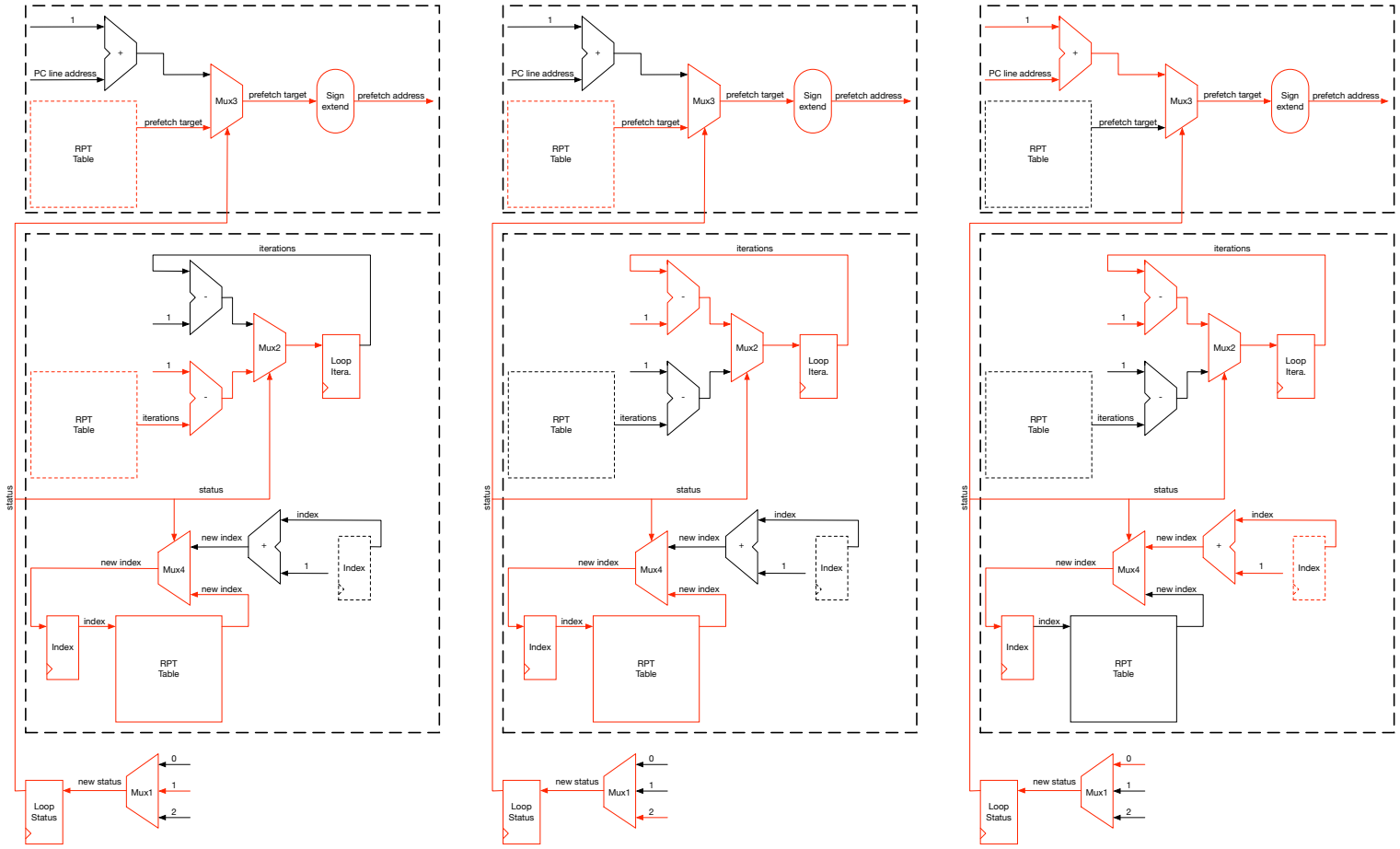


Figure 4.10: *Return*-module

Loop module

The *Loop* module, illustrated in Figure 4.11, is triggered when the prefetcher encounters a loop structure in the code. Its architecture is designed to operate in three modes called *entry mode*, *iteration mode* and *exit mode*. In Figure 4.11, the components of the module that are active in one mode are marked in red color. In all three modes, the upper part of the architecture, framed with a dotted line rectangle, generates the prefetch-target address whenever the *Loop* module is triggered. This part of the architecture is comprised of one adder, one multiplexer and one sign extend unit. The lower part of the architecture has to calculate the value of the *Index* register for pointing to the next non-sequential entry in the RPT and to count the number of iterations that the loop has performed. This part comprises three adders, two multiplexers and one register called *Loop Iteration*. The mode status of the *Loop* module is saved in the *Loop Status* register, and it switches from one mode to another based on the number of loop iterations that have been left. In the following we describe the behavior of *Loop* module through each mode.

By default, the *Loop* module always starts in *entry mode*. This mode is active only when the loop executes its first iteration and once the loop finishes this iteration, the mode is immediately changed to *iteration mode* or *exit mode*, depending on the number of iterations that are left. As can be seen from Figure 4.11a, in this mode the prefetcher reads the prefetch target address directly from the RPT table and after passing it through sign extension it is releases a prefetch request. Because this is the first iteration of the loop, the prefetcher reads the number of iterations from the RPT. This value is firstly decremented and then saved in the *Loop Iteration* register. In this mode, the value of the *Index* register is read from the RPT table as well.



(a) loop entry mode

(b) loop iteration mode

(c) loop exit mode

Figure 4.11: *Loop*-module

If loop has more than one iteration then the next mode of the *Loop* module is *iteration mode*. The module stays in this mode until the loop reaches its last iteration. This mode differs from the previous one only in the part of the architecture that counts the number of iterations. While in *entry mode* the number of iteration was read directly from RPT table, *iteration mode* decrements the value of the *Loop Iteration* register. This can be observed in the lower part of the architecture illustrated in Figure 4.11b.

For the last iteration, the *Loop* module switches to *exit mode*, where at the end of the loop body the prefetcher will not prefetch the target of the loop back edge but will bring into the cache the next sequential memory block. Therefore, in the upper part of the architecture in Figure 4.11c, the prefetch target address is not read anymore from the RPT but its value is calculated as an increment of the actual PC address. The part of the architecture that counts iterations will reset the *Loop Iteration* register to zero, while the part for index calculation will increment the value of the *Index* register to point to the next RPT entry. When this is finished, the mode is switched back to the *entry mode* waiting for the next loop.

In order to simplify the presentation of the *Loop* module, in Figure 4.11 we have shown the example for only one loop. In reality loops can be nested. Thus the architecture of the *Loop* module has a set of *Loop Iteration* registers and *Loop Status* registers, which are active at different levels depending on the depth of the nested loop. The depth level of each loop is determined statically and is read from the RPT from the *Depth* column.

Bulk module

The architecture of *Bulk* module, illustrated in Figure 4.12, can be viewed as two separate components, where one is for counting the number of prefetched blocks by this module (Counter) and the other for calculating the target address of the issued prefetch requests (Address-generator). The Counter component consists of one multiplexer, one adder and one register, while the Address-generator has the same elements as the Counter plus one sign extend. Before the state machine of the *Controller* switches to state C, the counter component of the *Bulk* module reads the number of sequential memory blocks that can be prefetched with bulk-prefetching from the RPT and saves this number in the *Count* register. At the same cycle the Address-generator component increments the cache line address of the last memory block of the loop and saves it into the *Addr* register. Once these two registers are set, the bulk-prefetching can be started.

The *Bulk* module becomes active only when the state-machine switches to state C. In this state the prefetcher cannot use the cache-line switching as a trigger signal anymore, because the execution now runs through a loops that is already in the cache and if the same triggering approach is used then the rate of issued prefetch requests would be so high that the memory would not be able to handle all of them. To avoid this problem, *Bulk* module uses the memory bus signal for triggering, which enables the *tag* fragment of the address to be written in the cache. In the protocol of bringing the memory block into the cache, this signal is the last one in the sequence of issued signals and therefore it is employed to insure that the requested memory block has arrived and a request for new prefetch can be released. After each issued request, the part of the architecture that counts the requests (Counter) will decrement the value of the *Count* register, while the other part (Address-generator) will increment the value of the *Addr* register. This process continues until one of the following two conditions is reached. The first condition

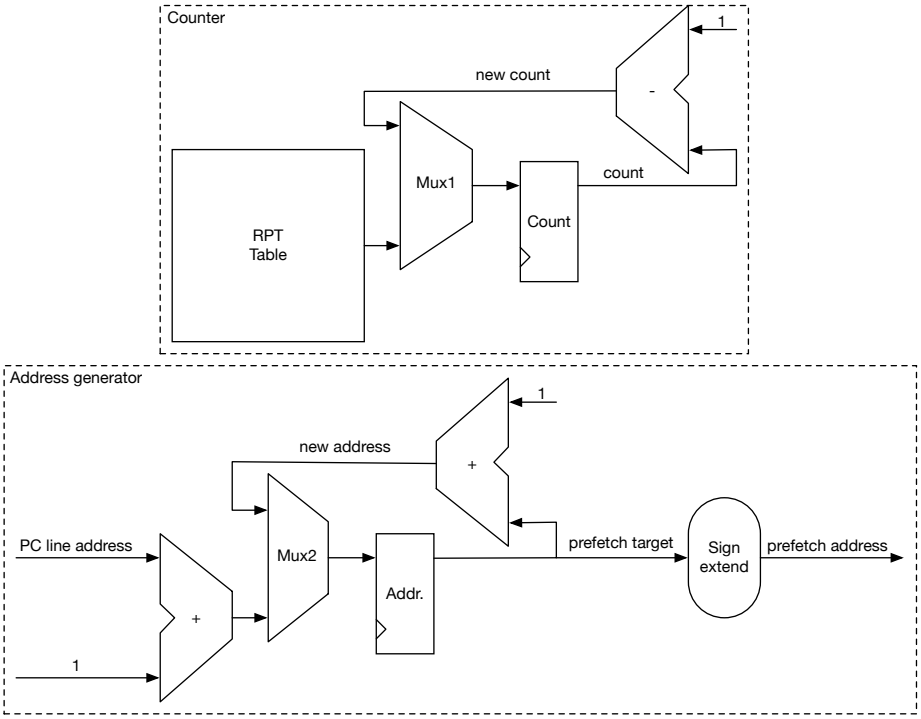


Figure 4.12: *Bulk*-module

is that the value in the *Count* register becomes zero because all planed memory blocks have been brought into the cache, and the second is that the iterations of the loop are finished and the execution continues with the first block outside the loop. The second condition is employed as a prevention mechanism because this type of prefetching is performed independently from the execution and it may happen that the execution exits the loop before bulk-prefetching finishes with its predetermined number of memory blocks. This way, the prefetcher immediately gets synchronized again with the flow of execution and continues to issue prefetch request whenever the execution switches to a new cache line.

4.4 Compact Representation of Code Behavior Information

To be effective, the instruction prefetcher requires information about the behavior of the code in order to guess the target of the prefetch addresses. This information can be generated dynamically during the runtime of the code and used later if the same execution stream is repeated, or it can be derived statically, before the execution starts. The advantage of the second approach is that the knowledge on code behavior is provided ahead of time and enables the prefetcher to guess correctly from the beginning of the execution without requiring any additional time to learn about the behavior of the code. Moreover, the prefetcher starts immediately with prefetching by covering even cold cache misses.

Single-path code experiences a lot of transformation through the conversion process by being converted into a code that has no more input-dependent branch conditions, longer sequential segments and loops that have a constant number of iterations. Thus, the transformation not only eliminates uncertainty on code behavior but also reduces the complexity of the code structure and with that also the amount of information required to describe its control-flow graph. Since single-path code has no dynamic decisions, the whole knowledge about its behavior can be derived statically. It is sufficient to extract information about the control-flow decisions and with that to achieve full coverage on the behavior of the code. Therefore, for guiding the single-path prefetcher through code execution we have decided to use statically derived information that reflects the control-flow graph of the code. In oder to encode statically derived control-flow information in a compact form, we have chosen to use the Reference Prediction Table presented in section 4.3. Each control-flow decision has only one entry in the table, no matter how many times it is executed and the size of the table is proportional to the number of the control-flow decisions in the single-path code.

Generation Process of Reference Prediction Table

The RPT is created by analyzing the execution trace of the single-path code. An execution trace represents a record of instruction addresses as they are referenced by the program counter. The single-path code has the same trace for any input-data, which means that the address trace of a single code iteration is sufficient to extract all the required information.

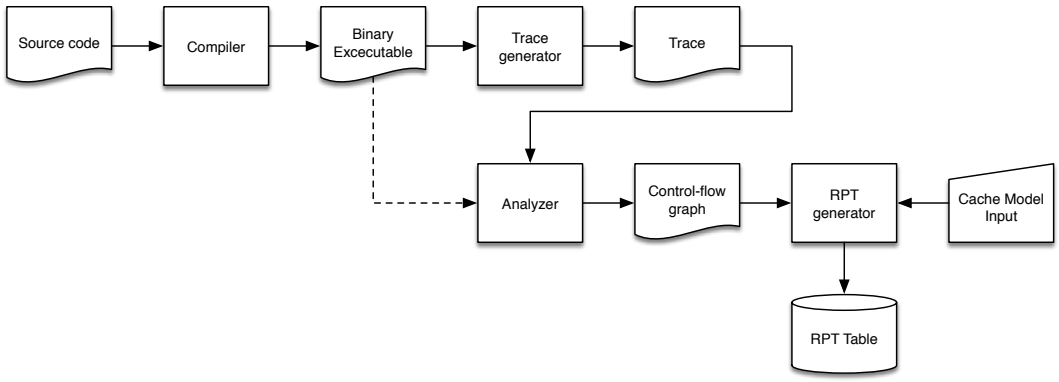


Figure 4.13: RPT generation process

Figure 4.13 shows the whole process of RPT table generation. After the single-path transformation has been performed and the code has been compiled, the executable binary is passed to the trace-generator to record the order of addresses as they are referenced by the processor and then sends the trace outcome to the analyzer. It is important to note that the compiler serializes all if-structures in the code and all converted loops have a constant number of iterations and a single exit point. The job of the analyzer is to use the trace and symbol table to identify the functions in the code, calls in each function, and build the control-flow graph of the function. The pseudo-code for the identification of the functions and calls within the function is shown in Algorithm 4.1. The algorithm uses as input the trace of addresses and the symbol table, while

```
input : address_trace, function_map
output: F, C = <set of functions, set of calls>
1  F ← {};
2  C ← {};
3  call_stack ← [];
4  previous_address ← Null;
5  for ∀ address ∈ address_trace do
6      entry_address , function_name ← function_map(address);
7      if function_name ∉ F then
8          F ← F ∪ (function_name, Function(function_name, entry_address);
9      end
10     func ← F(function_name);
11     func.cfg(address);                          // update the cfg of the function
12     if address = entry_address then
13         if func ∉ call_stack then
14             call_stack.push(previous_address, func);
15         end
16     else if func ≠ call_stack(top) then
17         call_site, callee ← call_stack.pop();
18         if call_site ∉ C then
19             C ← C ∪ (call_site, Call(call_site, address, callee));
20         end
21     end
22     previous_address ← address;
23 end
```

Algorithm 4.1: Trace analysis

the outcome is a set of functions F and a set of calls C. For each address the algorithm reads from F the function instance where the address belongs and add the address to the functions control-flow graph as a new node. If the function does not exist a new function instance is created. The algorithm also generates a *call* instance when the trace switches from one function to the other and places that instance in the set C. Once this is finished, the analyzer starts with the identification of the back edges in each function. The algorithm for recording the back edges in the function is shown in Algorithm 4.2. The input of the algorithm is the set of functions, while the output is a set of back edges for each function. The algorithm identifies the back edges by traversing through all nodes of the functions control-flow graph. If the node v is visited for the first time and its successor w has already been visited then the edge (v, w) is considered as a back edge and is added to the set. The next step is the building of the tree of loops for each function. The algorithm for the loop tree construction is shown in Alorithm 4.3. The loop tree is used for determining the depth of each loop. For each function, the algorithm creates a virtual back edge and considers that as a root of the tree.

The information produced from the analyzer are then sent to the RPT generator. The RPT generator uses this information to identify the position of the control-flow instructions in the code, as well as the type of code structure which they create. The process of table generation is shown in Agorithm 4.4. For each control-flow change in the code, the RPT generator generates

```
input  : F = <set of functions>
output: back_edges
1  for ∀ function ∈ F do
2      back_edges ← {};
3      visited_node ← {};
4      finished_node ← {};
5      stack.push(function_entry_address);
6      while length(stack) > 0 do
7          v ← stack.pop();
8          if v ∈ visited_node then
9              finished_node ← finished_node ∪ v;
10             continue
11         end
12         visited_node ← visited_node ∪ v;
13         stack.push(v);
14         for ∀ w ∈ cfg(v) do                    // w is successor of v in cfg
15             if w ∈ visited_node and w ∉ finished_node then
16                 back_edges ← back_edges ∪ edge(v, w);
17             end
18             if w ∉ visited_node then
19                 stack.push(w);
20             end
21         end
22     end
23 end
```

Algorithm 4.2: Identify back edges in each function

a unique RPT entry. At the beginning, the algorithm fills the table with loop, call and return entries of each function. For the classification of the loops, the RPT generator needs to know the size of the cache in order to determine if the loop is smaller or larger than the cache size. This information is provided from the Cache Model Input. Loop size includes not only the size of the actual loop but also the size of the code segments that are called from within the loop. If the loop fits into the cache then an entry for bulk prefetching will be generated, otherwise the entry will be for a normal loop. The Cache Model Input informs the RPT generator about the associativity of the cache as well. This information is important when the RPT generator determines the number of memory blocks that will be prefetched with bulk prefetching. For instance, if the cache is direct-mapped and the code has a loop which consists of calls to other functions and the size of the loop, including the size of the called functions from that loop is smaller than the size of the cache, the RPT generator will still not generate a bulk-prefetching entry if the memory blocks of the loop are in cache conflict with memory blocks of those functions but instead will classify the entry as normal loop. However, this rule is not effective if the cache is organized as full associative. The information on associativity enables the RPT generator to generate entries that will prevent the temporal properties of the cache. The next step of the algorithm is to sort all RPT entries by their trigger address. At the end, the data in the $Next$ column for loop and call type of entries is calculated.

```
input  : backedges
output: loop_tree
root ← Loop(entry_address, exit_address) ;                    // virtual loop
last ← root;
for ∀ edge ∈ backedges do
    previous ← last;
    new ← Loop(edge);
    while True do
        if new ∈ previous then                                 // check if loop is nested
            new.parent ← previous;
            previous.children ← previous.children ∪ new;
            break
        end
        previous ← previous.parent
    end
    last ← new
end
loop_tree ← root
```

Algorithm 4.3: Building loop tree

```
input  : cache_size, cache_line_size
input  : F, C = <set of function, set of calls>
output: RPT = <set of table entries>
RPT ← {};
for ∀ function ∈ F do
    rpt_group ← {};
    for ∀ loop ∈ function do
        if loop_size ⩽ cache_size and ¬tag_conflict(loop) then
            rpt_group ← Bulk_entry(loop);
        else
            rpt_group ← Loop_entry(loop);
        end
    end
    for ∀ call ∈ function do
        rpt_group ← Call_entry(call);
    end
    rpt_group ← Return_entry(function.last_address);
    sort_by_address(rpt_group);
    RPT ← RPT ∪ rpt_group;
end
for ∀ entry ∈ RPT do
    next_index(entry);
end
```

Algorithm 4.4: RPT generation

4.5 Chapter Summary

To summarize, a prefetcher can be predictable and effective if it successfully answers the questions of what to prefetch, when to prefetch and where to place the prefetched blocks. The static behavior of single-path code allows extracting a-priory all required information on code behavior, which is succinctly summarized in the RPT and later used for guiding the prefetcher in the time and value domain in order to eliminate any possibility for speculative behavior. Using this advantage, the proposed single-path code prefetcher achieves full accuracy on prefetch target anticipation as well as on time when the request should be issued. The constraint on the number of free bus cycles limits the improvement that can be achieved with prefetching.

The architecture of the prefetcher is designed as a set of modules whose behavior is coordinated by a controller. Such an approach allows the development of the prefetcher to be scalable and gives the possibility to make future changes on any of the modules without affecting the properties of the others.

CHAPTER 5

Time-predictable Memory Hierarchy

In this chapter we introduce the architecture of the memory hierarchy for a system that runs single-path code. The chapter starts by enumerating the issues that the conventional memory hierarchy faces to achieve a stable execution of the single-path code. Next, the chapter continues with a description about modifications that should be implemented in on-chip and off-chip memories in order to provide stable execution and performance improvement when single-path code is executed. This chapter also describes the integration of the prefetcher as part of the memory hierarchy as well as the types of filtering that are implemented in order to maximize the benefits of prefetching.

5.1 Memory Hierarchy for Single-path Code

An embedded system that runs single-path code must consist of a memory hierarchy whose timing is stable, predictable and has short access latency. Otherwise the system will loose all the benefits gained from the single-path transformation. This can be provided only if the timing of instructions is repeatable for any iteration of the code in all layers of the memory hierarchy.

The property of single-path code to have a single trace of execution forces the instruction path of the memory hierarchy to always run through the same order of states, which means that the timing of instructions becomes repeatable through any iteration of the code. However, this is not true for a system with a conventional memory hierarchy since not all of its components provide stable timing. First, the single-path transformation does not perform any changes in the data part of the code. Although all *load* and *store* instructions are executed, the timing of instructions will vary through different iterations of the code due to the presence of the conventional data cache. Second, the interference that occurs between instruction and data accesses on the shared main memory affects the timing of instructions due to unpredictable blocking from data accesses. The effect of the interference between these two paths is also reflected in the stability of the execution time of the whole code. Third, to deal with increased code size, modern embedded systems require most of the time to employ high-capacity off-chip memory like Dynamic

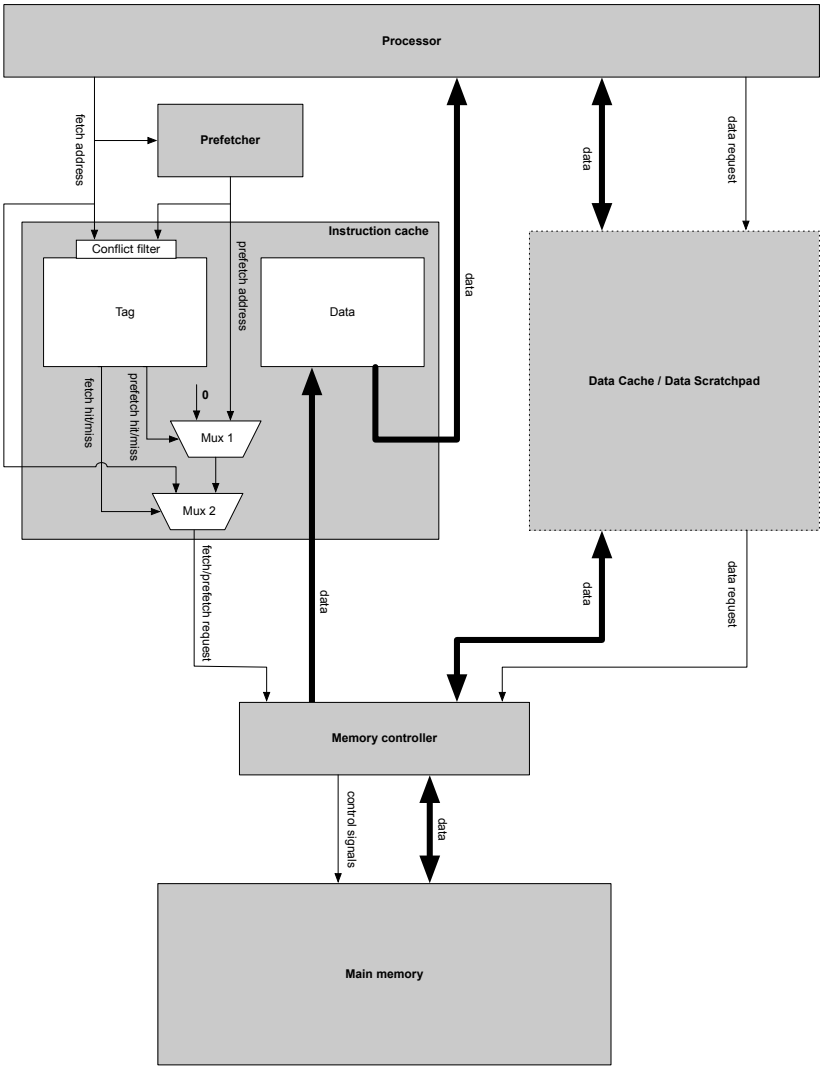


Figure 5.1: Time-predictable memory hierarchy for single-path code

Random Access Memory (DRAM). As a consequence, the timing of the individual instructions is not stable anymore due to the variability that this technology imposes on the timing of each instruction.

Figure 5.1 illustrates the architecture of the memory hierarchy that provides timing stability and performance improvement for systems with single-path code. The hierarchy is composed of the single-path code prefetcher, instruction and data on-chip memory, memory controller and main memory. In the following part of this chapter, we describe the configuration and modification of each of these components in order to achieve stability at the level of instructions and better performance in their execution.

5.2 Organization of the On-chip Memory

The on-chip layer of the memory hierarchy for single-path code is comprised of an instruction cache and a data scratchpad. Most of instruction cache organizations provide stability by design since they repeat the same history of states whenever a single-path code is iterated. On the other hand, to achieve stable timing through the data path of the memory hierarchy we propose two possible solutions. The first approach is the use of a data scratchpad as an on-chip data memory, whose content is loaded before the execution starts. In this case, the timing of each load and store instruction will be the same for each run, since all required data will be available in the scratchpad. Furthermore, the presence of a data scratchpad eliminates the interference that can occur between instruction and data path because the access on the shared main memory for these two type of information will be performed in different phases. When the amount of data is too large to completely fit into the scratchpad, a dynamic software controlled data scratchpad can be used. With this solution the data content is reloaded periodically through special instructions inserted in the code [52]. The timing and the amount of data that the special instructions bring in the data scratchpad should be determined statically. We should note that the data path of the memory hierarchy is not subject of the research in this thesis and the proposed scratchpad solutions are only complements to the instruction path in order to achieve stable timing for the whole execution.

Modified Instruction Cache

The integration of the prefetcher into the memory hierarchy requires a modification of the cache architecture in order to allow regular fetching and prefetching to be performed in parallel without interfering the work of each other. Therefore, the instruction cache is a dual-port on-chip memory that accepts and processes address references issued from both processor and prefetcher. It is based on SRAM technology where each bit cell can be accessed for read and write concurrently form both ports. As the execution runs, each reference that arrives at the cache is firstly compared with the entries of the tag table to search for a possible match. In case of a miss, depending on the source of the request, the cache performs one of two types of actions. On a *processor cache miss*, the cache works like a conventional cache, it stalls the processor and forwards the processor request to the external main memory. On a *prefetch-cache miss*, the cache forwards the request of the prefetcher directly to the main memory without disturbing the operation of the processor. Distinguishing these two types of misses in the cache enables the prefetcher and the processor fetch stage to work in parallel without interfering the execution. Thus, while the fetch stage accesses instructions that are already in the cache, the prefetcher initiates the prefetching of upcoming cache lines.

As can be seen from Figure 5.1, it may happen that both types of address references arrive in the cache at the same time. If both of them result in a cache miss, then only one request can be forwarded to the main memory. In this case, the cache gives priority to the request issued from the processor because these requests stall the execution, but also in this way the cache restricts the prefetcher to read the memory only when the bus is idle. The hardware mechanism for this arbitration is implemented by a cascade of two multiplexers (Mux 1 and Mux 2) where Mux 1

releases the prefetch request only if the prefetch target is not in the cache, while Mux 2 gives priority to fetching before prefetching if both result in a cache miss.

The Impact of the Replacement Policy on Instruction Cache

One of the requirements of single-path code for having stability is that the replacement of cache lines within the instruction cache should be repeatable by generating the same trace of hits and misses for any iteration of the code. For a direct-mapped cache this is achieved by design and the same is valid for any other n-associative cache organization that implements a replacement scheme with well defined rules on the replacement of the cache content. The only exclusion are n-associative caches with *random replacement* or pseudo replacement policies. Another stability issue on cache level is the difference of hit/miss traces between cold and warm cache. The timing difference between the first and the following iteration can be eliminated if the cache content is flashed at the beginning of each iteration, thus always performing with cold cache. To achieve that, a simple hardware is added to the cache which invalidates all *valid* bits of the cache tags whenever execution starts a new iteration and is triggered through a special instruction inserted at the beginning of the code.

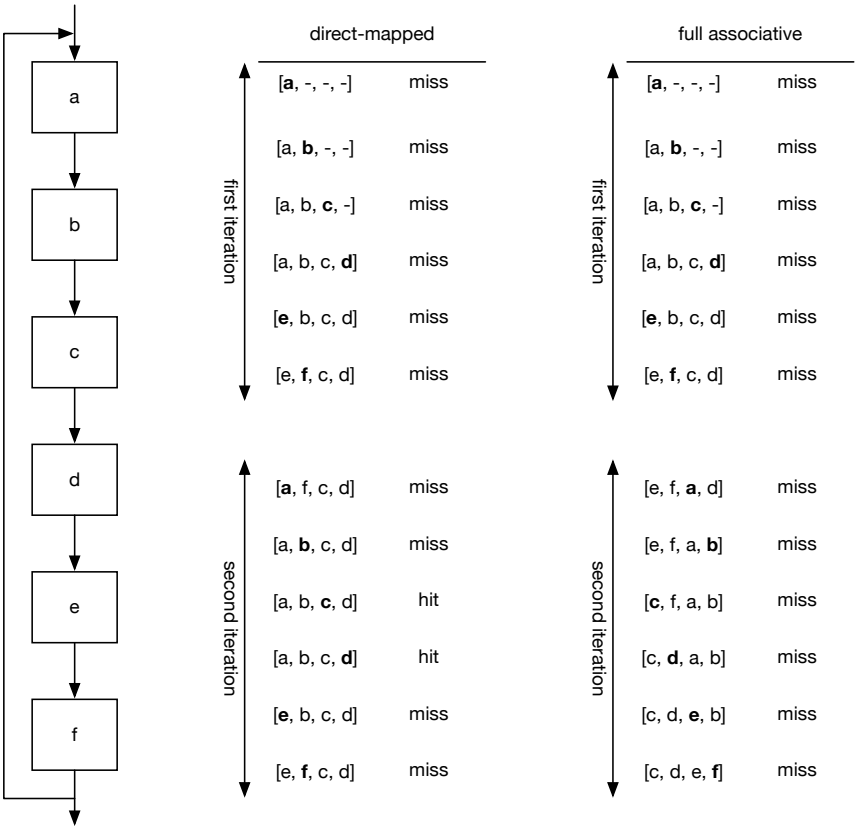


Figure 5.2: The impact of cache replacement policy on single-path loop

Besides predictability, the cache memory has to provide performance as well. Therefore we aim at organizing the cache in the way that would benefit from single-path properties. Intuitively, increasing the associativity of the cache means better exploitation of the codes temporal locality, since the selection of cache lines that can be evicted is larger, but this is not alway true for single-path code. The main concept of the single-path transformation is to convert all input-dependent alternatives of the code into a sequential segment. Thus, loops with sequential body are very often cases of single-path code. For such loops the direct-mapped cache can achieve higher hit rate than a fully associative cache when the loop is larger than the cache size. For example, Figure 5.2 shows a control-flow graph of a loop consists of six memory blocks (a,b,c,d,e,f) and two instruction caches with capacity of four cache line, where the first one is organized as direct-mapped and the second is fully associative cache with LRU replacement policy. During the first iteration all cache accesses will result in cache miss due to the cold cache state, but for any other iteration the direct mapped-cache will have two hits and four misses, while with a fully associative cache all accesses will result in cache miss. In such cases, fully associative cache can achieve the same performance as direct-mapped if a dynamic cache lock mechanism is implemented that will lock part of the cache lines with loop content and release them after the last loop iteration has been executed [19].

5.3 Organization of the Main Memory

The memory controller needs to arbitrate between the instruction and data path when they share the same bus to access the main memory. If the on-chip data memory is a scratchpad that is loaded before the execution starts, then the memory controller can employ any arbitration policy because the access to the main memory between instruction and data accesses will never interfere with each other. For the second solution from Section 5.2, when the data path of the memory hierarchy has a dynamic software controlled data scratchpad that is periodically loaded, the arbitration policy should be priority-based where the data accesses have higher priority than the instructions. Such a policy prevents the data cache to stall the processor for longer time, especially when the bus gets busy from the prefetcher which is in bulk prefetching mode.

The last layer of the hierarchy is the main memory which can be of SRAM or DRAM technology. The advantage of using SRAM is that this type of technology provides fast and stable timing for each access, no matter if the type of access is read or write. However, SRAM solutions are expensive and can be employed as main memory only when the system runs applications of small size. On the other hand, DRAM offers cheap storage with fast access but the duration of each access is dictated from the organization of DRAM itself. A DRAM bank with open-page policy serves a request immediately if the previous request was targeting the same row as the actual one and not so fast if the target of the previous access was on different row [78]. Knowing that single-path code has long sequential segments makes the employment of this solution much more beneficial than the closed-page policy. Timing variety with open-page policy occurs due to precharge command, which is released only when a bank row needs to be closed in order to bring into the sense amplifier the following row targeted from the actual request. Another reason for time variety is DRAM rank switching, because ranks share the same data bus and when targets of the current and the previous access are on different ranks then a penalty in form of an

idle cycle needs to be considered [31]. If the previous access was write and the actual is read, or vice versa, then the data bus requires a few clock cycles for bus turnaround as well [2]. All these time delays due to the dependency of the actual access from the state of the previous one, make the access time of the request variable. However, timing variety imposed from all these parameters has no impact on timing stability of the single-path code because the sequence of DRAM commands that are sent to the DRAM device are always the same for each code iteration. The only problem that emerges from the employment of DRAM technology in a system with single-path code is the occurrence of jitter due to DRAM refreshing. DRAM requires to be refreshed periodically and this process is not synchronized with DRAM accesses. If the memory controller tries to access the DRAM device when refreshing is in progress the request will be stalled until the refreshing operation is finished. Thus, the asynchronous interferences that this process imposes on the execution of the single-path code generates execution-time jitter. One solution that has been proposed for eliminating this issue is to synchronize the start of execution with the refreshing process by delaying the beginning of the execution until the first refresh operation takes place [78]. To implement this, a signal from the memory controller that triggers refreshing should be send to the processor as well. However the length of delay is still not constant. Another solution to eliminate the refreshing jitter is to disable auto-refreshing and employ software-assisted refreshing instead [11]. This can be implemented through a new task that will do burst refreshing by reading each row of memory banks. The other option is to reprogram the refresh interval of the DRAM control register to perform burst refreshing of all rows of the DRAM memory, where its activation/deactivation is controlled by a separate software task. Our proposal is to use a special instruction that would trigger refreshing at the beginning of each iteration no matter when the previous refresh has been performed. In this way the period between two refreshes will be reset and synchronized with the beginning of the iterations. Since the refresh will stall the beginning of each iteration, it is recommended the refresh period to be of type distributed in order to minimize the delay from refreshing on applications with short execution time. By doing so, the refresh process becomes repeatable, since its triggering moment becomes repeatable for each iterations of the code.

5.4 Prefetch Filtering

The prefetcher is located between the processor and instruction cache and has one input for monitoring the fetch requests issued from the processor and one output for releasing the prefetch request. The output of the prefetcher can be connected directly to the main memory or it can reach the memory through the cache. The advantage of the first approach is that the cache remains unchanged and no lookup search is triggered in the cache since all prefetch requests are sent directly to the main memory. But if the prefetch target is a memory block that is already in the cache then the whole prefetching process becomes redundant. In such a case the prefetcher will degrade the system performance by wasting the bus cycles with useless memory traffic. The second solution treats each prefetch request as if it were a normal fetch request by sending it firstly to the cache. This technique is called *prefetch probe filtering*, since firstly it checks if the potential prefetch target is already in the cache or not and then releases it to the memory [95]. In case of a hit the filter will just discard the request and wait for the next one, otherwise the

request will be forwarded to the main memory. The output of the single-path code prefetcher is connected to the cache as illustrated in Figure 5.1.

Usually, memory hierarchies that are augmented with a prefetch unit have a cache organized as n-way-associative in order to avoid any possible conflict between fetching and prefetching processes if both targets are mapped on the same cache location. But this type of conflict can easily occur if the cache has a direct-mapped organization. For example, if the fetch stage of the processor fetches instructions for the n-th line of cache and in the meantime the prefetched block is mapped on the same cache line then prefetching will destroy the content of that cache line by generating additional cache misses. In such cases the prefetcher only degrades system performance. To eliminate this problem we propose a *conflict filter* which monitors the addresses of prefetch requests and compares them with the address of the actual fetch. If the address fragment that identifies the cache line is the same then the filter will drop the prefetch request before it enters the cache. This way, the filter insures that the prefetcher does not evict the cache line that holds the currently executing instruction. Its position in memory hierarchy is shown in Figure 5.1.

5.5 Chapter Summary

To summarize, a real-time system that runs single-path code can become predictable only when the memory hierarchy of the system supports predictability on all of its layers. This means, the timing of instructions should be repeatable whenever the same instruction is executed through different iterations. Therefore, in this section we have defined a memory hierarchy that would satisfy these requirements. Apart form predictability, the proposed memory design provides performance as well by employing implementations that are more suitable for single-path code without affecting its predictable properties. To prevent any form of redundant memory traffic, prefetch filtering has been proposed to discard prefetch requests for memory blocks that are already in the cache.

CHAPTER 6

Implementation and Evaluation

This chapter presents the implementation and evaluation of the time predictable memory hierarchy for single-path code. The chapter begins by introducing the T-CREST platform and Patmos processor as a framework that supports the execution of single-path code. Next, the chapter describes the implementation process of the single-path code prefetcher and the instruction cache as part of the memory hierarchy. The chapter ends by evaluating the performance of the memory hierarchy through running a number of experimental executions and discussing the results.

6.1 T-CREST platform and Patmos Processor

The intention of the proposed memory hierarchy is to improve the execution time performance of the single-path code. To achieve that, the memory hierarchy is required to be build upon a system consisting of a time-predictable processor that supports predicated execution. To the best of our knowledge, the T-CREST platform [99] with Patmos processor [101] is the only platform that supports predicated instructions and has a compiler that transforms conventional code into single-path code.

The T-CREST platform is a real-time system platform designed to make the WCET analysis simple and the execution of the worst-case faster. The platform covers technologies at processor level, on-chip communication, compiler with WCET optimization and single-path conversion and WCET analysis tools. All hardware components of the platform are time-predictable and allow WCET analysis. The processor node includes the Patmos processor, special instruction and data cache memory, and local scratchpad memories for instructions and data. Patmos is a 32 bit dual-issue processor with instruction set of RISC-style. Both issues of the processor share between them a register file with 32 registers. Patmos can be configured to have two-pipeline units for high performance or only a single one to save hardware resources. The architecture of the processor is a Very Long Instruction Word (VLIW) 5-stage pipeline with incorporated forwarding paths. The stream of instructions is statically scheduled by the compiler. All instruction delays are well defined and visible at ISA level. All instructions are predicated and take at

most three register operands. The execution time of each instruction is constant, independent from the value of the predicate. Patmos can supports bundles that are 32 or 64 bits wide. The local memory of Patmos is a set of caches consisting of a standard instruction cache or method cache, data cache, stack cache as well as instruction and data scratchpad. The last two types of scratchpads can be used as substitutes for instruction and data cache or in addition to them. Splitting of the local memories has the purpose of simplifying the model of caches and with that the WCET analysis.

The compiler used in T-CREST is an extension of the LLVM compiler framework [60], which integrates the option for an automatic transformation of the conventional code into single-path code. When the single-path option is selected, the compiler serializes all if-statements, transforms all loops into loops with constant number of iterations and maintains the value of predicates through function calls. In addition, each loop is generated with a single exit edge at the end of the loop body. The compiler performs transformation in the backend and operates on the control flow graph representation rather than on the source code. The only requirement that the compiler has for single-path transformation is that the loop bounds for each loop have to be provided in the source code. However, the current compiler version does not make distinction between input-dependent and input-independent branches by transforming all of them into a sequential code. Thus, the new generated code has no execution alternatives that are input-independent.

6.2 Implementation of the single-path code prefetcher

For the implementation of the single-path code prefetcher, we chose to use *Chisel* [8], which is a hardware-design language embedded in the *Scala* programing language [77]. The intention of *Chisel* language is to be the platform that offers the possibility to design low-level hardware blocks by using modern programing language features. Compilation of the source code can generate a fast cycle-accurate simulator or Verilog code suitable to be synthesized for an FPGA or ASIC solution.

The single-path code prefetcher is implemented as a state machine, as illustrated in Figure 6.1. It consists of combinatorial logic for the calculation of the output (prefetch address), combinatorial logic for the calculation of the next *Index* value and the values of the next state, sequential logic for saving the state, the RPT for reading the parameters and the *Index* register to point at the actual row in the RPT. As can be seen from Figure 6.1, the output of the prefetcher depends not only on the present state but also on the input, which makes the state-machine to be of Mealy type.

The pseudo-code for implementation of the prefetcher is shown in Algorithm 6.1. The first two lines define the states of the state machine and dictate state *A* as initial one. Based on Section 4.5, the model of the state machine should consist of three states as the one shown in Figure 4.6, but in the current version state *B* has been not implemented because the compiler prototype used for our evaluation serializes all *if*-branches in the code. Thus, for transformed code that has no if-structure the state machine will never transfer to state *B*. Lines 3-36 of the algorithm cover the hardware design for state *A*, lines 37-43 for state *C*. The hardware generated from the first **when** condition within state *A* checks if the execution has switched to a new cache

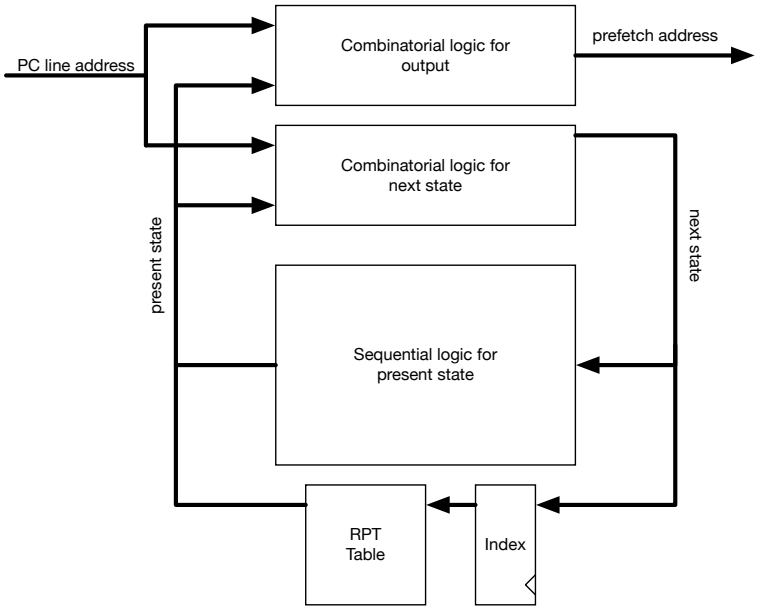


Figure 6.1: Representation of the single-path code prefetcher as state-machine

line. When this condition is fulfilled, the prefetcher continues with the selection of the prefetch module based on conditions defined in the other **when** cases. Lines 6-7 define the hardware for the sequential module, lines 8-11 for the call module, lines 12-15 for the return module and lines 25-36 for the loop module. As can be seen, the loop module has three other **when** conditions within, in order to define the hardware for all three modes of the loop as described in Section 4.11. Lines 16-24 of state *A* belong to *bulk* module. The hardware defined in lines 16-21 is used only to set the parameters for bulk prefetching, while the transit of the state machine from state *A* to state *C* is done through the hardware in lines 22-24. The transition is performed on the next prefetch trigger in order to ensure that the actual prefetching process is finished and there will be no conflict with the request issued from bulk prefetching. In state *C* prefetcher triggering cannot be synchronized with cache line switching anymore because the execution now runs through a loop which is already in the cache. Therefore, in this state the triggering is done only when *tag_signal* is high, which is defined in line 35. *tag_signal* writes the tag of the memory block brought in the cache and goes high only when the last bit of the memory block arrives in the cache. Thus, using the same signal the prefetcher gets informed that the requested prefetch block is already in the cache and a new prefetch request can be issued. The target of the next prefetch address is calculated through the hardware generated from lines 38-41. The condition to exit burst prefetching is given in line 42. The state machine switches to state *A* if the number of memory blocks dictated by the RPT have been all prefetched or the execution has exited the loop and continues through the next cache line. To simplify the presentation of the algorithm, we have not presented implementation details that deal with rare conditions like when the loop size is smaller or equal to the size of the cache line or when the prefetch target and the destination are on the same cache line.

```
   input  : RPT, PC_line_address
   output: prefetch_address
 1 state: A, C;
 2 state ⟵ A;
 3 when (state == A)
 4     when (PC_line_address ≠ cache_line_reg)
 5         cache_line_reg ← PC_line_address ;
 6         when (no_match_with_RPT and ¬change_state)
 7             prefetch_address ← PC_line_address + 1;
 8         when (match_with_RPT and ¬change_state and type == call)
 9             prefetch_address ← RPT;
10             index ← RPT;
11             push;
12         when (match_with_RPT and ¬change_state and type == return)
13             prefetch_address ← stack;
14             index ← stack;
15             pull;
16         when (match_with_RPT and ¬change_state and type == burst)
17             change_state ← true;
18             index ← index + 1;
19             count_reg ← RPT;
20             address_reg ← PC_line_address;
21             trigger_reg ← PC_line_address + 1;
22         when (change_state)
23             state ← C;
24             change_state ← false;
25         when (match_with_RPT and ¬change_state and type == loop)
26             when (first_loop_iteration)
27                 prefetch_address ← RPT;
28                 index ← RPT;
29                 iteration_reg ← RPT;
30             when (next_loop_iteration)
31                 prefetch_address ← RPT;
32                 index ← RPT;
33                 iteration_reg ← iteration_reg − 1;
34             when (last_loop_iteration)
35                 prefetch_address ← PC_line_address + 1;
36                 index ← index + 1;
37 when (state == C)
38     when (tag_signal and count_reg ≠ 0)
39         prefetch_address ← address_reg;
40         address_reg ← address_reg + 1;
41         count_reg ← count_reg − 1;
42     when (count_reg == 0 or trigger_reg == PC_line_address)
43         state ← A;
```

Algorithm 6.1: Algorithm of single-path code prefetcher

6.3 Implementation of the cache

The cache memory is implemented as a composition of three components: front-end, back-end and storage. The front-end provides an interface to the processor fetch stage and prefetcher, by accepting their requests. Next, the front-end searches through the cache and checks if these references are present or not and based on that declares cache hit or miss for each reference no matter if it is issued from fetch stage or prefetcher. Patmos is a dual-issue VLIW processor, which fetch bundle of two instruction on each fetch operation. Since *Chisel* does not support true dual-port memory, we were forced to split the tag table of the cache in two parts, defined as table with *even addresses* and table with *odd addresses*, in order to allow parallel search through them. Furthermore, the cache allows matching for prefetch requests to be done in parallel with fetch operation/request. Therefore, we have two more replica of *even* and *odd* tag tables, used only by the prefetcher. The hardware that performs these operations is implemented through Algorithm 6.2. The hardware has as input the *even* and *odd* fetch addresses issued from the processor and the prefetch address issued from the prefetcher, while outputs are the hit/miss signals for all three issued requests and the address reference that should be fetched from the main memory. The first four lines of the algorithm define the default values of the output signals, because the whole component is built on combinatorial logic. Lines 5, 7 and 10 of the algorithm generate the hardware to do matching between tag bits of the addresses issued from the processor and prefetcher with tag bits of the tag table as well as checking the valid bit of that tag table if it is *true* or *false*. Since cache is organized as direct-mapped, only one location from the tag table is required to be compared and the position of that entry is extracted from the index bits of the reference addresses. *Chisel* converts *when-elsewhen* structures into a cascade of multiplexers, where the first condition has the highest priority, then the second one and so on. Such an order gives priority firstly to *even* and *odd* fetch references and then to the prefetching as they are defined in lines 4, 9 and 13. Lines 6, 8 and 12 are signals about the outcome of the search in the tag table. When any of them becomes *false* a fetch request is forwarded to the main memory, but only *hit_even* and *hit_odd* signals can stall the processor, thus enabling fetching and prefetching to be performed in parallel. The condition in line 11 is optional for direct-mapped cache because it compares the index bits of fetch and prefetch addresses in order to avoid possible cache conflicts between these two processes. In case the prefetch address is mapped to be placed in a cache line that is actually used by the fetch stage, the hardware synthesized from line 11 will drop the prefetch request (conflict filter).

The back-end component of the cache deals with requests released from the front-end and forwards them to the main memory. When the requested memory block arrives, the job of the back-end is also to place the block into the determined cache line. The hardware implementation of this component is based on the state-machine shown in Figure 6.2. As can be seen, the state-machine can be in state *init*, *idle*, *wait* and *tran*, where the first state *init* is also the initial one. The Patmos processor can read instructions from cache or scratchpad memory. Thus, the first state is to determine if the cache is the component from where the read of instructions should be done. Once the cache has been selected, the state machine transfers to the state *idle* and waits for the fetch request. If any of the *hit* signals from front-end component switches to *false*, the back-end immediately reads the fetch address and forwards it to the main memory. If the

input : address_even, address_odd, prefetch_address
output: fetch_address, hit_even, hit_odd, hit_pref

```
1  hit_even ← True ;
2  hit_odd ← True ;
3  hit_pref ← True ;
4  fetch_address ← address_even ;
5  when ((tag_cache_even ≠ tag_address_even) || (¬valid_even))
6      hit_even ← False ;
7  .elsewhen ((tag_cache_odd ≠ tag_address_odd) || (¬valid_odd))
8      hit_odd ← False ;
9      fetch_address ← address_odd ;
10 .elsewhen ((tag_cache_prefetch ≠ tag_address_prefetch) || (¬valid_prefetch))
11     when ((index_even ≠ index_prefetch) && (index_odd ≠ index_prefetch))
12         hit_prefetch ← False ;
13         fetch_address ← prefetch_address ;
```

Algorithm 6.2: Algorithm for cache and prefetch hit\miss detection

request is acknowledged by the main memory then the state machine transits to state *tran*. If the memory is occupied from some other master component in the system, the response to the back-end will be negative. In that case the state machine switches to state *wait* and waits until the main memory becomes free. Once this happens, the back-end receives an acknowledge and the state machine transits to state *tran*. The state machine stays in state *tran* until all instructions of the actual cache line have been brought into the cache. When the transfer has finished, the state machine transits back to the state *idle* and waits for the next request. It is important to note that the tag address of each cache line is written at the moment when the last bit of the cache line has been brought into the cache.

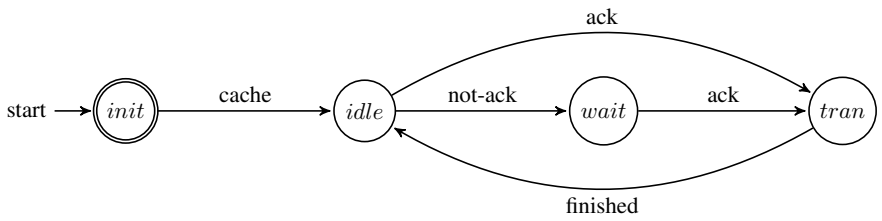


Figure 6.2: State-machine diagram of the cache back-end

The storage is the last component of the cache consisting of *tag* memories and *instruction* memories, which stores the memory blocks brought from the main memory. The number of *tag* memories that are generated depends on the organization of the cache. As we mentioned, direct-mapped cache has four *tag* memories (*even*, *odd*, *prefetch_even* and *prefetch_odd*) and two instruction memories (*instruction_even* and *instruction_odd*) due to limitations of the *Chisel* language and this number increases *n* times for n-way set-associative caches but at the same time the number of lines on each memory is reduced *n* times as well.

6.4 Evaluation Platform

In order to evaluate the performance of the new memory hierarchy, we have synthesized the prototype design of the system and uploaded it on an DE2-115 FPGA board consisting of an Altera Cyclone IV EP4CE115 device, which has 114480 logical elements (LEs) and 3888 Kbits of embedded memory. The synthesized prototype includes the Patmos processor, single-path code prefetcher, instruction and data cache memory, Open Core Protocol (OCP) for on-chip communication and SRAM memory controller. The system employs as main memory the 2MB SRAM chip located on board. This memory results in 21 clock cycles when a burst of 4 32-bit instructions are brought into the cache.

We have synthesized a set of systems with four different types of memory organizations. The first group employs conventional cache with direct-mapped organization, the second group uses 2-way set-associative cache with LRU replacement policy, the third group uses direct-mapped cache augmented with single-path code prefetcher and the last group uses a 2-way set-associative cache with single-path code prefetcher as well. For all four groups the employed instruction cache was of size 1KB, 2KB, 4KB, 8KB, 16KB and 32KB. In all cases the data cache was organized as write-through direct-mapped cache with a size of 4KB. The single-path code prefetcher alone consumes 2014 LE, 577 registers and has a *Fmax* of 84.56 MHz. The *Fmax* of the prefetcher is measured for the benchmark with the largest RPT from all evaluated benchmarks.

6.5 Evaluation

For the evaluation of the memory hierarchy, we use the collection of Mälardalen WCET benchmarks [45]. This set of programs is designed to evaluate and compare different types of WCET analysis tools and methods. The set consists of 36 programs, which are composed of different program constructs like nested loops, input dependent loops, switch cases, nested if-statement, bit manipulation, float point calculation, array and matrix calculation in order to test and evaluate the WCET for a wider range of program properties. The advantage of Mälardalen WCET benchmarks is that the programs contain their own inputs, and the bounds of the loops are given. The disadvantage of using this set is that the input vector that would trigger the WCET of the program is not given. Thus, the real WCET of the code derived through measurement and the WCET of the same code that transformed to single-path can not be compared to see the timing overhead that is generated from the transformation. From the whole set of benchmarks we excluded the *fac*, *recursion*, *matmul*, *st* and *duff*, as our compiler prototype does not handle recursion and calls to external libraries.

We had to instrument the code of each benchmark in order to perform end-to-end measurements of the execution time. To avoid the transformation of the instrumented code, we adapted all benchmarks by defining a new *main* function which contains the instrumented code and a function call to call the original main function of the benchmark. Thus, the compiler transformation target is only the original main function. In addition, we also have added an attribute to the benchmark's main function in order to avoid inlining of functions by compiler.

We compiled the set of selected Mälardalen WCET benchmarks with -O2 optimization using

the T-CREST cross-compiler under the Ubuntu operating system. The executable binary was then run on *pasim*, which is the simulator of Patmos processor, to generate the trace of the program counter (PC). Next, the *analyzer* combined with the RPT-generator used this trace information to generate the RPTs. In our implementation we synthesize each RPT as a ROM memory.

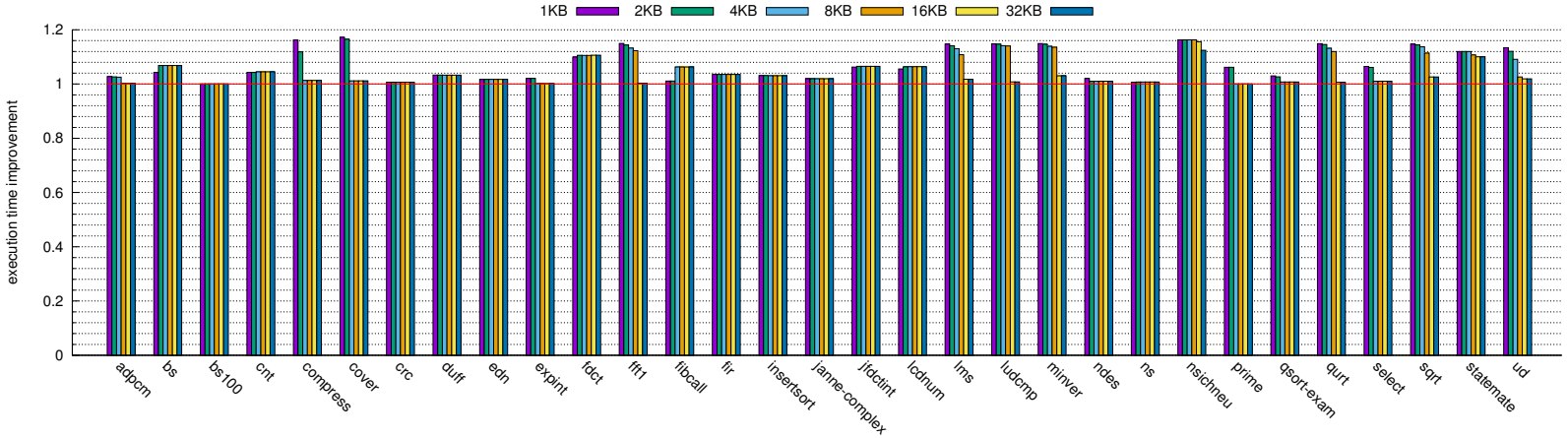
Evaluation results

We now present the results from our evaluation. We have evaluated the impact of the single-path code prefetcher on the execution time of the Mälardalen benchmarks for different memory configurations. We started by evaluating the predictability of the prefetcher. Next, we examined the impact that diverse cache organizations have on the performance of the prefetcher by changing the size of the cache, the cache line length and the associativity. We also quantify the impact of the data cache on the performance of the prefetcher. Finally, we compare the performance of the prefetcher and instruction cache with other forms of on-chip memory organization.

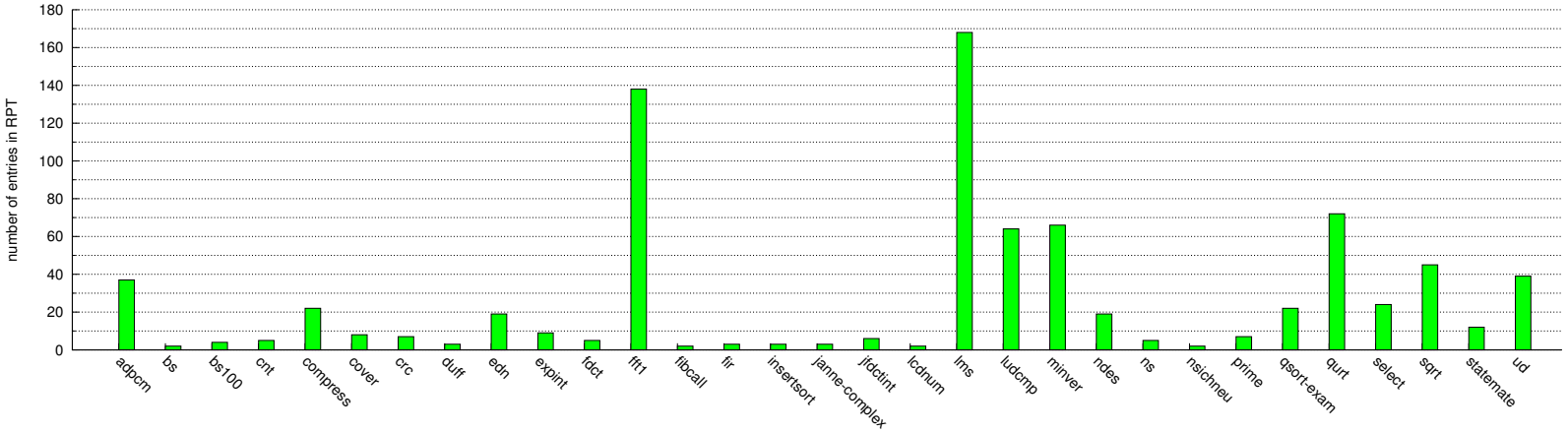
Figure 6.3a presents normalized values of the execution time to show the timing improvement that can be achieved when a memory hierarchy with a prefetcher is used. All benchmarks were run on two platforms that were identical, except for the prefetcher which was used only in one setup. In both cases the cache is organized as direct-mapped, with cache lines of four instructions where each instruction is 32 bits and cache size in the range of 1KB to 32KB. The horizontal red line, in the figure, positioned at 1 represents the threshold for the interpretation of the results. Any bar above the threshold means positive performance improvement and anything below means reduction. From the figure we observe that for all cases the bars are above the threshold which means that the positive improvement has been achieved in all cases. This is due to time-predictable behavior of the prefetcher which prevents any possibility for cache pollution or useless memory traffic that could slow down the execution.

However, even though the prefetcher guarantees improvement for any workload, the degree of improvement is not the same. If we observe the impact of the cache size, we see that for most of the benchmarks (*adpcm*, *compress*, *cover*, *fft1*, *lms*, *ludcmp*, *minver*, *prime*, *qurt*, *select*, *sqrt* and *ud*) the prefetcher is more successful for configuration with small cache size. This is due to the instruction footprint of the benchmarks. Larger cache size means greater exploitation of codes temporal locality and fewer conflict cache misses. Therefore, in cases with large cache the single-path code prefetcher is mainly effective only with cold cache misses. The scale of improvement also depends on the structure of the single-path code. For instance, codes with small loops will initiate more bulk prefetching. From the observed results of all benchmarks in Figure 6.3a, the range of improvement is from 0.01% for *bs100* to 17.1% for *cover* benchmark. The average performance improvement for cache with size of 1KB is 7.3% and this number decreases as the size of the cache is increased by reaching 3% in average for cache with size of 32KB.

Figure 6.3b shows the number of RTP entries that each evaluated benchmark generates. This number is proportionally dependent with control-flow instructions in the code since each of them generates one entry. In general, for Mälardalen benchmarks these tables are not so large. The benchmark with the largest table is *lms* and has 168 entries, while the average number of entries for all benchmarks is 27.



(a) Performance evaluation for different cache size



(b) Number of entries in RPT

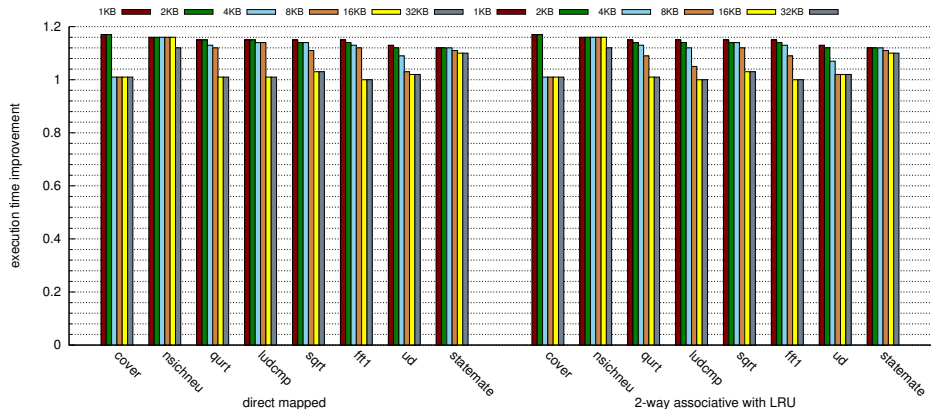
Figure 6.3: Performance evaluation of Mälardalen WCET benchmarks

Figure 6.4 shows the performance of the single-path code prefetcher for different cache organizations. For this figure we selected to show those Mälardalen benchmarks for which the effect of prefetching can mostly be observed. Figure 6.4a shows the results for a cache with line size of 16B, Figure 6.4b for a cache with line size of 32B and Figure 6.4c for a cache with line size of 64B. The bars on the left-hand side of each figure show the results for the caches with direct-mapped organization, while those on the right-hand side show the results for the caches with 2-way set-associative organization and LRU replacement policy. The size of the cache varies in the range from 1KB to 32 KB. Increasing the length of the cache line means longer overlap period between fetching and prefetching process, but this is beneficial for prefetching only when the whole cache line is executed, as demonstrated in Figure 6.4. From the results we observe that increasing the length of the cache line affects positively the prefetch performance only for *cover* benchmark (from 17% to 19%) and *nsichneu* benchmark (from 16% to 18%), while for the rest of the benchmarks increased cache line reduces the prefetch performance. The *cover* and *nsichneu* benchmarks contain larger sequential segments and therefore prefetching in these cases becomes more efficient. On the other hand, the change of cache organization from direct-mapped to 2-way set-associative results in small improvements only for *ludcmp* and *fft1* for cache of size 8KB, which results from cache conflict reduction due to increased associativity. For all other cases, the switching from direct-mapped to 2-way associative is almost irrelevant for the prefetching performance.

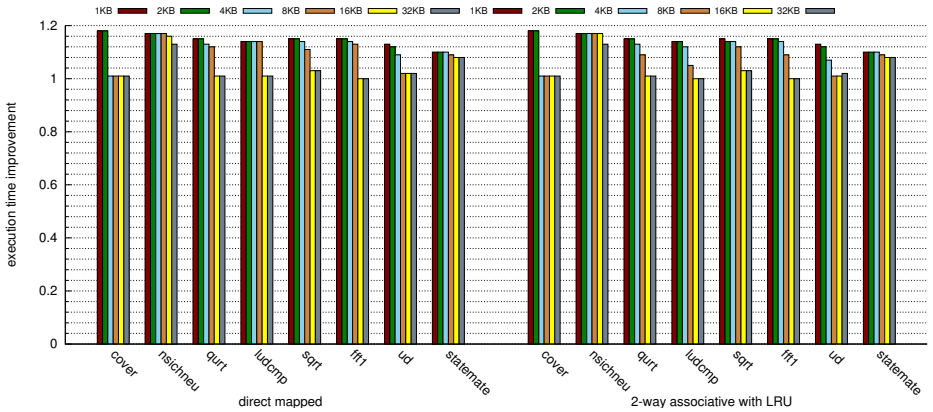
It is important to note that the execution times of the benchmarks are derived through end-to-end measurement. For such an approach, the measured time is a composition of time required to bring instructions into the instruction cache (t_i), time required to bring data into the data cache (t_d) and time to process them (t_p). The presence of prefetcher affects only (t_i), while the other two timing parameters remain unchanged. Therefore, benchmarks with small instruction footprint and large data have poor prefetching performance. The results shown in this way represent relative improvement of the execution time.

To understand the impact of the prefetcher in greater depth, we have executed single-path transformed *binary-search* algorithm on four almost identical systems which differ only in their on-chip memory organization. The first system has a memory consisted of instruction and data cache, the second system additionally includes a single-path code prefetcher, the third system consisted of instruction cache and data scratchpad, while the fourth system has the on-chip memory like the third one plus the prefetcher. Configurations with data scratchpad were uploaded with all required data before the execution started. For evaluation we used an array of size between 10 and 100 elements. Figure 6.5a shows the execution time of the binary search algorithm in all four systems. From the results we observe that as the size of the array increases the execution time gap between systems with data cache and data scratchpad also gets wider. In Figure 6.5b we show execution time improvement that can be achieved with prefetching in systems with data cache and data scratchpad for the same binary search algorithm. From the results we can observe that the presence of a prefetcher in a system with conventional data cache improves the execution time in the range from 8.2% to 10.7%, while for a system with data scratchpad the improvement is in the range from 13.3% to 14.8%, due to elimination of t_d. Another phenomenon that can be observed from the same figure is that as the number of elements increases the timing improvements get lower for both configurations because t_p increases while

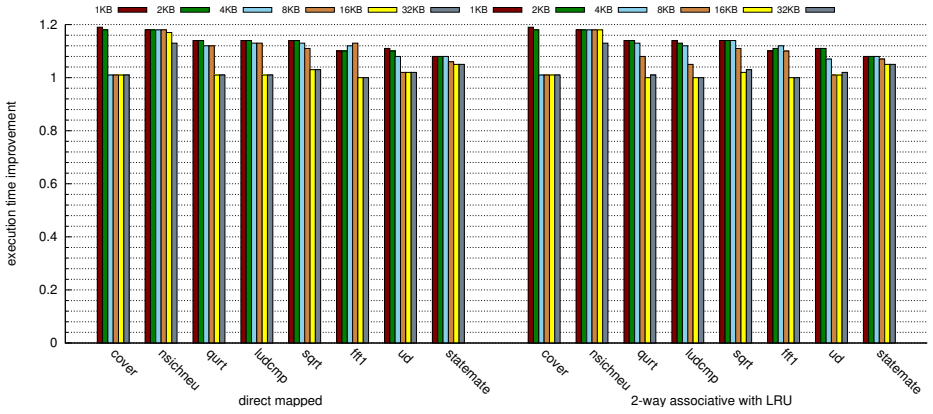
t_i remains constant.



(a) cache with 16B line size

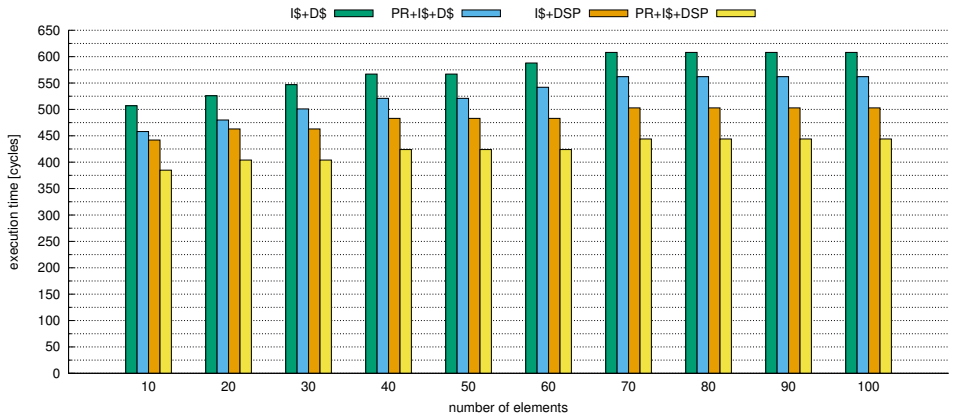


(b) cache with 32B line size

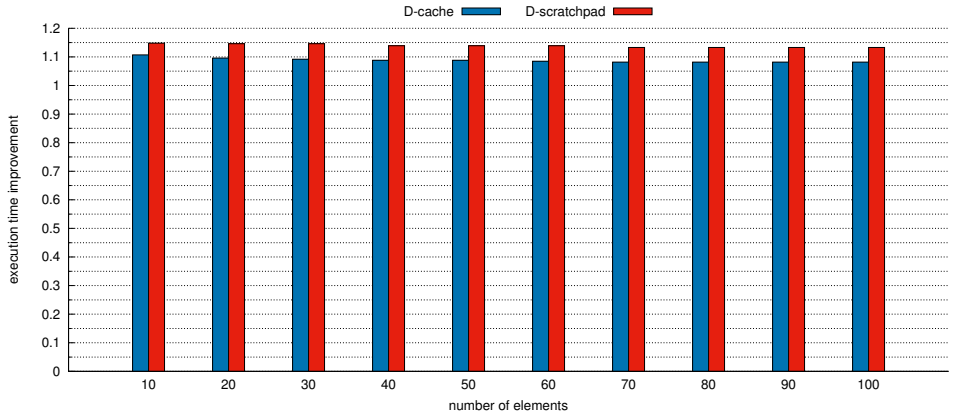


(c) cache with 64B line size

Figure 6.4: Performance evaluation for diverse cache organization



(a) Execution time of binary sort algorithm



(b) Performance of prefetcher for memories with data cache and data scratchpad

Figure 6.5: Execution of binary sort algorithm on system with different memory configurations (I$ - instruction cache, D$ - data cache, PR - prefetcher, DSP - data scratchpad)

6.6 Chapter Summary

For the prototype we used Patmos processor because it supports predicated instruction and has a compiler that generates single-path code. We also implemented the prefetcher and integrated it in the memory hierarchy. The cache has been modified to allow fetching and prefetching to be performed in parallel. For main memory a SRAM chip is used. For the evaluation we executed the Mälardalen WCET benchmarks. From the results it can be observed that the prefetcher behaves in a predictable fashion by improving performance for all benchmarks. The scale of improvement depends on the code structure and the size of the instruction footprint within the code.

CHAPTER 7

Related Work

In this chapter we show related work on techniques for improving the predictability of memory hierarchies. In general, research in this area can be observed going in two directions. On one side, there is ongoing research on building analysis tools for analyzing the timing behavior of conventional memory architectures. On the other side, we see the development of new techniques that are employed in memory hierarchies to improve predictability and simplify the analysis. The chapter starts by giving an overview of state-of-the-art static cache analysis used in systems with conventional cache architectures and then continues with approaches that include the effect of prefetchers in WCET bound estimation. Next, the chapter describes techniques designed to improve predictability of the memory. The chapter ends by showing existing WCET-aware prefetch solutions that improve estimated WCET bounds in a predictable form.

7.1 Static Analysis of Conventional Cache Memory

Predicting the dynamic behavior of the cache is one of the main issues in WCET analysis due to the history dependency of the cache states. To make the analysis tractable, state-of-the-art tools are using an abstract interpretation of cache behavior [76]. They classify memory accesses in *always hit*, *always miss*, or *not classified* if classification for the access cannot be determine and then based on that outcome determine the memory time of each access. In fact, the analysis associates concrete cache states with abstract cache states for each point in a program. For caches organized as n-set associative, the age of cache lines is also modeled. The update of abstract cache states is performed in a way as a concrete cache would have been updated. Since the analysis is performed on a control-flow graph, whenever a join point emerges the states of all incomes are combined. At this point the analysis uses *must* or *may* merge functions depending on the type of classification that is performed. The *must* function generates the common updated abstract cache state with content that is guaranteed to be in the concrete cache state at that program point. The *may* function computes a common abstract cache state with content that might be found in the concrete cache state at that program point. If the memory block is not

in the abstract cache state then it is safe to be classified as cache miss. If the memory block cannot be classified as always hit or always miss then it is not classified. The precision of the analysis was further improved with persistence analysis, which classifies cache accesses within loop structures where the accesses for the first iteration may result in a miss but for all the other iteration in a hit [39]. The analysis was firstly valid only for caches with LRU replacement policy, but later it was extended also for FIFO, MRU and PLRU [94].

Static Analysis of Cache with Prefetcher

Utilizing a prefetcher in an real-time system can do both, harm performance or improve it. If the prefetcher brings instructions into the cache before they are required it will reduce the number of cache misses, but if the prefetched block evicts useful cache lines then the number of cache misses will be increased. The other issue is blocking, where a prefetch access can block instructions or data fetch by prefetching a useless memory block with an incorrect target. In both cases, the presence of a prefetcher affects the WCET bound of the system, which means its inclusion in the analysis is unavoidable. However, including the behavior of the prefetcher in the analysis increases the complexity even more by making the prefetcher not so suitable for real-time systems.

Yan and Zhang, in [123], extend the static cache simulation approach [6] by including the impact of a next-N-line prefetcher on the instruction categorization outcome. The basic idea of this approach is to firstly compute the status of each instruction and then update their status based on the changes due to prefetching. The algorithm consist of four steps: initialization, loop analysis without prefetching, loop analysis with prefetching and branch analysis. The initialization step determines the status of each instruction and calculates their access time. The loop analysis without prefetcher categorizes all loop instructions in case they can be categorized as first miss or first hit. The loop analysis with prefetcher updates the status of the categorized loop instructions by including the effect of cache conflict due to prefetching. The last step, branch analysis, deals with non-fall-through targets by updating their status based on the outcome of the conflict set consisting of prefetched instructions. The evaluation shows that the presence of a next-N-line prefetcher can increase the WCET of the code due to the cache pollution that it generates.

Later, Ding and Zhang in [30] improve the WCET from the previous approach by extending the previous next-N-line prefetcher with loop back-end prefetching and call it Loop-Based instruction prefetching. The scheme performs next-N-line by default and switches to the other scheme only when execution reaches a loop boundary. At that moment, instead of the next sequential memory block, the block that has the head of the loop is prefetched. Switching between loop-directed prefetching and next-N-line prefetching is controlled through a *Loop-BranchEnable* signal, which is active only when a loop-branch instruction is executed. The solution requires a special instruction for loop branches in order to distinguish them from the other control-flow instruction. Loop prefetch targets are stored in a hardware table as a pair of loop-bound address and the address of the loop head. Since they can be identified statically, they are derived before the execution starts. The WCET analysis, similar to the previous solution, is based on static cache simulation. The only difference is that the loop-based-prefetching now eliminates the cache conflict between loop instructions and the following instructions after the

loop, as it was the case with the previous solution. Now the conflict occurs only in the last iteration of the loop because the prefetcher always triggers loop head prefetching at the end of the loop.

7.2 Scratchpad Memories as an alternative in Hard Real-time Systems

Scratchpads are fully predictable concerning the time of memory references [116]. Even though the content can be managed statically or dynamically, both of these approaches are fully under software control, which supports predictability. The disadvantage of using scratchpads is the necessity for the correction of the execution flow with additional control-flow instructions due to the dislocation of the code parts into different memory address spaces.

Wehmeyer and Marwedel in [115] have analyzed the influence of scratchpad memory size on WCET. They use static allocation where the selection of scratchpad-allocated objects is formulated as an integer programing problem. The objective is the allocation of basic blocks with higher execution or access frequency to scratchpad memory. The result shows that increasing the scratchpad size drastically improves the WCET value. However, the used algorithm has the goal of energy efficiency and not WCET performance. A WCET-aware static allocation algorithm for scratchpad memory has been proposed by Falk and Kleinorge in [36]. The scratchpad-allocation problem is again formulated as an ILP problem, but now it has the goal of optimizing the worst-case execution path (WCEP). The algorithm is applied exclusively to the program segments that are part of the WCEP. After each allocation decision, the algorithm checks if another code path has become the new WCEP. The results show that the use of scratchpad can improve the WCET up to 40% compared to the same architecture without on-chip memory.

Patel et al. in [80] proposed a static instruction scratchpad allocation scheme that would satisfy time constrains for the PRET architecture [68]. PRET uses the ARM ISA extended with a *deadline instruction*. This instruction is used in pair in order to define the execution time deadline of the code fraction enclosed between these two instructions called timed block. The first deadline instruction is at the beginning, while the second at the end. The objective of the scratchpad allocation scheme is to allocate instructions from those timed blocks which are violating the local time deadlines. The selection is based on a greedy algorithm. The approach was later improved by Prakash and Patel [83], who formulate the allocation as an ILP problem. The ILP approach allocates only the minimum number of instructions that are necessary to meet the local time deadline, thus leaving the remaining free scratchpad space to be used for the allocation of other instructions from other critical blocks in the same program to meet timing requirements specified in their blocks.

Metzlaff et al. in [75] propose an instruction scratchpad that dynamically manages the content during the run-time of the code. The idea is to avoid a conflict between instruction and data paths by dynamically allocating the functions to the scratchpad. In every call or return, the scrachpad controller copies the functions from main memory to the scratchpad, where the granularity of transaction is the whole function. Thus, every instruction of the code that is part of a function is accessed from the scratchpad. A function can only be loaded or replaced on the whole, which restricts the minimal size of the scratchpad. The solution is implemented in a

simultaneous multithreaded architecture in order to increase processor utilization. Thus, while one thread is executing instructions from the scratchpad, the other one will load the upcoming function. To avoid collisions between threads, the scratchpad is partitioned with each thread having a separate space assigned. Avoidance of interference is important for precise WCET estimation.

7.3 Cache Locking Techniques for Predictability Improvement

Cache locking is a promising approach for simplifying the WCET analysis of the cache and increasing the cache predictability, since the contents of the locked instructions are statically known. Once the cache block is locked, it cannot be evicted anymore under the replacement policy [1]. This guarantees that the access of those instructions will always result in a cache hit. The disadvantage of such a technique is that it reduces the size of the unlocked part of the cache available for the rest of the program. Cache locking can be performed in a *static* or *dynamic* manner. The static implementation loads and locks the selected code fractions into the cache during the system boot time and keeps them unchanged throughout the whole execution, while the dynamic solution locks and unlocks the cache content interchangeably during the execution time of the code. In the following we discuss cache locking techniques for improving predictability and WCET bound.

In [37], Falk et al. present a static instruction-cache locking technique for minimizing the WCET. The objective of the approach is to lock functions that are part of the WCEP and with that to also reduce the WCET. The algorithm firstly builds a context-specific function call graph of the code and then uses a heuristic algorithm to determine the WCEP of the set of paths. The next step is the selection of the most frequently accessed functions to be locked. Since the WCEP can be changed after each locked function, the algorithm takes also into consideration the possibility of WCEP switching. In fact, the algorithm recalculates the WCEP after each locking decision. Liu et al. in [69] use the same approach of function locking, except that the heuristic algorithm for function selection is replaced with an ILP model. Later Plazar et al. in [82] expand the ILP model to the level of basic blocks. Furthermore, the model takes into account the effect of locked memory blocks and with that it also eliminates the need for a repetitive calculation of the WCEP. Li et al. in [65] combine cache locking with code positioning. After the selection of cache locking content, the code positioning is employed to reduce the conflicts among the blocks to be locked. If a cache conflict is detected, the code positioning will allocate and place those blocks continuously in the memory layout. Such changes also require the insertion of additional jump instructions in order to make the control flow correct. However, all these approaches have implemented full cache locking which means that caching other instructions during the run-time can not be considered anymore an option. Therefore, Ding et al. in [28] introduce partial cache locking that has the flexibility to lock only part of the cache while allowing the rest of cache content to be still an object of replacement policy. The locking mechanism integrates cache locking with cache modeling, where cache modeling determines which accesses are predictable while cache locking optimizes the WCET by locking unpredictable ones. The selection of the cache locking contents can be done through ILP formulation if the analysis can be performed on concrete cache states, or by heuristic approaches when the complexity of the cache analysis

increases and abstract cache states need to be used.

However, static cache locking has shown to be successful only for large caches where the scope of WCET optimization is larger. In this context, dynamic cache locking manifest an even further improvement on WCET optimization. Puaut in [85] suggests greedy and genetic algorithms for dynamic locking of the cache content. The proposed algorithms divide the code into regions from which the cache contents are statically selected. Each region has a reload point located at the beginning of the loop or function in order to enhance temporal locality. As the program execution moves from one reload point to the other, the cache content is loaded and remains unchanged until the next reload point. The greedy algorithm has predefined reload points and the selection of contents relies on the knowledge of the execution frequency of the blocks along the WCEP, while the genetic algorithm searches the space for cache contents and reload points in a blind manner. The cache-contents selection criteria for both algorithms are based on the execution frequencies of basic blocks along the WCEP. Ding et al. in [29] propose flexible loop-based dynamic cache locking approach that selects the memory blocks that should be locked and the position of the locking points. The approach differs from the previous one by adapting partial cache locking. Its main advantage is that the position of locking for memory blocks that belong to the inner loop can be locked at different loop levels. The selection of the memory blocks to be locked and their locking points is done through ILP modeling. Vera et al. in [113] use dynamic cache locking to lock the data cache on those code regions where the addresses of memory accesses can not be determined at compile time. The cache is locked just before the execution goes through such a region and unlocked after the region is executed. The content that is load and locked in the data cache is determined by using a simple analysis based on reuse analysis. This approach can be implemented on instruction cache as well by locking code regions which increases the uncertainty of the WCET bound.

7.4 Instruction Cache with Time-predictable Architecture

Method Cache

A method cache was firstly proposed by Schoeberl in [97] as part of the Java Processor for Java programs and later also extended for functions and procedures of procedural languages [27]. In contrast to conventional cache, a method cache has a coding granularity of a method. Thus, whenever a cache miss occurs, the whole method is loaded into the cache. This strategy improves predictability by restricting the occurrences of cache misses only to method calls and returns. Any other access within the method will result in a hit. Another advantage of the method cache is that it removes the interference with data cache since the instructions from main memory are loaded only on call or return.

The simplest configuration is the method cache that holds only one method. The timing model for such a configuration consists of the transfer time needed to load the method from the main memory to the cache and the hit time of each access within the method. This makes the integration of method cache analysis into the WCET analysis a very easy process. However, single-method caches are not so performance efficient especially when the loaded method has a large overhead and only a fraction of it is executed. For performance improvement, a method

cache with a variable number of blocks was also proposed, where the size of the blocks can be fixed or variable [27]. In the first case, the on-chip memory is divided into blocks with a fixed size and the maximum number of cached methods is limited by the number of blocks. The second solution is more flexible since the methods can be allocated variably in the on-chip memory. Compared with the single-method cache, multi-block method caches are more complex due to the additional hardware required for the positioning of the methods into the cache and also for the replacement policy, a disadvantage which is also reflected in the timing analysis of the method cache.

7.5 Memory with WCET-aware Instruction Prefetcher

A Dual-mode Instruction Prefetch Scheme

The dual-mode instruction prefetcher, proposed by Lee et al. in [64], is an alternative to instruction caching. Its main concept is based on special threads associated with each instruction block to guide the prefetcher through the prefetching process. The scheme operates in real-time and non real-time mode depending on the criticality of the executed task. In real-time mode, the prefetcher has the tendency to improve the WCET, while in non real-time mode the average-case execution time is improved. For real-time tasks, threads are generated during compile stage through the analysis of the worst case execution path of the code. Non real-time tasks have threads which are updated dynamically by pointing to a block that is most likely to be accessed again. The hardware for supporting this scheme consist of two instruction buffers, one prefetch control unit and a thread write buffer. One of the two instruction buffers places the currently executed memory block, while the other places the memory block being prefetched. When the execution in the current block is finished, the blocks alternate their functions. The prefetch-control unit searches through the buffers if the requested reference is already present or it should be fetched from the memory. The thread-write buffer is active only for non real-time tasks in order to update the threads in case they have done wrong target guessing. The prefetcher was later extended with a larger prefetch buffer in order to keep loops and reduce repeated prefetching when loops can fit into the buffer [63]. The timing scheme used for WCET estimation was also revised to include the prefetch buffer.

TickPAD Memory

TickPAD is a memory system proposed by Kuo et al. in [59] for a predictable and efficient execution of synchronous programs. The memory consists of four types of memory components. The first component is called *spatial memory pipeline* and its job is to perform sequential prefetching by utilizing free bus cycles and bringing the next sequential block. Its architecture consists of two buffers of the size of one cache line that are interleaved to present fetch and prefetch stage. In addition, the component has hardware that monitors the high bits of instructions in order to detect branch instructions at disabled prefetching when they are encountered. The second component is composed of the *tick address queue*, designed to maintain the list of resumption addresses at thread-switching points, and the *tick instruction buffer* to place the preloaded instructions. The third component is the *associative loop memory*, which is a small

memory for loading loops. Which instructions of the loop will be allocated is determined statically and during the runtime. When the loop is encountered, the processor is stalled until the whole loop is loaded. The last component is called *command table*, which is a lookup table that contains a set of commands for instructing the TickPAD during runtime. The command in the table can be: *discard* to reset the content of the loop memory, *store* to load the loop memory, *fill tick instruction buffer* to bring the first line of the thread, and *load tick address queue* to maintain the list of thread resumptions.

Shared Memory Tree Prefetching

Garside and Audsley in [42] proposed a hardware prefetch design for multicore real-time systems. The system accesses the main memory across a dedicated tree-based interconnection, where the leaves of the tree are representing connected CPUs, while the external memory is connected at the root. The tree enables the WCET analysis to be composable by splitting the memory bandwidth among all the tasks. Therefore, when the WCET bound of a single task is estimated, the analysis must assume a maximal blocking from all other tasks that access the memory through the same interconnection. However, the assumption of fully loaded memory interconnect is not the case most of the time, thus generating many reserved spare slots that are not used. The prefetcher utilizes these spare time slots within the memory systems to perform safety prefetching without harming the WCET bound. Its position between the main memory and the tree interconnection allows it to snoop all read requests before they enter the memory controller in order to detect spare slots and assign them for prefetching. In addition, if a prefetched block is referenced from the CPU then the fetch slot for that reference will be replaced with another prefetch slot. The architecture of the prefetcher consists of the *demand queue* to store fetch requests, the *hit queue* to store spare slots, a *prefetch calculator* which monitors the fetch requests and decides when a prefetch request should be generated, a *stream buffer* to store the existing stream data, the *demand mem queue* to store request which are not discarded, the *prefetch buffer* to store prefetch requests, the *PF merger* which merges prefetch requests with spare slots, a *PF queue* stores merged prefetches ready to be issued, and the *Squash filter* to discard fetch requests that are already prefetched. Placing the prefetching directly into the cache may affect the WCET by evicting useful cache lines. For this reason, prefetched blocks are placed into additional cache memory called prefetch cache.

7.6 Chapter Summary

To summarize, cache memories are the main source of unpredictability in the memory hierarchy. A precise timing analysis of them is almost infeasible, and with the use of abstraction a lot of information is lost. Implementation of cache locking and scratchpad memory improves timing predictability through the memory hierarchy, but their presence most of the time lowers the performance of the system compared to the one that has conventional cache. Thus they can be considered as beneficial only when the new derived WCET bound becomes shorter as result of increased predictability. On the other hand, the memories with WCET-aware prefetcher also have their disadvantages. The dual-mode prefetching scheme requires additional threads

attached to each memory block to guide the prefetcher through the WCEP, the TickPAD is designed to work with synchronous languages, while the Shared Memory Tree Prefetching can only grantee that the WCET will not get worse due to the presence of the prefetcher.

CHAPTER 8

Conclusion and Future Work

In this chapter we summarize our contribution and give directions for our future work, on further improvements of timing performance in systems that run single-path code.

8.1 Conclusion

The single-path conversion is an alternative for reducing the complexity of timing analysis in hard-real time systems. The approach eliminates all timing variation through the conversion of the conventional code into a code that has a singleton trace of execution for any input data. It is sufficient to run the code once and with that to derive its WCET bound. However, the advantage of generating single-path time-predictable code comes at the cost of longer execution time.

A single-path code prefetcher offers the opportunity of improving timing performance for the system that runs single-path code. The prefetcher reduces the cache miss penalty time and the cache miss rate by prefetching instructions into the cache before they are required for execution. The key concept of the proposed solution is based on the fully predictable execution behavior of single-path code, which allows the prefetcher to operate in a fully predictable fashion both in time domain, by determining the exact moment when the prefetch requests should be issued, and in the value domain, by determining the correct value of the prefetch target addresses. The prefetcher also takes into consideration the organization of the cache in order to avoid an eviction of any useful cache block with a prefetched block. This way, the prefetcher not only contributes in exploitations of codes spatial locality but it also improves exploitations of codes temporal locality. Using the metrics for prefetch performance comparison, it can be seen that the proposed prefetcher achieves full coverage and full accuracy on all possible cache misses that can occur during the execution of the code. Moreover, the modular design makes the prefetcher scalable and keeps its design complexity under control, by allowing each module to be further developed in the future, without affecting the behavior of the other modules.

To bring stability to the execution timing, we presented a new organization of the memory hierarchy. Although the single-path code forces the execution to run through the same order

of instructions, there is still timing variety that occurs in each layer of the memory hierarchy. Stable timing is achieved when the sequence of states of memory hierarchy becomes repeatable for any iteration of the code. At the cache level the only restriction is not to use an n-way associative cache organization with random replacement policy. Any other organization of the cache that has well defined rules on cache-line replacement satisfies the requirement for cache state repeatability for single-path code. At an arbitration level, to avoid any unpredictable interference between instruction and data path of the memory, we propose the use of explicitly controlled data scratchpad that will upload all required data before the execution starts, or in scheduled fashion with a strictly defined moment if the data amount is much larger than the size of the scratchpad. For the main memory with DRAM technology, we propose the use of a special instruction that will reset the refreshing controller at the beginning of each task, thus synchronizing both the refreshing and memory accessing process.

We also demonstrate the benefit of the new memory hierarchy with the single-path code prefetcher in the context of performance improvement. The implementation was synthesized and uploaded on an FPGA board where a set of Mälardalen benchmarks were executed. The results show that the presence of the prefetcher is always beneficial but the scale of improvement depends on the structure of the code.

8.2 Future Work

The work in this thesis opens several new challenges that would further improve the timing performance of single-path code. First, in pipelined systems, a wrong speculation on branch target can cost the execution a few more cycles. Nowadays, systems employ *branch predictors* to predict branch conditions and branch targets. However, the decision is speculative since it is based on the previous behavior of the branch. By using time-predictable properties of single-path code, we can build a fully time-predictable branch prediction. Furthermore, the information of the RPT can also be used for guiding the branch predictor, thus eliminating the need for additional memory to store the decisions and the targets of the branches.

Second, aligning the beginning of a single-path loop with the beginning of cache line can significantly reduce the number of cache misses for loops that are larger than the cache size, considering that after transformation they are set to iterate for the maximum number which is equal to the bound of the original loop. This can be implemented in the compiler, where after transformation padding with *nop* instruction can be done until the beginning of the loop gets aligned with the cache line. As a concept, the benefit of such an approach was demonstrated in [20]. The strategy itself is quite old, but so far it has been implemented only for data caches.

Last, the single-path code prefetcher can also be adopted for multi-task systems with a table driven scheduler. In such a system, the sequence of tasks and the preemption moments of task switching are clearly off-line defined. This exact knowledge about task sequence and timing can be utilized by a prefetcher to start the prefetching of the next task before task switching is performed.

Bibliography

[1] Tosiron Adegbija and Ann Gordon-Ross. Phase-based cache locking for embedded systems. In *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, pages 115–120. ACM, 2015.

[2] Benny Akesson and Kees Goossens. Architectures and modeling of predictable memory controllers for improved system integration. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[3] Benny Akesson and Kees Goossens. *Memory controllers for real-time embedded systems*. Springer, 2011.

[4] John R Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189. ACM, 1983.

[5] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri. A post-compiler approach to scratchpad mapping of code. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 259–267. ACM, 2004.

[6] Robert D Arnold, Frank Mueller, David B Whalley, and Marion G Harmon. Bounding worst-case instruction cache performance. In *RTSS*, pages 172–181, 1994.

[7] Pavel Atanassov, Stefan Haberl, and Peter Puschner. *Heuristic worst-case execution time analysis*. In Proceedings of the 10th European Workshop on Dependable Computing, 1999.

[8] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[9] Adam Betts and Guillem Bernat. Tree-based wcet analysis on instrumentation point graphs. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 8–pp. IEEE, 2006.

[10] Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, 2003.

[11] Balasubramanya Bhat and Frank Mueller. Making dram refresh predictable. *Real-Time Systems*, 47(5):430–453, 2011.

[12] Ishwar Bhati, Mu-Tien Chang, Zeshan Chishti, Shih-Lien Lu, and Bruce Jacob. Dram refresh mechanisms, penalties, and trade-offs. *IEEE Transactions on Computers*, 65(1):108–121, 2016.

[13] Dominique Brière and Pascal Traverse. Airbus a320/a330/a340 electrical flight controls-a family of fault-tolerant systems. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 616–623. IEEE, 1993.

[14] Claire Burguiere and Christine Rochange. On the complexity of modeling dynamic branch predictors when computing worst-case execution time. In *Proceedings of the ERCIM/DECOS Workshop On Dependable Embedded Systems*. Citeseer, 2007.

[15] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 40–52. ACM, 1991.

[16] Pei Cao, Edward W Felten, Anna R Karlin, and Kai Li. A study of integrated prefetching and caching strategies. *ACM SIGMETRICS Performance Evaluation Review*, 23(1):188–197, 1995.

[17] I-Cheng K Chen, Chih-Chieh Lee, and Trevor N Mudge. Instruction prefetching using branch prediction information. In *Computer Design: VLSI in Computers and Processors, 1997. ICCD'97. Proceedings., 1997 IEEE International Conference on*, pages 593–601. IEEE, 1997.

[18] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers*, 44(5):609–623, 1995.

[19] Bekim Cilku, Daniel Prokesch, and Peter Puschner. A time-predictable instruction-cache architecture that uses prefetching and cache locking. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2015 IEEE International Symposium on*, pages 74–79. IEEE, 2015.

[20] Bekim Cilku and Peter Puschner. Towards a time-predictable hierarchical memory architecture-prefetching options to be explored. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2010 13th IEEE International Symposium on*, pages 219–225. IEEE, 2010.

[21] Bekim Cilku and Peter Puschner. Designing a time predictable memory hierarchy for single-path code. *ACM SIGBED Review*, 12(2):16–21, 2015.

[22] Antoine Colin and Guillem Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *Real-Time Systems, 2002. Proceedings. 14th Euromicro Conference on*, pages 50–59. IEEE, 2002.

[23] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2-3):249–274, 2000.

[24] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[25] Harvey G Cragon. The elements of single-chip microcomputer architecture. *Computer*, (10):27–41, 1980.

[26] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *Parallel Processing, 1993. ICPP 1993. International Conference on*, volume 1, pages 56–63. IEEE, 1993.

[27] Philipp Degasperi, Stefan Hepp, Wolfgang Puffitsch, and Martin Schoeberl. A method cache for patmos. In *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 100–108. IEEE, 2014.

[28] Huping Ding, Yun Liang, and Tulika Mitra. Wcet-centric partial instruction cache locking. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 412–420. IEEE, 2012.

[29] Huping Ding, Yun Liang, and Tulika Mitra. Wcet-centric dynamic instruction cache locking. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 27. European Design and Automation Association, 2014.

[30] Yiqiang Ding and Wei Zhang. Loop-based instruction prefetching to reduce the worst-case execution time. *IEEE transactions on computers*, 59(6):855–864, 2010.

[31] Leonardo Ecco, Adam Kostrzewa, and Rolf Ernst. Minimizing dram rank switching overhead for improved timing bounds and performance. In *Real-Time Systems (ECRTS), 2016 28th Euromicro Conference on*, pages 3–13. IEEE, 2016.

[32] Philip G Emma, Allan Hartstein, Thomas R Puzak, and Viji Srinivasan. Exploring the limits of prefetching. *IBM Journal of Research and Development*, 49(1):127–144, 2005.

[33] Jakob Engblom. Processor pipelines and static worst-case execution time analysis. 2002.

[34] Andreas Ermedahl and Jan Gustafsson. Deriving annotations for tight calculation of execution time. In *European Conference on Parallel Processing*, pages 1298–1307. Springer, 1997.

[35] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *OASIcs-OpenAccess Series in Informatics*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[36] Heiko Falk and Jan C Kleinsorge. Optimal static wcet-aware scratchpad allocation of program code. In *Proceedings of the 46th Annual Design Automation Conference*, pages 732–737. ACM, 2009.

[37] Heiko Falk, Sascha Plazar, and Henrik Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 143–148. ACM, 2007.

[38] Babak Falsafi and Thomas F Wenisch. A primer on hardware prefetching. *Synthesis Lectures on Computer Architecture*, 9(1):1–67, 2014.

[39] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[40] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. Proactive instruction fetch. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 152–162. ACM, 2011.

[41] Michael Ferdman, Thomas F Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal instruction fetch streaming. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2008.

[42] Jamie Garside and Neil C Audsley. Wcet preserving hardware prefetch for many-core real-time systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, page 193. ACM, 2014.

[43] Edward H Gornish and Alexander Veidenbaum. An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. In *Parallel Processing, 1994. ICPP 1994 Volume 2. International Conference on*, volume 2, pages 281–284. IEEE, 1994.

[44] Daniel Grund, Jan Reineke, and Reinhard Wilhelm. A template for predictability definitions with supporting evidence. In *OASIcs-OpenAccess Series in Informatics*, volume 18. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2011.

[45] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks: Past, present and future. In *OASIcs-OpenAccess Series in Informatics*, volume 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[46] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Bjorn Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution.

In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 57–66. IEEE, 2006.

[47] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *ACM SIGBED Review*, 12(1):28–36, 2015.

[48] Christopher Healy, Mikael Sjödin, Viresh Rustagi, David Whalley, and Robert Van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2-3):129–156, 2000.

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

[50] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[51] Ibrahim Hur and Calvin Lin. Memory prefetching using adaptive stream detection. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 397–408. IEEE Computer Society, 2006.

[52] Bruce Jacob, Spencer Ng, and David Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[53] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 252–263. ACM, 1997.

[54] Norman P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pages 364–373. IEEE, 1990.

[55] Raimund Kirner and Peter Puschner. Obstacles in worst-case execution time analysis. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 333–339. IEEE, 2008.

[56] Raimund Kirner and Peter Puschner. Time-predictable computing. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 23–34. Springer, 2010.

[57] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Symbolic loop bound computation for wcet analysis. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 227–242. Springer, 2011.

[58] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[59] Matthew Kuo, Partha Roop, Sidharta Andalam, and Nitish Patel. Precision timed embedded systems using tickpad memory. In *Application of Concurrency to System Design (ACSD), 2013 13th International Conference on*, pages 206–215. IEEE, 2013.

[60] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[61] Edward A Lee. Absolutely positively on time: what would it take?[embedded computing systems]. *Computer*, 38(7):85–87, 2005.

[62] Insup Lee, Joseph YT Leung, and Sang H Son. *Handbook of real-time and embedded systems*. CRC Press, 2007.

[63] Minsuk Lee, Sang Lyul Min, and Chong Sang Kim. A worst case timing analysis technique for instruction prefetch buffers. *Microprocessing and Microprogramming*, 40(10-12):681–684, 1994.

[64] Minsuk Lee, Sang Lyul Min, Chang Yun Park, Young Hyun Bae, Heonshik Shin, and Chong-Sang Kim. A dual-mode instruction prefetch scheme for improved worst case and average case program execution times. In *Real-Time Systems Symposium, 1993., Proceedings.*, pages 98–105. IEEE, 1993.

[65] Fuyang Li, Mengying Zhao, and Chun Jason Xue. C3: Cooperative code positioning and cache locking for wcet minimization. In *2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 51–59. IEEE, 2015.

[66] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. Modeling control speculation for timing analysis. *Real-Time Systems*, 29(1):27–58, 2005.

[67] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM SIGPLAN Notices*, volume 30, pages 88–98. ACM, 1995.

[68] Isaac Liu, Jan Reineke, and Edward A Lee. A pret architecture supporting concurrent programs with composable timing properties. In *2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers*, pages 2111–2115. IEEE, 2010.

[69] Tiantian Liu, Minming Li, and Chun Jason Xue. Minimizing wcet for real-time embedded systems via static instruction cache locking. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 35–44. IEEE, 2009.

[70] Paul Lokuciejewski and Peter Marwedel. *Worst-case execution time aware compilation techniques for real-time systems*. Springer Science & Business Media, 2010.

[71] Chi-Keung Luk and Todd C Mowry. Architectural and compiler support for effective instruction prefetching: a cooperative approach. *ACM Transactions on Computer Systems*, 19(1):71–109, 2001.

[72] Thomas Lundqvist and Per Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Real-time systems symposium, 1999. Proceedings. The 20th IEEE*, pages 12–21. IEEE, 1999.

[73] Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):05–1, 2016.

[74] Peter Marwedel. *Embedded system design: Embedded systems foundations of cyber-physical systems*. Springer Science & Business Media, 2010.

[75] Stefan Metzlaff, Sascha Uhrig, Jörg Mische, and Theo Ungerer. Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. In *Proceedings of the 9th workshop on MEmory performance: DEaling with Applications, systems and architecture*, pages 38–45. ACM, 2008.

[76] Frank Mueller. Timing analysis for instruction caches. *Real-time systems*, 18(2-3):217–247, 2000.

[77] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.

[78] Marco Paolieri, Eduardo Quiñones, and Francisco J Cazorla. Timing effects of ddr memory systems in hard real-time multicore architectures: Issues and solutions. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(1s):64, 2013.

[79] Joseph CH Park and Mike Schlansker. *On predicated execution*. Hewlett-Packard Laboratories Palo Alto, California, 1991.

[80] Hiren D Patel, Ben Lickly, Bas Burgers, and Edward A Lee. A timing requirements-aware scratchpad memory allocation scheme for a precision timed architecture. Technical report, DTIC Document, 2008.

[81] Jim Pierce and Trevor Mudge. Wrong-path instruction prefetching. In *Microarchitecture, 1996. MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on*, pages 165–175. IEEE, 1996.

[82] Sascha Plazar, Jan C Kleinsorge, Peter Marwedel, and Heiko Falk. Wcet-aware static locking of instruction caches. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 44–52. ACM, 2012.

[83] Aayush Prakash and Hiren D Patel. An instruction scratchpad memory allocation for the precision timed architecture. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 659–664. EDA Consortium, 2012.

[84] Daniel Prokesch, Stefan Hepp, and Peter Puschner. A generator for time-predictable code. In *Real-Time Distributed Computing (ISORC), 2015 IEEE 18th International Symposium on*, pages 27–34. IEEE, 2015.

[85] Isabelle Puaut. Wcet-centric software-controlled instruction caches for hard real-time systems. In *18th Euromicro Conference on Real-Time Systems (ECRTS'06)*, pages 10–pp. IEEE, 2006.

[86] Peter Puschner. Transforming execution-time boundable code into temporally predictable code. In *Design and Analysis of Distributed Embedded Systems*, pages 163–172. Springer, 2002.

[87] Peter Puschner. The single-path approach towards wcet-analysable software. In *Industrial Technology, 2003 IEEE International Conference on*, volume 2, pages 699–704. IEEE, 2003.

[88] Peter Puschner and Alan Burns. Writing temporally predictable code. In *Object-Oriented Real-Time Dependable Systems, 2002.(WORDS 2002). Proceedings of the Seventh International Workshop on*, pages 85–91. IEEE, 2002.

[89] Peter Puschner, Bekim Cilku, and Daniel Prokesch. Constructing time-predictable mpsocs: Avoid conflicts in temporal control. In *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2016 IEEE 10th International Symposium on*, pages 321–328. IEEE, 2016.

[90] Peter Puschner, Raimund Kirner, Benedikt Huber, and Daniel Prokesch. Compiling for time predictability. In *International Conference on Computer Safety, Reliability, and Security*, pages 382–391. Springer, 2012.

[91] Peter Puschner, Raimund Kirner, and Robert G Pettit. Towards composable timing for real-time programs. In *Future Dependable Distributed Systems, 2009 Software Technologies for*, pages 1–5. IEEE, 2009.

[92] Peter P Puschner and Anton V Schedl. Computing maximum task execution times—a graph-based approach. *Real-Time Systems*, 13(1):67–91, 1997.

[93] Thomas R Puzak, Allan Hartstein, Philip G Emma, and Viji Srinivasan. When prefetching improves/degrades performance. In *Proceedings of the 2nd conference on Computing frontiers*, pages 342–352. ACM, 2005.

[94] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[95] Glenn Reinman, Brad Calder, and Todd Austin. Fetch directed instruction prefetching. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 16–27. IEEE, 1999.

[96] Khalid Sayood. *Introduction to data compression*. Morgan Kaufmann, 2017.

[97] Martin Schoeberl. A time predictable instruction cache for a java processor. In *OTM Confederated International Conferences On the Move to Meaningful Internet Systems*, pages 371–382. Springer, 2004.

[98] Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, 2009(1):1, 2009.

[99] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, et al. T-crest: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

[100] Martin Schoeberl, Benedikt Huber, and Wolfgang Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Systems*, 49(1):1–28, 2013.

[101] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: a time-predictable microprocessor. *Real-Time Systems*, pages 1–35, 2018.

[102] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[103] Alan J Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, 1978.

[104] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[105] James E Smith and W-C Hsu. Prefetching in supercomputer instruction caches. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 588–597. IEEE Computer Society Press, 1992.

[106] Lawrence Spracklen, Yuan Chou, and Santosh G Abraham. Effective instruction prefetching in chip multiprocessors for modern commercial applications. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 225–236. IEEE, 2005.

[107] Viji Srinivasan, Edward S Davidson, Gary S Tyson, Mark J Charney, and Thomas R Puzak. Branch history guided instruction prefetching. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 291–300. IEEE, 2001.

[108] Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[109] David B Stewart. Measuring execution time and real-time performance. In *Embedded Systems Conference (ESC)*, 2001.

[110] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Proceedings of the 43rd annual Design Automation Conference*, pages 358–363. ACM, 2006.

[111] John Tse and Alan Jay Smith. Cpu cache prefetching: Timing evaluation of hardware implementations. *IEEE Transactions on Computers*, 47(5):509–526, 1998.

[112] Steven P Vanderwiel and David J Lilja. Data prefetch mechanisms. *ACM Computing Surveys (CSUR)*, 32(2):174–199, 2000.

[113] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for tight timing calculations. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(1):4, 2007.

[114] Joachim Wegener, Harmen Sthamer, Bryan F Jones, and David E Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997.

[115] Lars Wehmeyer and Peter Marwedel. Influence of onchip scratchpad memories on wcet prediction. In *Proceedings of the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2004.

[116] Lars Wehmeyer and Peter Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Design, Automation and Test in Europe*, pages 600–605. IEEE, 2005.

[117] Lars Wehmeyer and Peter Marwedel. *Fast, Efficient and Predictable Memory Accesses*. Springer, 2006.

[118] Ingomar Wenzel, Raimund Kirner, Peter Puschner, and Bernhard Rieder. Principles of timing anomalies in superscalar processors. In *Fifth International Conference on Quality Software (QSIC'05)*, pages 295–303. IEEE, 2005.

[119] Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. Static timing analysis for hard real-time systems. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 3–22. Springer, 2010.

[120] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[121] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966, 2009.

[122] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[123] Jun Yan and Wei Zhang. Analyzing the worst-case execution time for instruction caches with prefetching. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(1):7, 2008.

Curriculum Vitae

Bekim Çilku

ADDRESS
Schottenfeldgasse 1/20
1070 Vienna, Austria

EDUCATION

Vienna University of Technology
Department of Real-Time Systems
Ph.D. Student, Present
Thesis Topic: *Time-predictable Memory Hierarchy*
Advisors: Ao.Prof.Dr. Peter Puschner

Electrotechnical Faculty Skopje
Department of Computer Engineering
MSc., August 2007
Thesis Topic: *Grid Computing Implementation in Ad-Hoc Networks*
Advisors: Prof.Dr. Aksenti Grnarov

Electrotechnical Faculty Skopje
Department of Computer Engineering
Dip. Ing., December 2003
Thesis Topic: *Routing Protocols in Ad-Hoc Networks*
Advisors: Prof.Dr. Aksenti Grnarov

RESEARCH EXPERIENCE

Research Assistant
Vienna University of Technology
Department of Real-Time Systems
1040 Vienna, Austria
January 2012 - November 2016

Research Assistant
South East European University
Department of Computer Science
1200 Tetovo, Macedonia
October 2004 - November 2009

PUBLICATIONS

1. Cilku, B., Puffitsch, W., Prokesch, D., Schoeberl, M. and Puschner, P., 2017, May. *Improving performance of single-path code through a time-predictable memory hierarchy*. In Real-Time Distributed Computing (ISORC), 2017 IEEE 20th International Symposium on (pp. 76-83). IEEE.

2. Schoeberl, M., Cilku, B., Prokesch, D. and Puschner, P., 2017. *Best Practice for Caching of Single-Path Code*. In OASIcs-OpenAccess Series in Informatics (Vol. 57). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

3. Puschner, P., Cilku, B. and Prokesch, D., 2016, September. *Constructing time-predictable MPSoCs: Avoid conflicts in temporal control*. In Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2016 IEEE 10th International Symposium on (pp. 321-328). IEEE.

4. Cilku, B., Crespo, A., Puschner, P., Coronel, J. and Peiro, S., 2015, June. *A TDMA-based arbitration scheme for mixed-criticality multicore platforms*. In Event-based Control, Communication, and Signal Processing (EBCCSP), 2015 International Conference on (pp. 1-6). IEEE.

5. Cilku, B. and Puschner, P., 2015. *Designing a time predictable memory hierarchy for single-path code*. ACM SIGBED Review, 12(2), pp.16-21.

6. Cilku, B., Prokesch, D. and Puschner, P., 2015, April. *A time-predictable instruction-cache architecture that uses prefetching and cache locking*. In Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2015 IEEE International Symposium on (pp. 74-79). IEEE.

7. Cilku, B., Crespo, A., Puschner, P., Coronel, J. and Peiro, S., 2014, December. *A memory arbitration scheme for mixed-criticality multicore platforms*. In Proc. 2nd Workshop on Mixed Criticality Systems (WMC), RTSS (pp. 27-32).

8. Cilku, B., Frömel, B. and Puschner, P., 2014, July. *A dual-layer bus arbiter for mixed-criticality systems with hypervisors*. In Industrial Informatics (INDIN), 2014 12th IEEE International Conference on (pp. 147-151). IEEE.

9. Cilku, B., Kammerer, R. and Puschner, P., 2015. *Aligning single path loops to reduce the number of capacity cache misses*. ACM SIGBED Review, 12(1), pp.13-18.

10. Cilku, B. and Puschner, P., 2013. *Towards temporal and spatial isolation in memory hierarchies for mixed-criticality systems with hypervisors*. Proc. ReTiMiCS, RTCSA, pp.25-28.

11. Cilku, B. and Puschner, P., 2011, March. *Using a local prefetch strategy to obtain temporal time predictability*. In Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on (pp. 227-233). IEEE.

12. Cilku, B. and Puschner, P., 2010, May. *Towards a time-predictable hierarchical memory architecture-prefetching options to be explored.* In Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2010 13th IEEE International Symposium on (pp. 219-225). IEEE.

13. Cilku, B., Grnarov, A. and Abazi, A., 2009, December. *Usability constraints of grid in ad hoc networks*. In Innovations in Information Technology, 2009. IIT'09. International Conference on (pp. 248-252). IEEE.

14. Cilku, B. and Grnarov, A., 2009, June. *New algorithms for efficient scheduling in Grid Ad-Hoc networks*. In Information Technology Interfaces, 2009. ITI'09. Proceedings of the ITI 2009 31st International Conference on (pp. 591-596). IEEE.

15. Grnarov, A., Cilku, B., Miskovski, I., Filiposka, S. and Trajanov, D., 2008. *Grid computing implementation in Ad hoc networks*. In Advances in Computer and Information Sciences and Engineering (pp. 196-201). Springer, Dordrecht.