ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Engineering and Architecture School.
Electronic Engineering.

Electronic and Communications Science and Technology Master Program.

Master Thesis

On

Hardware-Software Design of Embedded Systems

Instruction Prefetching Techniques for Ultra Low-Power Multicore Architectures

Candidate.
Maryam Payami

Supervisor.
Prof. Luca Benini

Academic Year 2015/2016

Session II

I would like to dedicate this thesis to my beloved parents and lovely husband

Abstract

As the gap between processor and memory speeds increases, memory latencies have become a critical bottleneck for computing performance. To reduce this bottleneck, designers have been working on techniques to hide these latencies. On the other hand, design of embedded processors typically targets low cost and low power consumption. Therefore, computer architects tend to adopt techniques that can satisfy these constraints for embedded domains. While out-of-order execution [1][2], aggressive speculation [3], and complex branch prediction algorithms [4] can help hide the memory access latency in high-performance systems, yet they can cost a heavy power budget and are not suitable for embedded systems.

Prefetching is another popular and effective method for hiding the memory access latency, and has been studied very well for high-performance processors to bridge the CPU-memory gap [5][6]. Similarly, for embedded processors with strict power requirements, the application of complex prefetching techniques is greatly limited, and most of the proposed techniques often suffer from a significant energy loss due to a large amount of wasteful over-prefetching operations and/or the complicated prefetching hardware components. This is while, for embedded systems low energy consumption is also one of the key design issues, especially for those used in battery driven mobile/hand-held devices where large and heavy batteries are not feasible. For this reason, a low power/energy solution is mostly desired in this context.

In this work, we focus on instruction prefetching in ultra-low power processing architectures and aim to reduce energy overhead of this operation by proposing a combination of simple, low-cost, and energy efficient prefetching techniques. We study a wide range of applications from cryptography

to computer vision and show that our proposed mechanisms can effectively improve the hit-rate of almost all of them to above 95%, achieving an average performance improvement of more than 2X. Plus, by synthesizing our designs using the state-of-the-art technologies we show that the prefetchers increase system's power consumption less than 15% and total silicon area by less than 1%. Altogether, a total energy reduction of 1.9X is achieved, thanks to the proposed schemes, enabling a significantly higher battery life.

Acknowledgements

I would like to thank my supervisor Professor Luca Benini for his support and guidance, and for providing this exciting opportunity to work with the state-of-the-art technologies and hardware platforms. My gratitude also goes to Dr. Igori Loi for defining this project and for his valuable insights. Lastly, I would like to thank my lovely husband Erfan for bearing with me and for all the guidance which I received from him throughout this work.

Contents

List of Figures

Chapter 1

Introduction and Related Works

The speed of integrated circuits has increased significantly during the last decades but the speed of memory circuits have not increased at the same rate. Therefore the memory has a large latency compared to the speed of the processor. Because of this large memory latency it is very important for the performance that the correct instructions are fetched at the correct time. Unfortunately the correct time is in most cases before the processor knows what instruction to fetch.

On the other hand, design of embedded processors typically targets low cost and low power consumption. Therefore, computer architects tend to adopt techniques that can satisfy these constraints. However, it has been widely known that improving program execution speed with advanced architectural features such as aggressive prefetching [7], speculation [3], branch prediction [4], etc., can cost a heavy power budget. As an example, prefetching is an effective and well-studied technique in high-performance processors to bridge the CPU-memory gap [5][6]. Plenty of complex prefetching techniques have been proposed to reduce I-cache misses for high system performance [7][8][9]. However, the existing schemes mainly focus on improving cache performance and often suffer a significant energy losses due to a large amount of wasteful over-prefetching operations and/or the complicated prefetching hardware components. For this reason, in embedded processors with strict power requirements the application of complex prefetching techniques is greatly limited, specially for the ones used in battery driven mobile/hand-held devices, where large and heavy batteries are not feasible. Therefore, a low power and low energy solution applicable to power and energy constrained embedded systems is highly desirable.

In this chapter, first we describe several commonly used memory latency hiding techniques. Then we introduce briefly the main characteristics of the cache memories

and finally we describe about prefetching techniques and present the related works in this area.

1.1 Memory Latency Hiding Techniques

Memory latency has become increasingly important as the gap between processors speeds and memory speeds grows [10]. Many methods have been proposed to overcome this disparity, such as caching [11], prefetching [7], multi-threading [12], and out of order execution [2]. These techniques fall broadly into two categories: those that reduce latency, and those that tolerate latency. Techniques for reducing latency include caching data and making the best use of those caches through locality optimizations. Techniques for tolerating latency include buffering, pipelining, prefetching, and multithreading [10].

In this thesis our main goal is the design of highly low-power processing platforms targeting embedded systems. For this reason, we do not focus on out-of-order and speculative execution and their design implications, because these techniques are usually used in high-performance systems with power hungry processors [1]. On the other hand, multithreading, caching, and prefetching are more general techniques which can be applied to different domains for latency reduction and toleration.

Multi-threading/programming allows for execution of multiple independent threads or programs and switching between them whenever some of them are stalled behind a memory or IO access [12]. While this technique is very effective at latency-hiding, it does not improve the performance of a single thread. Therefore, if an application is not inherently parallel or lack enough parallelism, it can not benefit from multi-threading. Effectiveness of multi-threading has been studied before, extensively, and several parallel processing platforms exist today [13][14][15]. For this reason, in this thesis we use a state-of-the-art parallel processing platform [13], and focus on the two other techniques for latency reduction: caching and prefetching. In the next subsections, we give an overview of caching and prefetching and introduce different concepts related to them.

1.2 An Overview of Caching

Caches are a critical first step toward coping with memory latency. A cache operates within a memory hierarchy by providing lower levels of the hierarchy with faster access

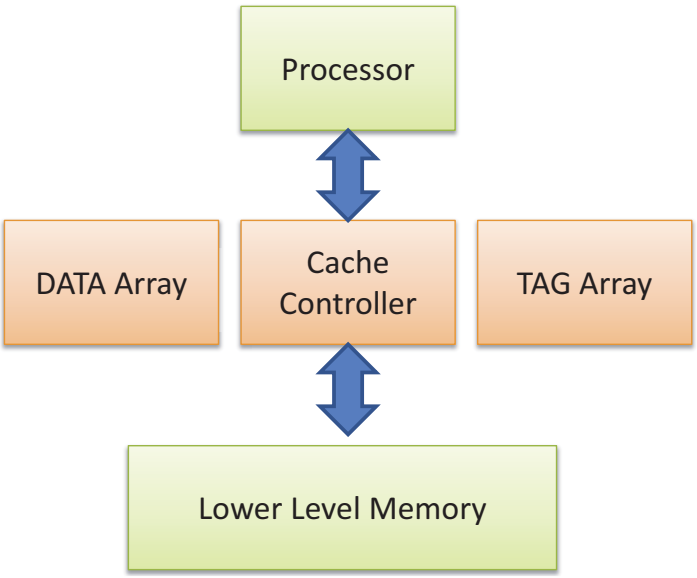

Figure 1.1: A simple block diagram of a cache.

to portions of higher levels of the hierarchy by storing the data within low latency memory. Caches work well due to the principle of locality [16]: It is assumed that if a section of memory is accessed at a certain point in time, it is probable that more memory accesses will occur close to that area of memory. By storing the block of memory around this area within the cache, it is hoped that further memory accesses can circumvent the latencies associated with main memory accesses [11].

A *cache* is a small fast memory located near the processor which contains the data recently accessed. A *cache-line* is the smallest amount of data that can be transferred between the upper level memory and the cache. If the data/instruction required by the processor is located in the cache, it is a *HIT*. Otherwise, it is a *MISS*. The *mapping* defines how to assign one upper level memory block to one cache line. In subsection 1.2.2 *mapping* is explained in more details. Also *replacement policy* is explained in subsection 1.2.1.

A simplified block diagram of a cache is shown in Figure 1.1. As can be seen, all caches are composed of some form of controller (Cache Controller), and data and tag arrays. These two memory structures are conventionally implemented with SRAM based memory, however, it is also possible to implement them using standard cell memories (SCM) using controlled placement [17][13]. SCMs allow for further voltage and energy reduction and can be beneficial in embedded low-power platforms.

Caches can be used both for the instruction and the data interfaces of the processor, and a hierarchy of multiple caches with different characteristics can exist in high-

performance systems. In the context of low-power embedded systems, however, usually only instruction-caches are implemented and other levels of caches are avoided due to their large area and power consumption [18]. For this reason, in this thesis we focus on the instruction caches and instruction prefetching for latency reduction.

1.2.1 Replacement Policy

When a miss occurs, the cache controller must select a block to be replaced with the desired data. A replacement policy determines which block should be replaced. With "direct-mapped placement" the decision is simple because there is no choice: only one block frame is checked for a hit and only that block can be replaced. With "fully-associative" or "set-associative placement", there are more than one block to choose from on a miss. There are two primary strategies: Random and Least-Recently Used (LRU). Here is a list of the most widely used replacement policies [11]:

- **Random** replacement is used to spread allocation uniformly. Candidate blocks are randomly selected. It is simple to implement in hardware but ignores the principle of locality.
- **LRU** algorithm, evicts the least recently used line. The idea behind this is to keep the recently used data in the cache. Because it may be used soon, thanks to the principle of locality. All the accesses to the blocks are recorded and the replaced block is the one which has been the least recently used. It is thus very computationally expensive to implement for large caches with a large number of ways. For this reason, usually an approximation of this algorithm is implemented.
- **First In First Out (FIFO)/Round Robin** It removes block in the order they were brought in the cache, thereby taking advantage of the locality principle in a simpler way.
- **Not Most-Recently Used (NMRU)** is easier-to-implement with respect to LRU. NMRU is equal to LRU for 2-way set-associative caches.
- **Least-Frequently Used (LFU)** This algorithm keeps track of the frequency of accesses of the lines and replaces the LFU one. LFU is sometimes combined with a Least Recently Used algorithm and called LRFU.

In this thesis, we use a Pseudo-Random replacement policy for our baseline architecture, and also implement an approximation of the LRU replacement in chapter 2.

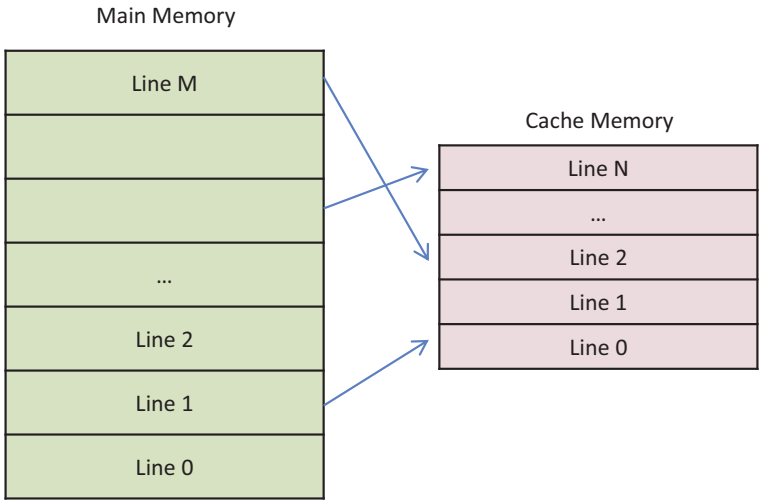

Figure 1.2: The mapping in a fully associative cache

We will study the impact of the replacement policy on the performance of different benchmarks, and also on system-level area and power consumption.

1.2.2 Block Mapping

One important question to answer is how to assign an upper level memory block to a cache line. Three mappings are usually used: direct-mapped, fully-associative and N-way-set associative.

Fully Associative Cache

The first cache organization to be discussed is Fully-Associative cache. Figure 1.2 shows a diagram of a Fully Associative cache. This organizational scheme allows any line in main memory to be stored at any location in the cache. Main memory and cache memory are both divided into lines of equal size. For example Figure 1.2 shows that Line 1 of main memory is stored in Line 0 of cache. However this is not the only possibility, Line 1 could have been stored anywhere within the cache. Any cache line may store any memory line, this is why it is called fully-associative.

One disadvantage of this scheme is the complexity of implementation which comes from having to determine if the requested data is present in cache or not. The current address must be compared with all the addresses present in the Tag array. This requires content addressable memories (CAMs) with a large number of comparators that increase the complexity and cost of implementing large caches. Therefore, this type of

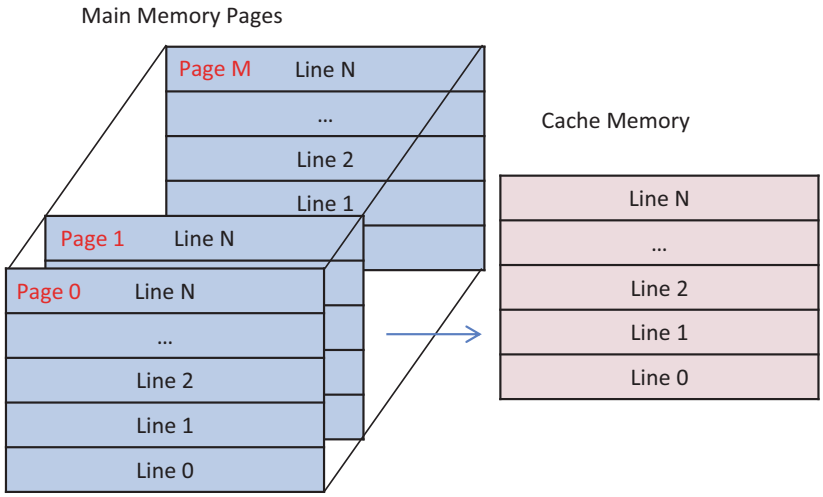

Figure 1.3: The mapping in a direct-mapped cache.

cache is usually only used for very small caches. Also, we will show in chapter 2, that fully-associative caches do not necessarily work better than direct-mapped caches for all applications. This also depends on the replacement policy.

Direct Mapped Cache

Direct-Mapped cache is also referred to as 1-Way set associative cache. Figure 1.3 shows a diagram of a direct map scheme. In this scheme, main memory is divided into cache pages. The size of each page is equal to the size of the cache. Unlike the fully associative cache, the direct map cache may only store a specific line of memory within the same line of cache. For example, Line 0 of any page in memory must be stored in Line 0 of cache memory. Therefore if Line 0 of Page 0 is stored within the cache and Line 0 of page 1 is requested, then Line 0 of Page 0 will be replaced with Line 0 of Page 1. This scheme directly maps a memory line into an equivalent cache line, for this reason it is called Direct Mapped. A Direct Mapped cache scheme is the least complex of all three caching schemes. Direct Mapped cache only requires that the current requested address be compared with only one cache address. Since this implementation is less complex, it is far less expensive than the other caching schemes. The disadvantage is that Direct Mapped cache is far less flexible making the performance much lower, especially when jumping between cache pages.

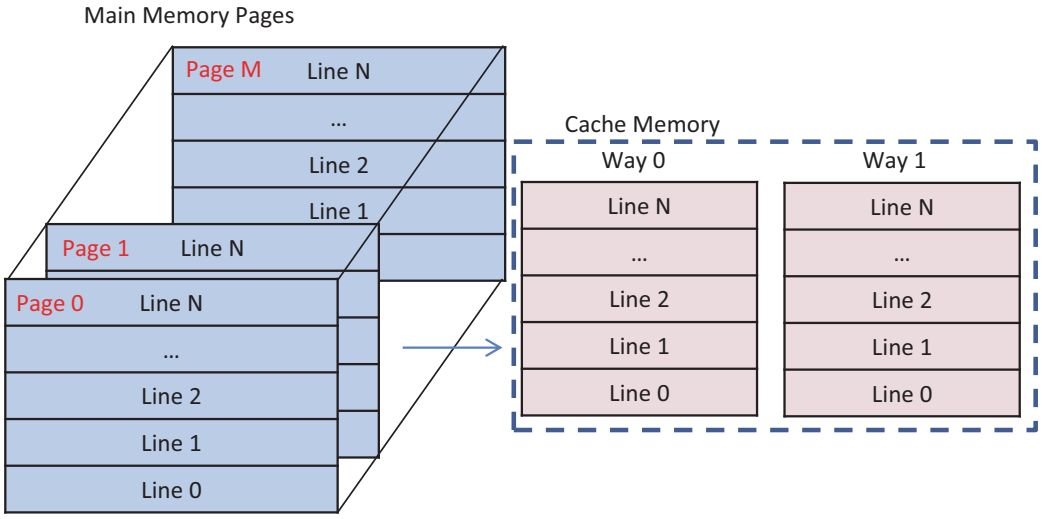

Figure 1.4: The mapping in a 2-way set-associative cache.

Set Associative Cache

A Set-Associative cache scheme is a combination of Fully-Associative and Direct Mapped caching schemes. A set-associate scheme works by dividing the cache into equal sections (2 or 4 sections typically) called cache ways. The cache page size is equal to the size of the cache way. Each cache way is treated like a small direct mapped cache. Figure 1.4 shows a diagram of a 2-Way Set-Associate cache scheme. In this scheme, two lines of memory with the same index can stored at the same time. This allows for a reduction in the number of times the cache line data is written-over. This scheme is less complex than a Fully-Associative cache because the number of comparators is equal to the number of cache ways. A 2-Way Set-Associate cache only requires two comparators making this scheme less expensive than a fully-associative scheme.

1.2.3 Different Types of Cache Misses

The cache misses can be roughly categorized in three groups [11]:

- **Compulsory Misses:** These are the misses on a cold start. Data must be fetched at least once from the lower level memory to be present in the cache. These misses are not removable by just increasing the cache size or associativity, and some sort of prefetching mechanism is required to completely eliminate them.

- **Capacity Misses:** These misses occur when the working set (data/instructions required for the program) exceeds the cache size. When the working set cannot

be contained in the cache, useful values evict one another from the cache. They can be avoided by increasing the number of cache lines or by enlarging the size of the lines in the cache. However, increasing the size of the lines without modifying the cache size leads to more conflict misses. Also, extending the size of the cache leads to more power consumption and more area. These solutions have thus strong negative impact on key points of embedded systems. Moreover, the size of the cache cannot be continuously increased because it makes the cache access time longer. In this thesis, we will show that prefetching can also help reduce these misses.

- **Conflict Misses:** These misses result from the mapping of two different items to the same cache line. Usually increasing associativity can help reduce conflict misses, but this also depends on the replacement policy. Plus, increasing the associativity of the cache can be problematic. Because apart from the increase in area, it also requires many power consuming lookups in parallel. This is a key point in embedded processors. For this reason, a proper choice of replacement policy and associativity is crucial.

1.3 An Overview of Prefetching

Prefetching is a mechanism to speculatively move data to higher levels in the cache hierarchy in anticipation future use for this instruction/data. Prefetching can be done in hardware, software, or a combination of both [19]. Software prefetching is directly controlled by the program or the compiler and therefore it is their responsibility to issue proper prefetch requests at the right time. Hardware prefetching is the alternative case, where a hardware controller generates prefetch requests from information it can obtain at run-time (e.g., memory reference and cache miss addresses). Generally, software prefetchers use compile-time and profiling information while hardware prefetchers use run-time information. Both have their advantages and both can be very effective [7]. Prefetching reduces the cache miss rate because it eliminates the demand fetching of cache lines in the cache hierarchy [20]. It is also called a latency hiding technique because it attempts to hide the long-latency transfers from lower levels to higher levels of the memory hierarchy behind periods of time during which the processor executes instructions.

A central aspect of all cache prefetching techniques is their ability to detect and predict particular memory reference patterns. Prefetching must be done accurately

and early enough to reduce/eliminate both miss rate and miss latency. There are four basic questions which need to be answered:

- **What addresses to prefetch:** Prefetching useless data wastes resources and consumes memory bandwidth. Prediction can be based on past access patterns or by using the compilers knowledge of data structures. Nevertheless prefetching algorithm determines what to prefetch.

- **When to initiate a prefetch request:** If prefetching is done too early then prefetched data might not be used before it is evicted from storage. On the other hand if prefetching is done too late it might not hide the whole memory latency. This is defined by the timeliness of the prefetcher. Prefetcher can be made more timely by making it more aggressive (try to stay far ahead of the processors access stream (hardware) or moving the prefetch instructions earlier in the code (software) [7].

- **Where to place the prefetched data:**

  Prefetched data can be placed inside the cache or in a separate prefetch buffer. If it is placed inside the cache it will have a simple design, however it can cause cache pollution and if a separate prefetch buffer is designed, demand data will be protected from prefetches so there is no cache pollution. However the design is more complex and costly. These complexities include how to place the prefetch buffer, when to access the prefetch buffer (parallel vs. serial with cache), when to move the data from the prefetch buffer to cache, and how to size the prefetch buffer.

- **How to do prefetching:** Prefetching can be performed in hardware, software, or as a cooperation of both. Also, it can rely on statically profiling the application and analyzing its patterns, or it can be dynamic. In this section we will introduce the general concepts of how to perform prefetching, and then later in section 1.4 we will present the related works in this area.

1.3.1 Software-based Prefetching

Software prefetching provides facilities for the programmer/compiler to explicitly give prefetch requests whenever they want. This can be done either by including a fetch instruction in a microprocessors instruction set, or through some registers configurable

and programmable by software. Software prefetching can be either done directly by the programmer (e.g. in the C code), or by the compiler in the optimization phase, and on the final assembly code.

Choosing where to place a prefetch instruction relative to the corresponding instruction is known as prefetch scheduling. Although, software prefetching can use more compile-time information for scheduling than the hardware techniques, it is not sometimes possible to make exact predictions. Because the execution time between the prefetch and the matching instructions may vary, as will memory latencies. If the compiler schedules fetches too late, the data/instruction will not be in the cache when CPU needs it. If the fetch occur too early, cache may replace that block for a new prefetch. Early prefetches might also replace the data that CPU is still using and this will cause a miss that would not have occurred without prefetching, which is called cache pollution.

Focusing on instruction prefetching, prefetch instructions can be manually inserted right before function calls, to prefetch them completely and make sure that they are available inside the cache before they start to execute. This can be highly beneficial for codes which have too many function calls or calls to external libraries. Of course, this requires a prior profiling of the application and identification of the address and size of the codes blocks and different functions. On the other hand, using explicit fetch instructions may also bring some performance penalties because the code size is increasing by addition of the prefetch instructions. Also each prefetch command might take more than one cycle to complete depending on how it is implemented. For this reason, it is important to optimize the location and size of the prefetch commands to make sure the optimal performance is achieved [21].

To summarize, software prefetching gives the programmers control and flexibility, and allows for complex compiler analysis and profiling of the applications. Also, it does not require major hardware modifications. But on the other hand, it is not very easy to perform timely prefetches, and prefetch instructions can increase the code footprint. For this reason, extensive profiling is needed beforehand.

1.3.2 Hardware-based Prefetching

Hardware based prefetching is typically accomplished by having a dedicated hardware mechanism in the processor that watches the stream of instructions or data being requested by the executing program, recognizes the next few elements that the program might need based on this stream, and prefetches them into the processor's cache [22].

Hardware monitors the memory access pattern of the running program and tries to predict what data the program will access next and prefetches that data/instruction. Then it memorizes the patterns/strides of the application and so it will generates prefetch addresses automatically. There are few different variants of how this can be done.

Sequential Prefetching

Sequential prefetching can take the advantage of spatial locality by prefetching consecutive smaller cache blocks, without introducing some of the problems that exist with large blocks.

Next-line prefetching (one-block look-ahead) is the simplest form of instruction prefetching [23]. In this scheme, when a cache line is fetched, a prefetch for the next sequential line is also initiated. One way to do this is called the prefetch-on-miss algorithm [24], in which the prefetch of block b+1 is initiated whenever an access for block b results in a cache miss. If b+1 is already cached, no memory access is initiated. Next-N-line prefetch schemes extend this basic concept by prefetching the next N sequential lines following the one currently being fetched by the processor [25]. The benefits of prefetching the next N-lines include, increasing the timeliness of the prefetches, and the ability to cover short non-sequential transfers (where the target falls within the N-line "prefetch-ahead" distance). Also this method is simple to implement and there is no need for sophisticated pattern detection. This scheme works well for sequential/streaming access patterns. For simplicity, throughout this thesis, we refer to both these methods as next-line-prefetching (NLP).

A **stream prefetcher** looks for streams where a sequence of consecutive cache lines are accessed by the program. When such a stream is found the processor starts prefetching the cache lines ahead of the program's accesses [26]. Again, this method is simple to implement and as we will show in chapter 3, it can be designed as an extension to the basic next-line prefetchers.

A **stride prefetcher** looks for instructions that make accesses with regular strides, that do not necessarily have to be to consecutive cache lines. When such an instruction is detected the prefetcher tries to prefetch the cache lines ahead of the access of the processor [27]. It should be noted that stream prefetching can be considered as a special case of stride prefetching (with stride of 1). Also, strides >1 does not apply to instruction caches and is only useful for data. For this reason in this work we do not focus on stride prefetchers.

Non-sequential Prefetching

In many applications cache misses occur because of transitions to distant lines, especially when the application is composed of small functions or there are frequent changes in control flow. There are some kinds of prefetchers specifically targeted at non sequential misses. Target-line prefetching, for example, tries to address next-line prefetchings inability to correctly prefetch non sequential cache lines. It uses a target prefetch table maintained in hardware to supply the address of the next line to prefetch when the current line is accessed [8][23]. Also hybrid schemes can be built by combining different prefetching techniques. For example, in a combination of next-line and target prefetching both a target line and next line can be prefetched, offering double protection against a cache line miss [8]. Finally, a combination of hardware and software prefetching mechanisms can be implemented [28] to benefit from both of their capabilities. In the next section we will explain the state-of-the-art prefetching techniques in more details.

1.4 Related Works

The simplest form of prefetching can be considered to have **Long Cache Lines** [29]. When an instruction cache miss occurs, more than one instruction is brought into the cache as a (long) cache line. So, probability that the next instruction needed is in the cache increases. This in turn results in a reduction in the number of cache misses. However this method increases memory traffic as well as cache pollution. Cache pollution increases because many lines may only be accessed partially before it is displaced. The choice of the length of cache lines depends on the locality and sharing property of programs as well as available memory bandwidth. Programs with good spatial locality usually benefit from using longer cache lines as most of the data in the cache line is likely to be used before it is invalidated [24]. In addition to the properties of the program, the length of the cache line is also determined by the available memory bandwidth. This is because as length of cache line increases, the width of the memory bus also has to increase.

Another approach to instruction prefetching is **next-line prefetching** [23], as introduced before. The sequential prefetch or next-line prefetch is a simple but effective design that easily exploits spatial locality and sequential access. As long as the code is sequential and the prefetch distance is sufficient this method will completely hide the memory latency [28]. But this method cannot not handle non-sequential cases

(conditional/unconditional branches and function calls) [7], because it predicts that execution will "fall-through" any conditional branches in the current line and continue along the sequential path. The scheme requires little additional hardware since the next line address is easily found and has been shown effective reducing cache misses by 20-50% in some cases [23]. In chapter 3 we will show that a combination of this simple mechanism with software-prefetching can effectively remove most of the misses in various applications.

Target-line prefetching tries to solve next-line prefetchings inability to correctly prefetch non sequential cache lines. Target-line prefetching uses a target prefetch table maintained in hardware to supply the address of the next line to prefetch when the current line is accessed. The table contains current line and successor line pairs. When instruction execution transfers from one cache line to another line, two things happen in the prefetch table. The successor entry of the previous line is updated to be the address of new current line. Also, a lookup is done in the table to find the successor line of the new line. If a successor line entry exists in the table and that line does not currently reside in the cache, the line is prefetched from memory [8][23]. Performing target prefetching with the help of a prefetch target table has some disadvantages. First, significant hardware is required for the table and the associated logic which performs the table lookups and updates. This uses additional chip area and could increase cycle time. Second, the extra hardware has only limited benefit. Table-based target prefetching does not help first-time accessed code since the table first needs to be set up with the proper links or current-successor pairs. Thus compulsory misses are unaffected by target prefetching. Finally, being able to prefetch the target of a control instruction means that the effective-address of the control instruction has to be known, even before the branch instruction is executed. This can be very costly in terms of logic delay and area.

Another prefetching mechanism proposed recently is called **Hybrid prefetching** [8]. This mechanism is a combination of next-line and target prefetching. In this method both a target-line and next-line can be prefetched, offering double protection against a cache line miss. Next-line prefetching works as previously described. Target-line prefetching is similar to that above except that if the successor line is the next sequential line, it is not added to the target table. This saves table space thus enabling the table to hold more non-sequential successor lines. The performance gain of the hybrid method is roughly the sum of the gains achieved by implementing next-line and target prefetching separately, but again, the hardware cost of this mechanism is

significant because of the target-line prefetcher maintained in hardware [8].

Wrong-path prefetching [8] is similar to the hybrid scheme in the sense that it combines both target and next-line prefetching. The major difference is in target prefetching. No target line addresses are saved and no attempt is made to prefetch only the correct execution path. Instead, in the simplest wrong-path scheme, the line containing the target of a conditional branch is prefetched immediately after the branch instruction is recognized in the decode stage. So, both paths of conditional branches are always prefetched: the fall-through direction with next-line prefetching, and the target path with target prefetching. Unfortunately, because the target is computed at such a late stage, prefetching the target line when the branch is taken is unproductive. A cache miss and a prefetch request would be generated at the same time. Similarly, unconditional jump and subroutine call targets are not prefetched since the target is always taken and the target address is produced too late. The target prefetching part of the algorithm can only perform a potentially useful prefetch for a branch which is not taken. But if execution returns to the branch in the near future and the branch is then taken, because of the previous prefetch, the target line will probably reside in the cache. The obvious advantage of wrong-path prefetching over the hybrid algorithm is that no extra hardware is required above that needed by next-line prefetching. All branch targets are prefetched without regard to predicted direction and the existing instruction decoder computes the address of the target. The main problems with this prefetching approach are the large amount of extra traffic generated, and the cache pollution. Some variations to the basic idea of wrong-path prefetching have been proposed in [8] to address these issues, however, they all come at the cost of increased complexity.

An alternative solution to caching and prefetching is proposed in [30][31] as the **loop-buffer**. The loop buffer is a small buffer used to store a number of recently executed instructions in a FIFO fashion. If there is a loop in the code the recent instructions will be executed again. If all the instructions of the loop fit inside the loop buffer, all required instructions will be in the loop buffer after the first iteration. So for the other iterations all instructions will be fetched from the loop buffer and not from the memory. This approach, however, is limited to small loops, and also is not applicable to nested loops and function calls.

A more complex prefetching mechanism is the **Markov prefetcher** [9] which consists of a prediction table and a prefetch queue. The predictor operates by listening to the cache miss stream. When a cache miss occurs, the predictor adds an entry into

the prediction table. The next couple of cache misses that occur are added as the prediction for the previous address. The exact number of addresses that are added to the table can be varied. These addresses form the prediction and are prefetched into memory when the corresponding miss address is referenced by the processor. The prefetcher assumes that the prediction addresses will be referenced shortly after the miss address is referenced. This process allows the predictor to improve caching by discovering reference patterns. The Markov predictor is able to improve cache performance, but there are several problems with the design. The most important problem is that the predictor has a large learning phase; it must wait for two cache misses before an entry is added to the table. A block is not prefetched until one of the missed addresses is referenced again. Another problem with the predictor is deciding when to add entries to the table and selecting the appropriate number of prediction addresses. If there are too many predictions, then the predictor will be less accurate and cause more cache pollution [7].

Lastly, **Software prefetching** [32][33] is based on the use of some form of explicit fetch instruction. Simple implementations may simply perform a non blocking load (perhaps into an unused register), while more complex implementations might provide hints to the memory system as to how the prefetched blocks will be used. Very little hardware needs to be introduced to take advantage of software prefetching. The difficulty in efficiently using this approach lies in the correct placement of the fetch instruction. The term prefetch scheduling refers to the task of choosing where to place the fetch instruction relative to the accompanying load or store instruction. Uncertainties that cannot be predicted at compile time, such as variable memory latencies and external interrupts, make it more difficult to precisely predict where in the program to position a prefetch so as to guarantee that a block arrives in the cache when it is required by the processor. It is possible to gain significant speed advantages by inserting a few fetch instructions manually in strategic portions of the program [34].

With software prefetching, the user can insert prefetch requests independently for different streams and blocks, while it is difficult for hardware prefetchers that are typically easily confused when there are too many blocks. Also, hardware prefetchers require training time to detect the direction and distance of a stream or stride. If the length of block is extremely short, there will not be enough cache misses to train a hardware prefetcher and load useful cache blocks. This problem does not exist in software prefetchers. Finally, software prefetching allows for prefetching complex access patterns and function calls. While complex hardware mechanisms are needed

to implement this in hardware. One interesting approach is to combine both hardware and software prefetching to be able to take advantage of both mechanisms [28][35]. In this thesis we choose a similar approach and show that our proposed mechanism can effectively remove most of the cache misses.

1.5 Contribution of this Thesis

In this work, we focus on ultra-low power multi-core architectures, and try to improve their instruction cache performance by the aid of prefetching. We study a combination of low-cost hardware and software based mechanisms, and propose a combined approach based on a simple Software Prefetcher (SWP), a Next-line Prefetcher (NLP), and a more intelligent Stream Prefetcher (STP). We will show that these mechanisms can effectively eliminate most of the cache misses or reduce the miss penalties of a wide range of applications, and can give significant performance gains.

This thesis is organized as follows: in chapter 2 we will introduce our methodology and baseline setup. We choose the PULP platform [13] as an ultra-low power parallel processing platform and present the baseline performance results for a wide range of benchmarks through Cycle-Accurate simulations. Then in chapter 3, we discuss our proposed prefetching mechanisms and explain about the required changes in the baseline PULP platform. In chapter 4 we study the performance impact of our proposed instruction prefetching mechanisms, and in chapter 5 we show their impact on silicon area, power consumption, and energy in the 28nm FDSOI technology by STMicroelectronics. Finally in chapter 6, we summarize our achievements and conclude the thesis.

Chapter 2

Methodology and Setup

This section describes the baseline setup and methodology in this thesis. First we will show our hardware configuration, then we show our benchmarks and evaluation methodology.

2.1 Hardware Configuration

The cycle-accurate RTL model of PULP platform has been used as the baseline model [13]. PULP is a multi-core platform achieving high energy-efficiency and widely-tunable performance, targeting the computational demands of the Internet-of-Things (IoT) applications [36] which require flexible processing of data streams generated by multiple sensors. As opposed to single-core micro-controllers, a parallel ultra-low-power programmable architecture allows for meeting the computational requirements of these applications, without exceeding the power envelope of a few mW typical of miniaturized, battery-powered systems.

An overview of a processing cluster in the PULP platform is illustrated in Figure 2.1. As can be seen, a cluster is formed by multiple RISC-V processors, a shared instruction cache, and a multi-ported tightly coupled data memory (TCDM). The instruction cache has been previously designed in [18]. It is a shared ICache System based on private cache-controllers and shared DATA/TAG banks. It is flexible and parametric and it has been designed based on standard cell memories. This is shown in Figure 2.2. As can be seen this cache has multiple ports which can connect to processor cores and it has another port which can connect it to the out-of-the-cluster L2 memory.

Whenever a fetch request comes from one of the processors the private controller associated with that core checks if it is a Hit or a Miss by looking at its TAG array. If

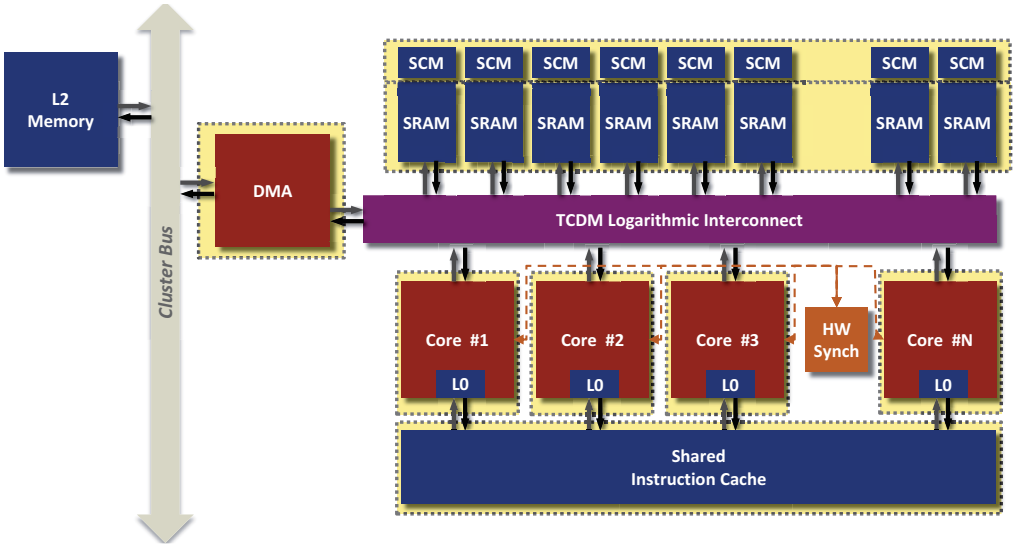

Figure 2.1: An overview of a processing cluster in the PULP platform.

a Hit happens the private controller reads its data and returns it to the processor in 1 Cycle. But if a miss happens, private-controller sends a miss request to Master Cache Controller. The miss requests from different private-controllers are arbitrated in the Logarithmic Interconnect (LINT) shown in Figure 2.2. Inside the Master Controller a hardware structure called Merge Refill merges these request to make sure there are no duplicate requests and then a refill request is sent to the L2 memory. When the response comes back after several cycles, Master Controller writes it back to the DATA array and validates its TAG. Then it notifies the private controllers that their data is ready.

This ICache is N-way set-associative where N is a configurable parameter. For example in Figure 2.2 associativity is 4. Also the size of the cache is a parameter ranging from 512B to 16KB. For the baseline configuration we consider a size of 1KB with an associativity of 2. The default replacement policy is Pseudo-Random and Cache block size is 16B, equal to another prefetch buffer (called L0 buffer) inside the RISC-V cores. The multiple banks shown in Figure 2.2 does not have any performance impact and has only implementation benefits. In total, our baseline cluster has 4 RISC-V cores connected to 8 SRAM banks forming a Tightly-coupled-data-memory (TCDM) of 16KB. We will use this configuration throughout this thesis and study the effect of prefetching on it.

Figure 2.3 shows a more simplified block diagram of the ICache shown in Figure 2.2. We will use this simplified block diagram throughout this thesis.

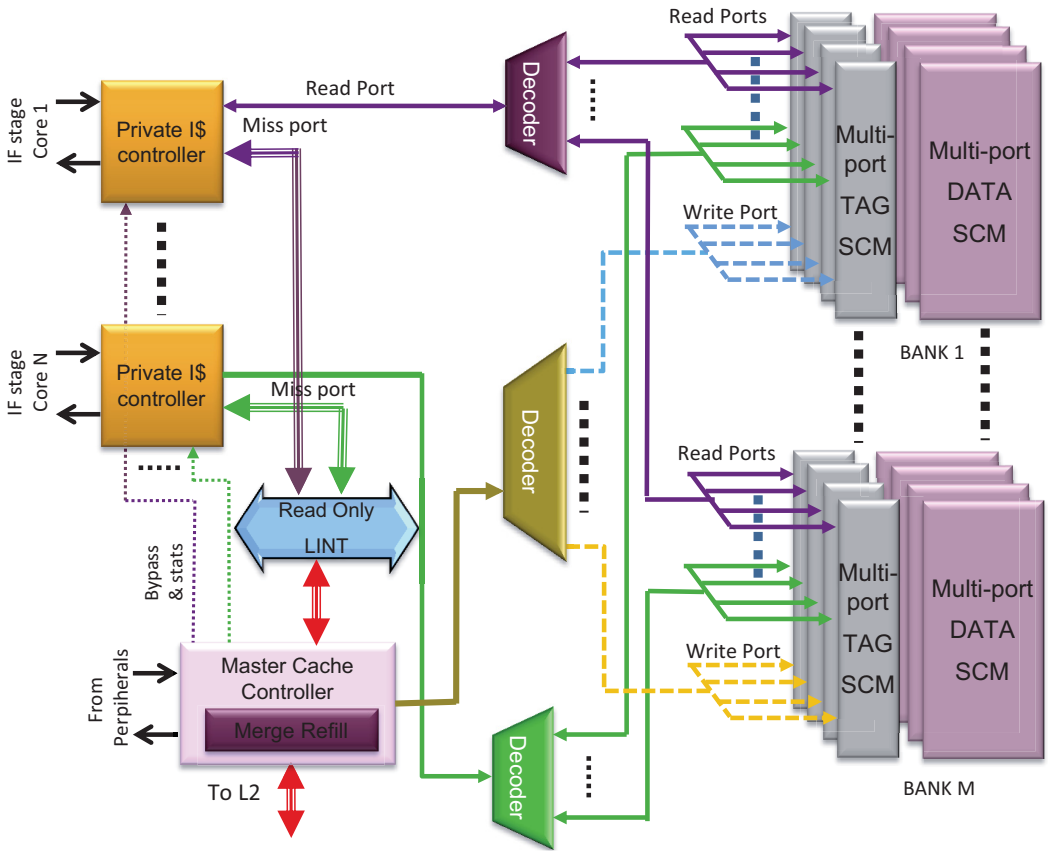

Figure 2.2: Detailed block diagram of the multi-ported shared instruction cache in PULP.

2.1.1 Replacement Policy

As mentioned before, in the baseline ICache the replacement policy is "Pseudo-Random". Whenever a refill request happens, if at-least one of the ways in one index is free, it will be used for the new block. But if all ways are full, one of them is selected randomly and replaced with the new block. One problem with random-replacement policy is that it may randomly evict useful blocks too, resulting in multiple refill requests for blocks which have already been present in the cache. However, this does not happen for a Least Recently Used (LRU) implementation. We will show this phenomenon through an example in section 2.4.

In general, implementation of true Least Recently Used (LRU) replacement policy is not easy and has very high hardware cost. Even its approximated version called Pseudo LRU (PLRU) can have a high hardware cost for large caches with a high associativity. However, a simple calculation can show that in our case, implementation of PLRU is not very costly. This is because our baseline cache is 1KB with an associativity of 2,

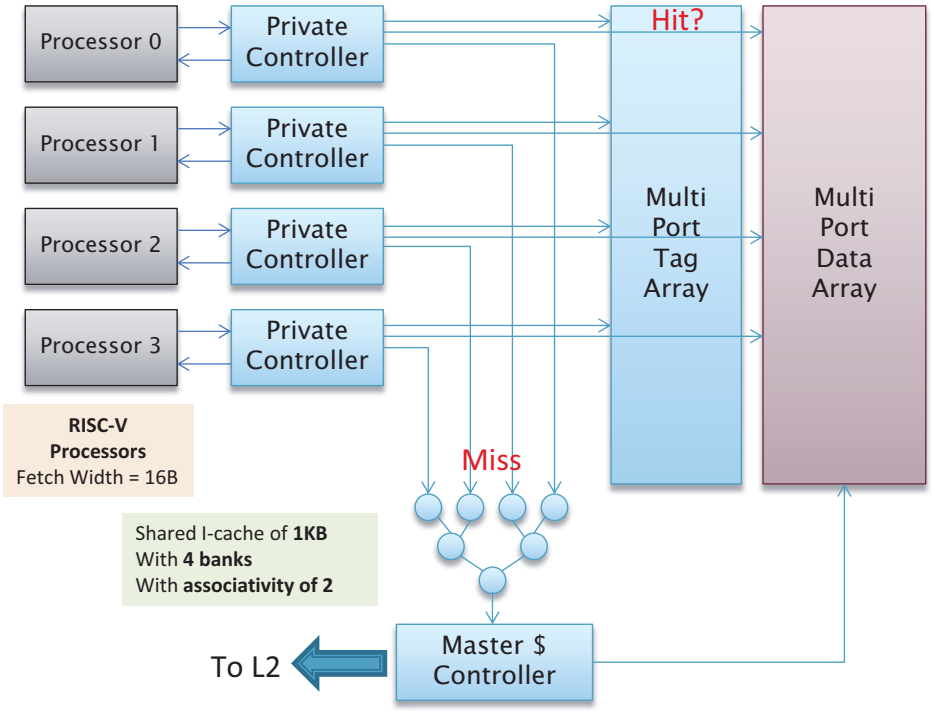

Figure 2.3: Simplified block diagram of the multi-ported shared instruction cache in PULP.

and each cache block is 16 Bytes. So there are a total of 32 cache rows. But since in each row there are only two ways, one additional bit is enough to identify the LRU way. This means that using 32 bits of registers we can identify the LRU way inside each row. This is shown in Figure 2.4.

Whenever a way is accessed, the LRU bit is changed accordingly to show that this way is not the LRU. And when a replacement needs to happen, the LRU bit indicates which way is the victim. But since our ICache is shared among 4 processors, each core should be able to access, read, and update all entries. This complicates the write logic for an ideal LRU implementation. Because in the same cycle for example, 2 core may want to access Way0, another core accesses Way1, and the last core tries to replace Way0. In this case it is not clear which way should become the new LRU. Different combinations of all these cases can significantly complicate the write logic.

In this work we used a heuristic and implemented Pseudo-LRU instead of LRU as follows: If in the same cycle, all processors access Way1, then the LRU is Way0, but if any of them accesses Way0, then the LRU is Way1. This simple heuristic reduces the number of gates to two four-input and/or gates. We will show later in chapter 5 the effect of this replacement policy on overall performance, hardware cost, and power

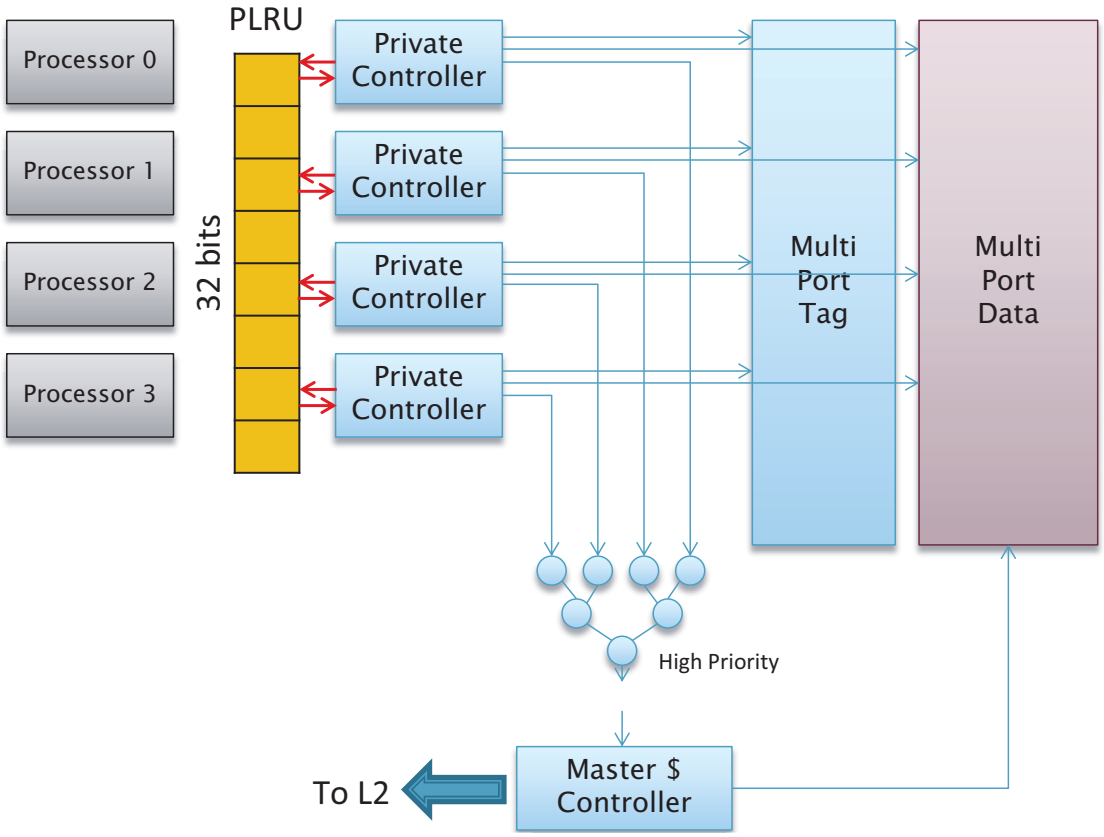

Figure 2.4: An overview of the PLRU counters added to the baseline cache.

consumption. One last point to mention is that, in the next chapters we design different instruction prefetchers and add them to the baseline ICache. This does not have much effect on the replacement policy, except that fan-in of its gates will increase from 4 to 5.

2.2 Gathered Statistics

In order to quantify the benefits of the proposed ideas, we measure different statistics inside the cycle-accurate simulation of the PULP platform and report them in this thesis. These are the statistics that we gather:

2.2.1 Average Hit-rate

For each processor we measure the total number of accesses to the multi-ported ICache and the percentage of them which result in a hit. In the end we take an average of the

hit-rates and report them in (%).

2.2.2 Memory Access Time (MAT)

We define memory access time as the number of cycles it takes for the ICache to respond to the request issued by the processor cores. This is measured in (Cycles) from the moment that one processor gives a fetch request, to the moment that it receives back the response from the ICache. If a hit happens, this latency is 1-cycles, but if a miss happens, it can take around 20 cycles, because the block must be refilled from the L2 memory, which is out of the processing cluster. We also report Average Memory Access Time (AMAT) in (Cycles) as the average of all MATs.

2.2.3 Miss Traffic

We measure the bandwidth on the AXI bus connecting the cluster to the L2 memory in MB/sec. We consider this as an indicator of the miss traffic. Because an application with 100% hit-rate results in no transfer from the L2 memory (Miss Traffic = 0), but as the hit-rate decreases, the bandwidth on this bus increases. The data-width of the AXI interconnect in our platform is 8-Bytes and it has a clock period of 20ns. For this reason, the L2 bandwidth can be at maximum 400MB/sec.

2.2.4 Total Cache Usage

In the end of the simulation we count the number of used cache blocks and report them in Bytes. This metric can show how large the footprint of the application under study is. If the application footprint is small it does not completely use the cache, but if it is too large it fills up the cache and results in many replacements.

2.2.5 Total Execution Time

We also measure the total execution time of the application in Cycles. This stat is measured only for the interesting part of the applications (their main computation function). Two commands from software GATHER_CACHE_STATS and REPORT_CACHE_STATS trigger the start and stop of the execution-time, and also all other statistics are gathered only between these two commands. This is an example of how these commands should be used in software:

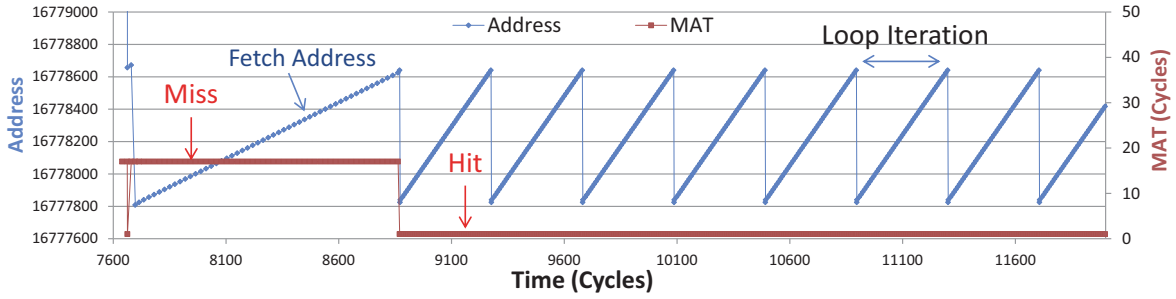

Figure 2.5: The Address and MAT plotted together for an application with a single loop.

```
Initializations();
GATHER_CACHE_STATS;
run_function();
REPORT_CACHE_STATS;
```

2.2.6 The Address and MAT Plot

To understand how different benchmarks behave we plot their memory access patterns and MATs over time in a single plot. An example of this is shown in Figure 2.5 for a single loop. On the left axis, the fetch address of the RISC-V processor is shown and plotted in blue color. While on the right axis, Memory Access Time (MAT) is shown in Cycles. The horizontal axis represents time in Cycles. It can be seen that the first iteration takes longer to complete because many compulsory misses happen, while the next iterations run faster.

In the next section we describe the benchmarks we used for evaluating our proposed methods.

2.3 Studied Benchmarks

Our main focus in this thesis has been on studying the effect of instruction prefetching on performance and power consumption of the PULP platform. For this reason we looked at various codes with different instruction memory access patterns and different characteristics. We group the studied benchmarks into two categories and briefly describe each of them in this section. In this section, we use following system configuration for all our experiments:

```
Cache Size:  1KB
```

```
Associativity:  2
Replacement:  PLRU
```

2.3.1 Group 1: Benchmarks with Large Loop Bodies

These benchmarks either have large loop bodies on their own, or their loop size can be increased by the help of loop-unrolling [37] and function-inlining [38]. These mechanisms also allow for improved instruction scheduling and better compiler optimizations. It will be later shown that the benchmarks in Group 1 benefit more from the hardware prefetching mechanisms proposed in this work. Here is a brief description of these benchmarks:

SHA-1 Hash

SHA-1 (Secure Hash Algorithm 1) is a cryptographic hash function that produces a 160-bit (20-byte) hash value from a given message. The address plot of this application is shown in Figure 2.6 as *sha1*. This code has a very large loop itself.

AES Cryptography

Advanced Encryption Standard (AES) is a symmetric encryption algorithm. The algorithm has been designed to be efficient in both hardware and software, and supports a block length of 128 bits and key lengths of 128, 192, and 256 bits. The unrolled version of this code is shown in Figure 2.6 as *aes-u*.

MD5 Sum

MD5 Sum is an algorithm which calculates and verifies 128-bit MD5 hashes. The MD5 hash (or checksum) functions as a compact digital fingerprint of a file. Again this code has a very large loop and its address plot is shown in Figure 2.6 as *md5*.

LU Decomposition

In numerical analysis and linear algebra, LU decomposition (where 'LU' stands for 'lower upper', and also called LU factorization) factors a matrix as the product of a lower triangular matrix and an upper triangular matrix. The product sometimes includes a permutation matrix as well. The unrolled version of this code (for a 4x4 matrix) has the address plot shown in Figure 2.6 as *lu-u*.

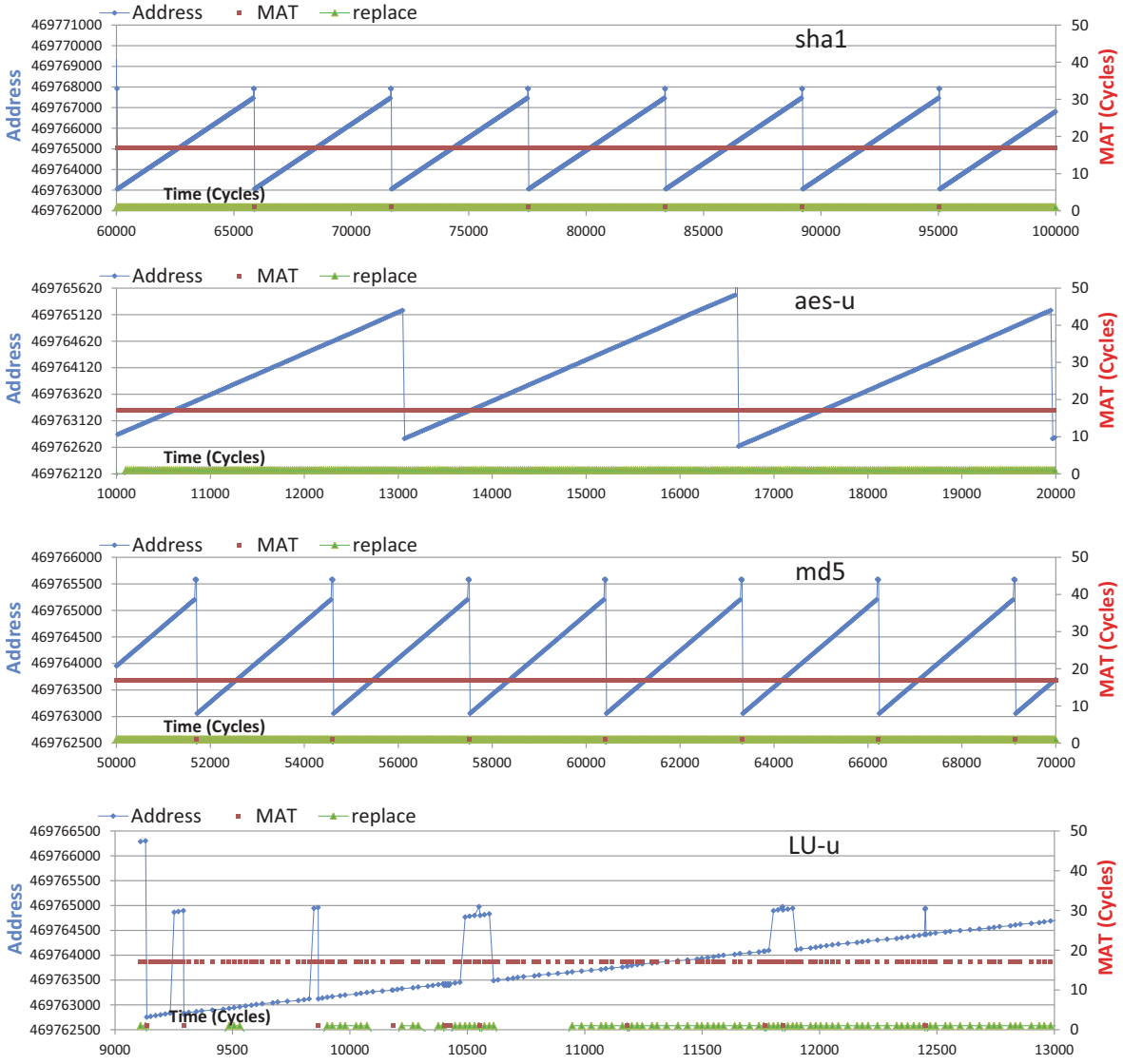

Figure 2.6: Address plot for sha1, aes-u, md5, and lu-u

Matrix Multiplication

This benchmarks is an integer dense matrix multiplication. The address plot of the unrolled version is shown in Figure 2.7 as *matrixmult-u*.

Strassen

In linear algebra, the Strassen algorithm, named after Volker Strassen, is an algorithm for matrix multiplication. It is faster than the standard matrix multiplication algorithm and is useful in practice for large matrices. This code uses many inline functions. It's address plot is shown in Figure 2.7 as *strassen*.

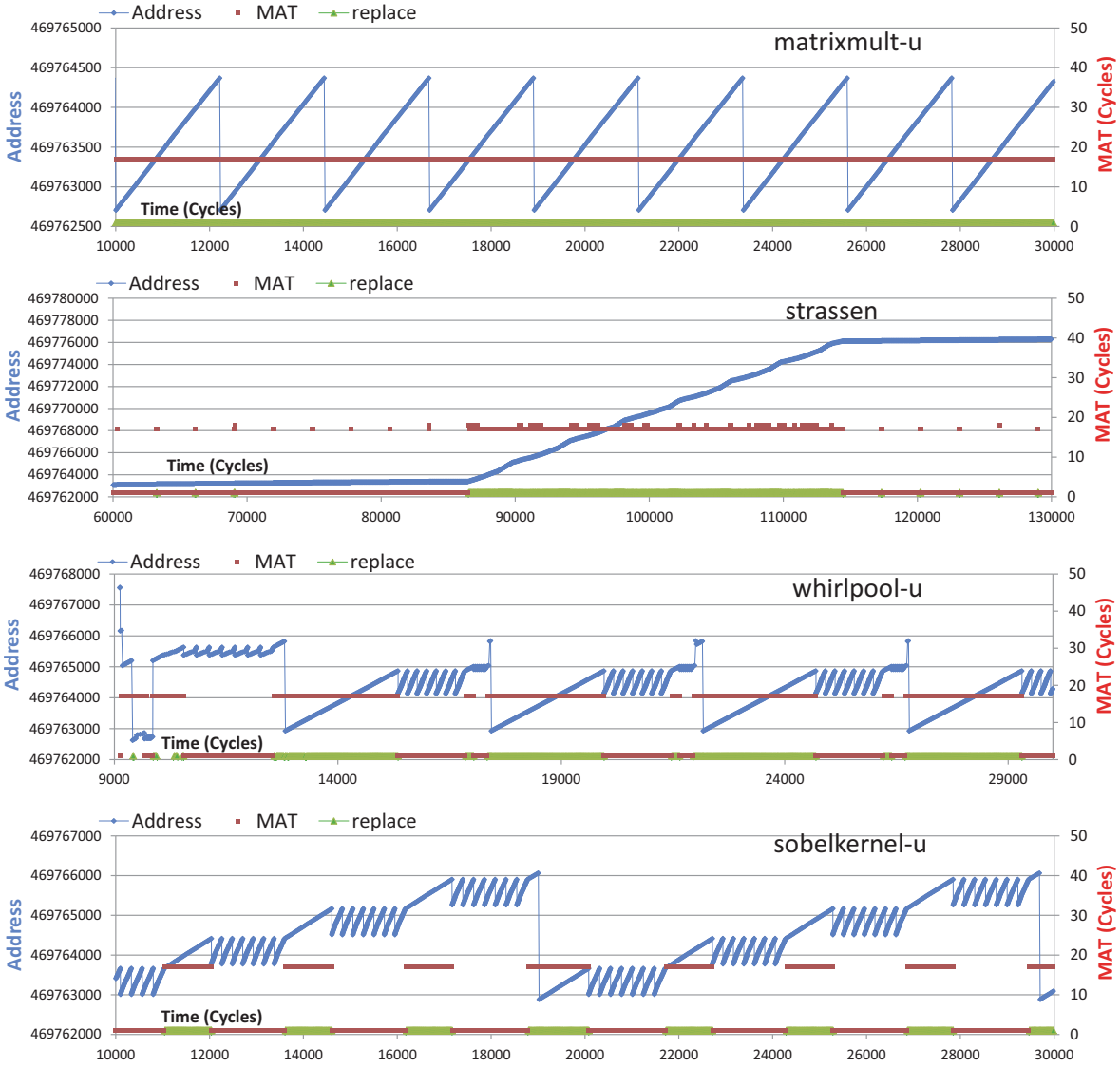

Figure 2.7: Address plot for matrixmult-u, strassen, whirlpool-u, and sobelkernel-u

Whirlpool Hash

Whirlpool is a cryptographic hash function. It was designed by co-creators of the Advanced Encryption Standard. The address plot for the unrolled version of this code is shown in Figure 2.7 as *whirlpool-u*.

Sobel Filter

The Sobel operator or Sobel filter, is used in image processing and computer vision, particularly within edge detection algorithms where it creates an image emphasizing

edges. We have unrolled its main kernel and its address plot is shown in Figure 2.7 as *sobelkernel_u*.

Convolution Kernel

A convolutional neural network (CNN, or ConvNet) is a type of feed-forward artificial neural network in which the connectivity pattern between its neurons is inspired by the organization of the animal visual cortex. In this work, we have focused only on the convolution-kernel used in the CNNs, and unrolled it. The resulting address plot is shown in Figure 2.8 as *cnnconv-u*.

Single-loop

This benchmark is a dummy single loop composed of 800 NOP instructions. This artificial benchmark has been purposefully built so that it does not fit in the instruction cache and generate many cache misses. The address plot for this benchmark is shown in Figure 2.8 as *singleloop*.

Multi-function

This is another artificial benchmark with a main loop calling four dummy functions, each of which contains a single loop with 600 NOP instructions. The address plot for this code is shown in Figure 2.8 as *multifunc*.

2.3.2 Group 2: Benchmarks with Many Function Calls

The second group of benchmarks studied in this thesis have many function calls (usually to external libraries). For example since the current version of the PULP platform does not have any hardware Floating Point Units (FPU), floating point computations should be emulated in software. This creates many function calls and results in irregular access patterns. Similarly, benchmarks with fixed-point computations require many function-calls to fixed-point computation libraries. Here is a brief description of these benchmarks:

SRAD

Speckle reducing anisotropic diffusion (SRAD) is a diffusion method tailored to ultrasonic and radar imaging applications. This benchmark uses floating-point computations and its address plot is shown in Figure 2.8 as *srad*.

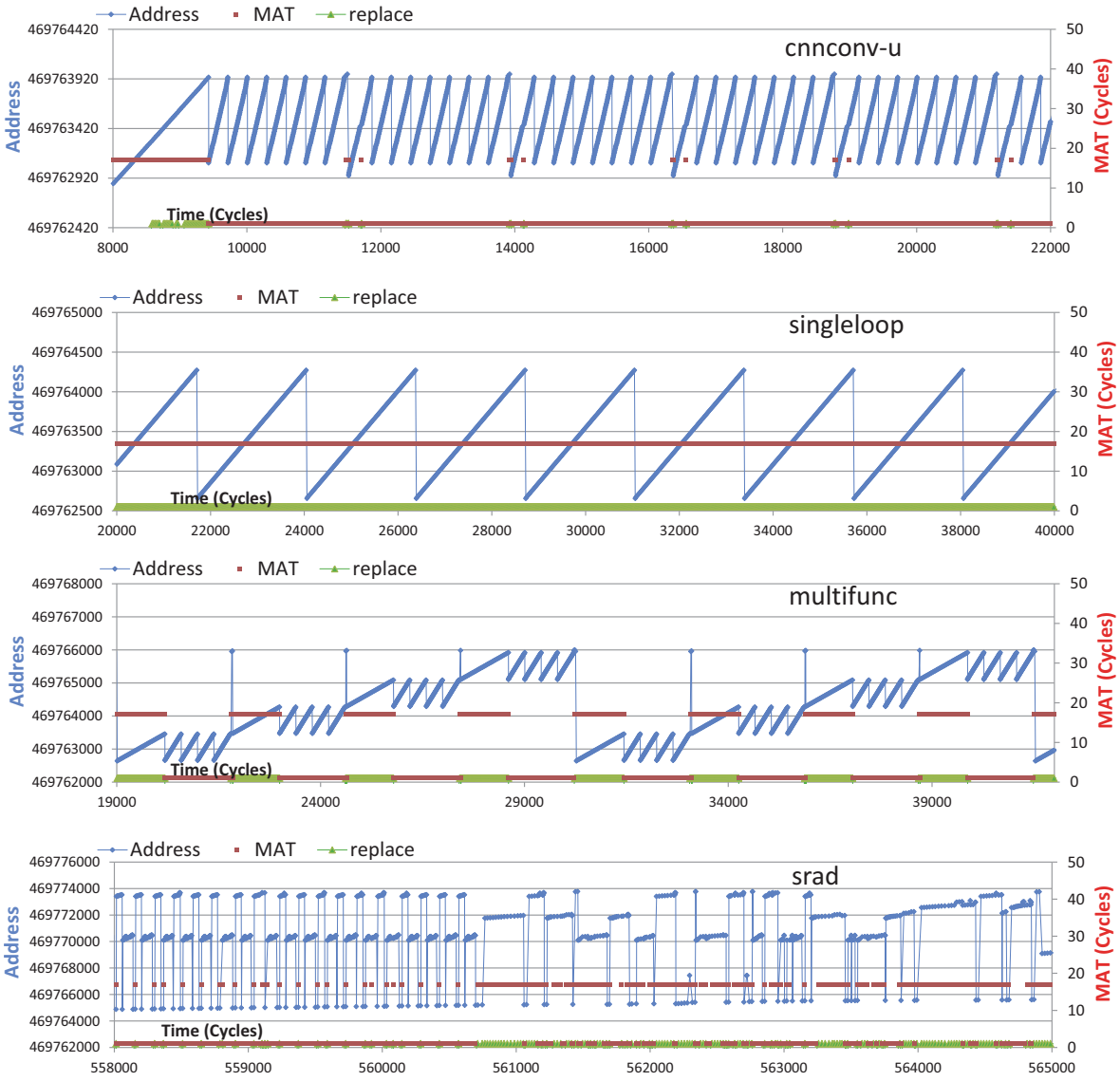

Figure 2.8: Address plot for cnnconv-u, singleloop, multifunc, and srad

Neural Network

This benchmark is a floating point implementation of a simple feed-forward network with 3 hidden layers. The address plot for this code is shown in Figure 2.9 as *neuralnet*.

FFT

Fast Fourier transform (FFT) algorithm computes the discrete Fourier transform (DFT) of a sequence, or its inverse. Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa. We

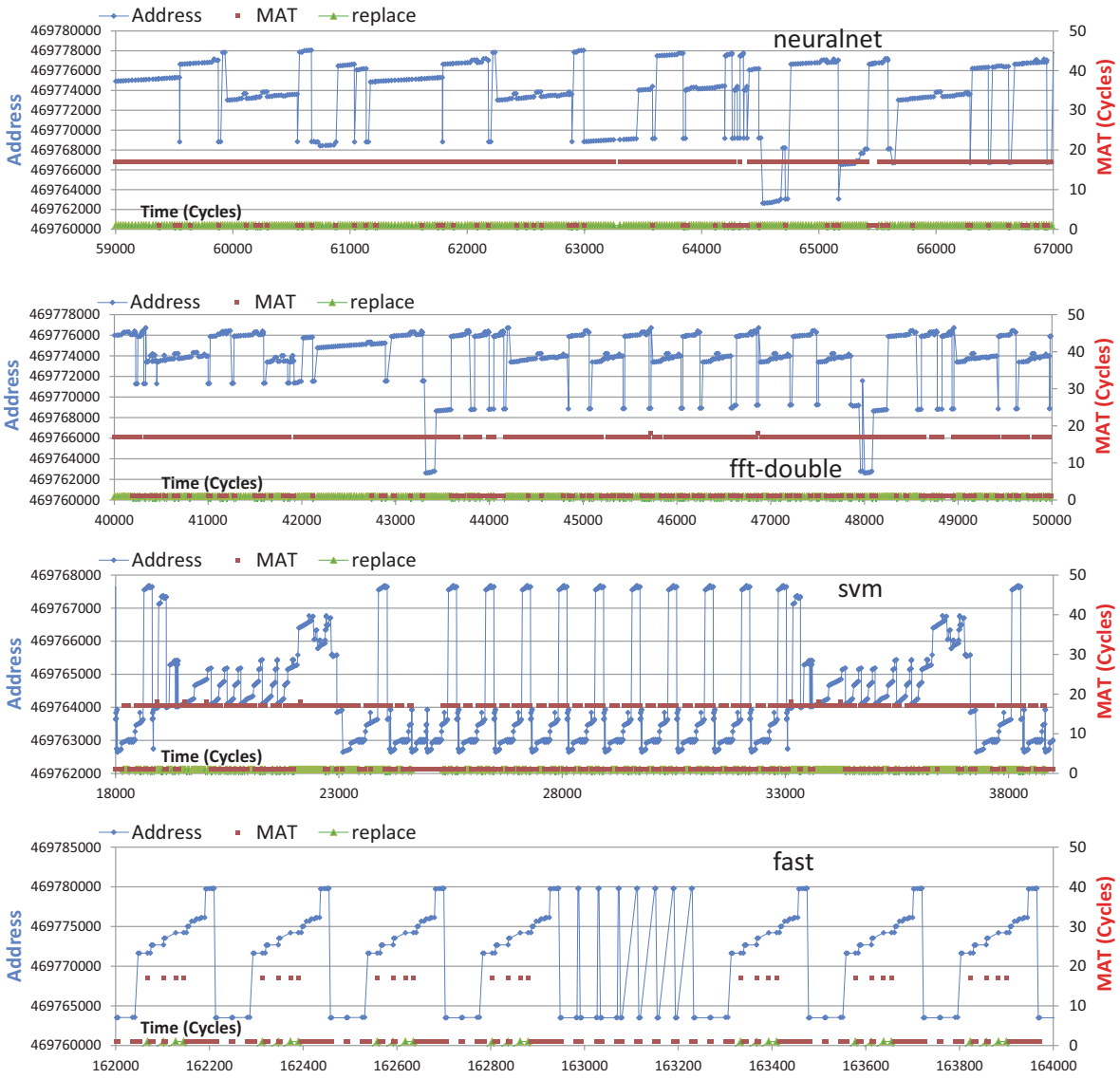

Figure 2.9: Address plot for neuralnet, fft-double, svm, and fast

have used a floating-point version of this benchmark and we show its address plot in Figure 2.9 as *fft-double*.

SVM

In machine learning, support vector machines (SVMs, also support vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. We have used a fixed-point implementation of SVMs for our studies. The address plot is shown in Figure 2.9 as *svm*.

Fast

Fast is a very efficient corner-detection algorithm in image processing. It has many conditional branch instructions which make it difficult to be used for prefetching. The address plot for this benchmarks is shown in Figure 2.9 as *fast*.

2.4 Baseline Results

In this section, we present our baseline results without any prefetching. We show the baseline statistics for all benchmarks and show the effect of architectural parameters (e.g. cache size and associativity) on their performance. In the next sections, we present our prefetching mechanisms and show their improvement. Here is a summary of the parameters used in the baseline configuration:

```
Cache Size:  1KB
Associativity:  2
Clock Period:  20ns
Replacement:  PLRU/PRAND
```

2.4.1 Effect of Replacement Policy

As the first experiment we compare the execution of *singleloop* once with PLRU and another time with PRAND replacement policy. The resulting address plot is shown in Figure 2.10. It can be seen that when the replacement is PLRU, compulsory misses happen in the first iteration, but after that all accesses turn into hit. But when the policy is PRAND, still for a couple of iterations some misses happen, even though the loop should completely fit in the cache. The reason behind this effect is that the PRAND replacement is randomly evicting blocks and it may happen that it removes the useful blocks, too. This problem does not occur with PLRU.

Next we plot average hit-rate for all benchmarks changing the replacement policy only. The results are shown in Figure 2.11. The two groups mentioned in section 2.3 are separated with a line. It is interesting to see that for some of the codes (e.g. sha1, aes, md5) PRAND is giving a higher hit-rate than PLRU. This can be explained as follows: When a code has a very large loop which does not fit in the cache, LRU replacement works poorly. This is because as the new blocks are brought to the cache they keep evicting the useful blocks which will be needed in the next iteration of the large loop. This way their hit-rate is close to zero. On the other hand, because PRAND randomly

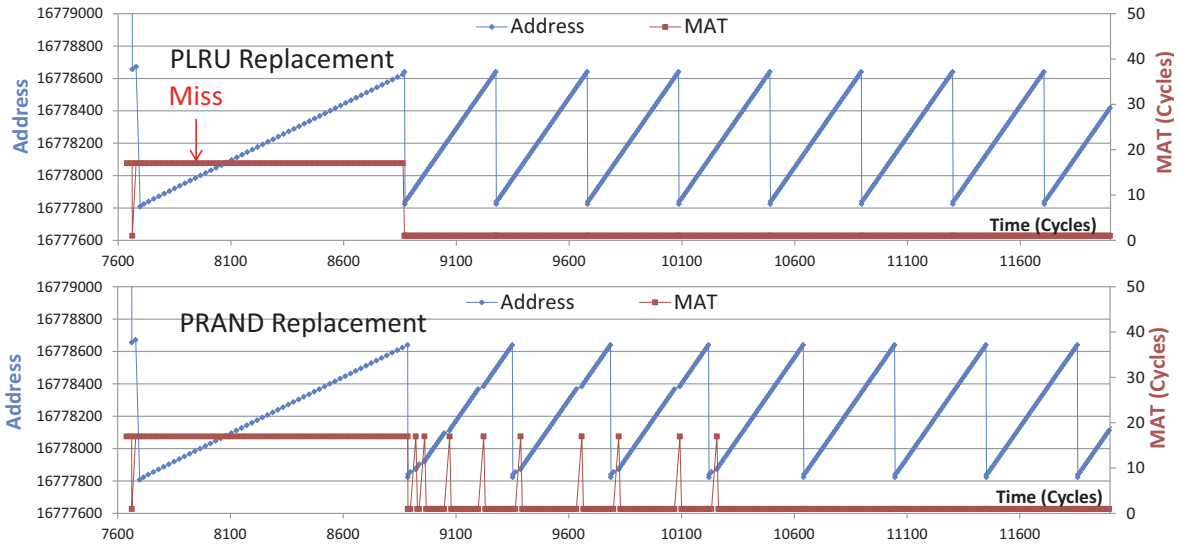

Figure 2.10: Comparison on PRAND and PLRU replacement policies for execution of the *singleloop* benchmark.

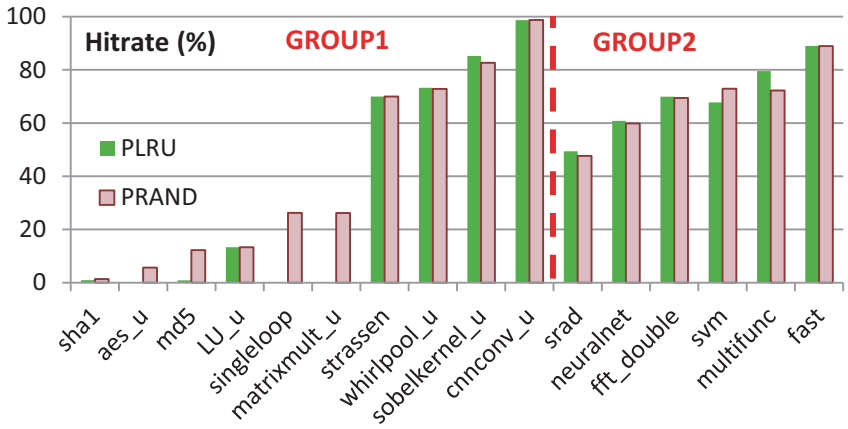

Figure 2.11: Comparison on PRAND and PLRU replacement policies for all benchmarks using the baseline hardware configuration (without prefetching).

evicts blocks, there is still a chance that some of these useful blocks are present in the cache. So they can be used in the next iteration. This way for benchmark with very large loops PRAND works better than PLRU. We will show however in the next chapters that prefetching changes this behavior. Another point to mention is that *cnnconv-u* already has a very high hit-rate (close to 100%) so probably it won't benefit from prefetching. In the next chapters we will study the benefit of different prefetching methods on these benchmarks.

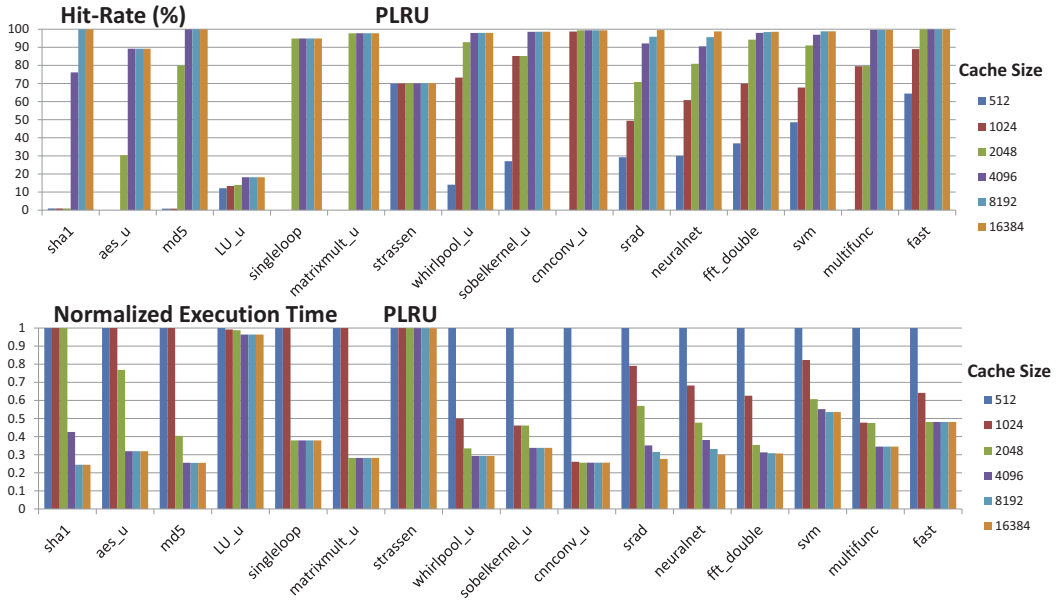

Figure 2.12: Effect of cache size on hit-rate and execution time of the benchmarks.

2.4.2 Effect of Cache Size

For the next experiment we change the size of the ICache from 512B to 16KB and study its effect on hit-rate and normalized execution time. No prefetching is implemented yet, associativity is 2, and replacement policy is PLRU. The results are shown in Figure 2.12. Something interesting in this plot is that *strassen* does not improve at all with increase in cache size. The reason is that *strassen* has many inline functions and each of these functions have very small loops. So increasing cache size has no effect on it because all misses are cold-misses (compulsory misses). A similar thing happens also for *lu-u* which is highly unrolled and was previously shown in Figure 2.6. In the next chapters we will see that only prefetcher can remove these types of misses.

Almost all other benchmarks need a large cache size (e.g. 16KB) to achieve a hit-rate close to 100% (except for *cnnconv-u* which already has a high hit-rate). The problem with large caches is higher silicon area and higher power consumption. Also large caches achieve lower clock frequencies because of having a higher access time. In the next chapters, we will introduce prefetching mechanisms which are able to work with small caches (e.g. 1KB) and still achieve very high hit-rates.

We also run the same experiment (changing cache size from 512B to 16KB) this time with PRAND replacement policy to see its effect. The improvement caused by PRAND over PLRU is averaged over all benchmarks and plotted in Figure 2.13 (left: hit-rate

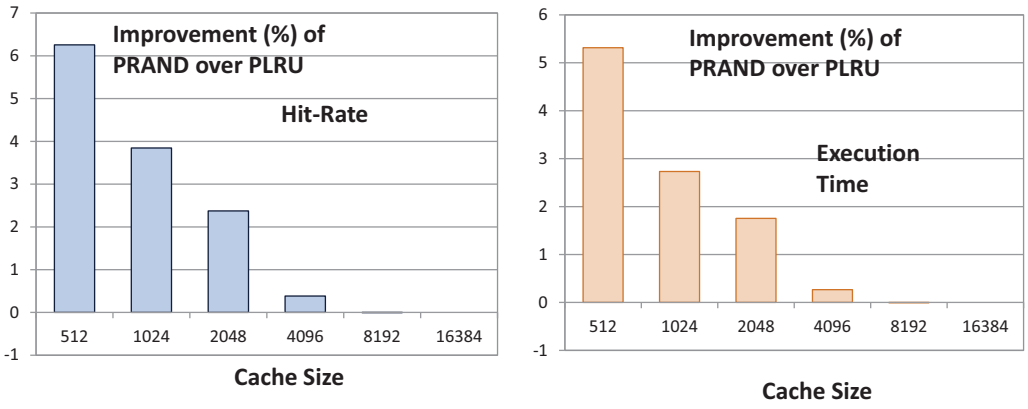

Figure 2.13: Average improvement of PRAND over PLRU in (left: hit-rate, right: execution-time) when cache size is changed from 512B to 16KB.

and right: execution time). It is interesting to see that when there is no prefetching, PRAND works slightly better than PLRU (up to 6% better in hit-rate and 5% better in execution time). But in the next chapters we will show that prefetching changes this behavior.

2.4.3 Effect of Cache Associativity

In the next experiment, the associativity of ICache is changed from 1 to 8, while replacement is PRAND and the cache size is 1KB. Hit-rate and execution-time results are shown in Figure 2.14. Again it can be seen that for *strassen*, no improvement happens from increasing the associativity. This is also true for *lu-u*. These two application mostly have compulsory misses and prefetching is needed to improve their performance.

Another interesting observation is that, increasing associativity for some benchmarks (e.g. *whirlpool-u*, *sobelkernel-u*, *cnnconv-u*, and *multifunc*) hurts their performance and reduces their hit-rate. This issue only happens for random replacement policy. For the codes which have multiple medium sized loops when we increase the associativity, we are actually increasing the probability that a block randomly gets evicted. For this reason, with higher associativity more useful blocks get evicted and performance drops. In this case having a direct-mapped cache works better than associative caches (with random replacement). This issue, however, does not happen when LRU or PLRU policies are implemented.

In the next chapter we will describe the design of our prefetching mechanisms.

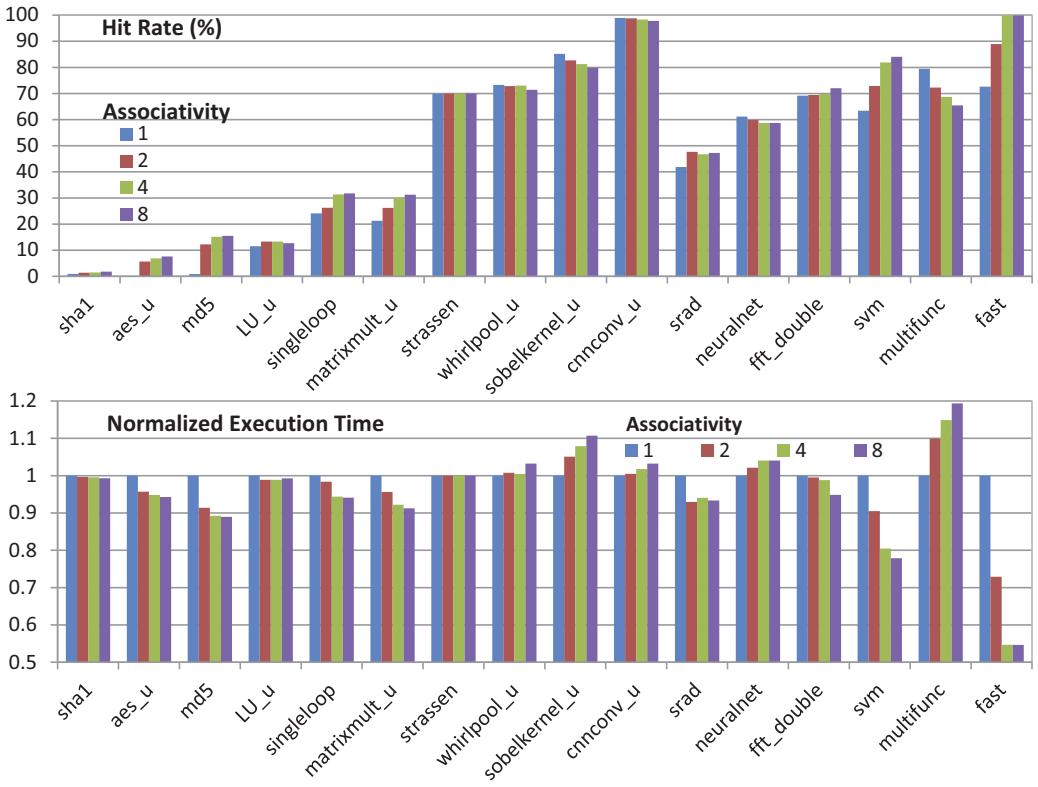

Figure 2.14: Effect of cache associativity on the performance of all benchmarks.

Chapter 3

Prefetcher Design

In the previous chapter we saw the baseline architecture of the multi-ported ICache connected to 4 processors. In this chapter we modify the baseline architecture and add low-cost hardware mechanisms to support both software and hardware prefetching.

3.1 Design of a Software Prefetcher (SWP)

As shown in Figure 3.1, we add a new finite state-machine (FSM) and connect it to the controller of the ICache (*icache ctrl*). We use two specific registers in the icache-controller to receive prefetch commands from software. The user application can give a software prefetch request by writing to these registers. The FSM is then connected to a new "private-controller" which is responsible for issuing the prefetch request. The difference between this private-controller and the others is that this one only has a read port to the TAG array and not to the DATA array. The reason is that, prefetcher does not need to access the data and only needs to issue a refill request for a block which is not currently in the cache. This ensures minor additional hardware complexity.

A prefetch request from software comes with two additional parameters: sw_pf_address is the address to be prefetched, and sw_pf_size indicates the number of bytes to prefetch from this address. Figure 3.2 shows the macros we have defined in C to help users with software prefetching. Also the registers inside the icache-controller are shown in this figure which receive the prefetch command from software.

When a software prefetch request arrives at the FSM (shown in Figure 3.1), depending on the size of that request, the FSM issues it to the private-controller word-by-word (each word is 16Bytes). The block diagram of this FSM is shown in Figure 3.3. For example if a prefetch request of 64B arrives, the FSM goes from the IDLE state to

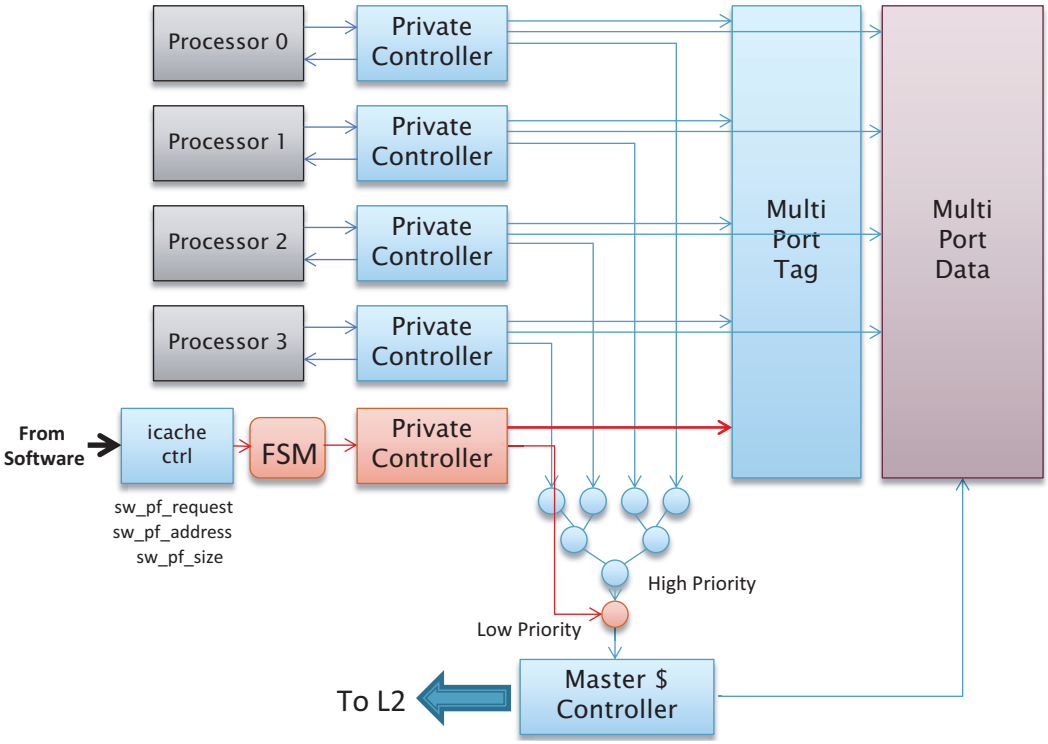

Figure 3.1: Hardware modifications for implementation of the software prefetcher.

Inside the C code (Software):

```
#define PREFETCH_ADDRESS(X)  {*(volatile int*) (ICACHE_CTRL_BASE_ADDR+0x08) = (int) ((X)) ;}
#define SET_PREFETCH_SIZE(X) {*(volatile int*) (ICACHE_CTRL_BASE_ADDR+0x18) = (int) ((X)) ;}

#ifdef ENABLE_SW_PF
SET_PREFETCH_SIZE(SW_PF_SIZE);
...
PREFETCH_ADDRESS(0x1C000000);
...
#endif
```

Inside Hardware

```
8'h08: // Maryam: Prefetche Command + Address
begin
  $display("Maryam: PREFETCH ADDRESS: 0x%x", wdata);
  globalcommand_prefetch_request = '1;
  globalcommand_prefetch_address = wdata;
  deliver_response = 1'b1;
  NS = IDLE;
end
8'h18: // Maryam: Prefetch Size
begin
  $display("Maryam: PREFETCH Size: 0x%x", wdata);
  prefetch_size_n = wdata;
  marker_event_n = '1;
  deliver_response = 1'b1;
  NS = IDLE;
end
```

new_icache_ctrl_unit

Figure 3.2: The software prefetching macros inside C, and the hardware registers added to icache-controller to enable software prefetch.

REQ state and issues a request of 16B starting from sw_pf_address. Then goes to the CHECK state, reduces the counter by 16B and increments the prefetch address by 16B, as well. If the counter reaches zero, the burst request has finished and the FSM goes

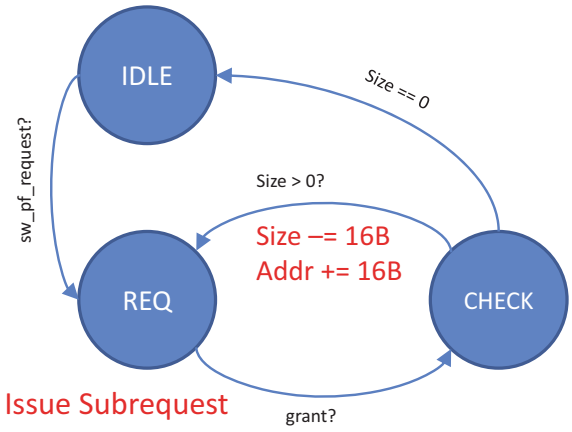

Figure 3.3: The finite-state-machine issuing for the prefetch requests.

back to IDLE. This way, the FSM is able to issue burst requests of any size. Also, if another prefetch request arrives while serving a previous request, the FSM drops the previous request and starts the new one. This mechanism is therefore "preemptive". The idea behind this is that if a new request comes, probably the previous burst request is stale and should not be continued. It should be noted that this "preemption" only happens for burst requests larger than 16B, while 16B requests are issued without being preempted.

Another point to mention is about the necessity of another private-controller, instead of just connecting the FSM in Figure 3.1 directly to the L2 bus. The reason is that when this private-controller receives a prefetch request, first it checks if it is already a HIT or not. If yes, it just ignores it. This way it ensures that the L2 bus is not polluted with useless requests consuming energy and bandwidth. Also if all target ways are full, the private controller needs to select one of them (based on the replacement policy) and ask the master controller to replace it when the refill response comes back from the L2 memory. On the other hand, since prefetcher does not need to access data, its private-controller is simpler than the other ones connected to the processors. In chapter chapter 5 we will show the effect of the addition of the new prefetcher on silicon area and power consumption.

One interesting benefit of the PLRU replacement appears when we use it with the prefetcher: we can treat prefetched blocks and normal blocks differently for replacement. Whenever the prefetcher accesses a block, we do not update the LRU counter, so the block which is touched by the prefetcher is not set to the most-recently-used (MRU) one. This way we make sure that prefetched blocks are more prone to being replaced than the normal blocks. This helps reduce the cache pollution caused by the

prefetcher. In section 4.1 we will show the performance results related to this software prefetcher.

3.2 Design of a Next-line Prefetcher (NLP)

In this section we describe the design of our hardware-prefetcher which is able to automatically send prefetch requests to improve hit-rate and performance. As we discussed before, next-line prefetchers (NLP) have the lowest hardware complexity and work well for codes with not so many branches. For this reason NLPs can be beneficial for our Group 1 benchmarks. An NLP waits until a cache-miss happens, and then gives a prefetch request to the 'next line' after the missed cache line. This method can also be extended to N-next-line prefetching.

We can easily implement NLP inside our platform by making a small change to the architecture of the ICache (previously showed in Figure 3.1) and to the FSM of the software-prefetcher (previously showed in Figure 3.3). As shown in Figure 3.4 we monitor the L2 bus (connecting the cluster to the L2 memory) to see if a refill request is happening. A request on this bus means that a cache-miss has happened and a refill request has been sent to the L2 memory. We send the miss_address to the FSM and make a small change to it as shown in Figure 3.5: In the IDLE state, in addition to checking for a software-prefetch command, we also check for a miss on the L2 buss. Upon a miss we go to the REQ state and start a burst prefetch starting from (miss_address + 16) with a parametric prefetch size. Note, that the cache line size is 16Bytes and also the RISC-V cores already have an L0 prefetch buffer inside them which allows them to fetch 16 Bytes at a time. For this reason the granularity of all operations is 16Bytes throughout this work.

Another important concern is if a new miss happens while serving a new prefetch request. Again, we treat this with "preemption", similar to the decision we made for software prefetching. Therefore in the CHECK state if another miss happens we ignore the rest of the current prefetch command and start from the new miss address. This ensures that our prefetching is timely and useful and we are not performing any useless prefetching. The obtained results related ton this next-line prefetcher are presented in section 4.2.

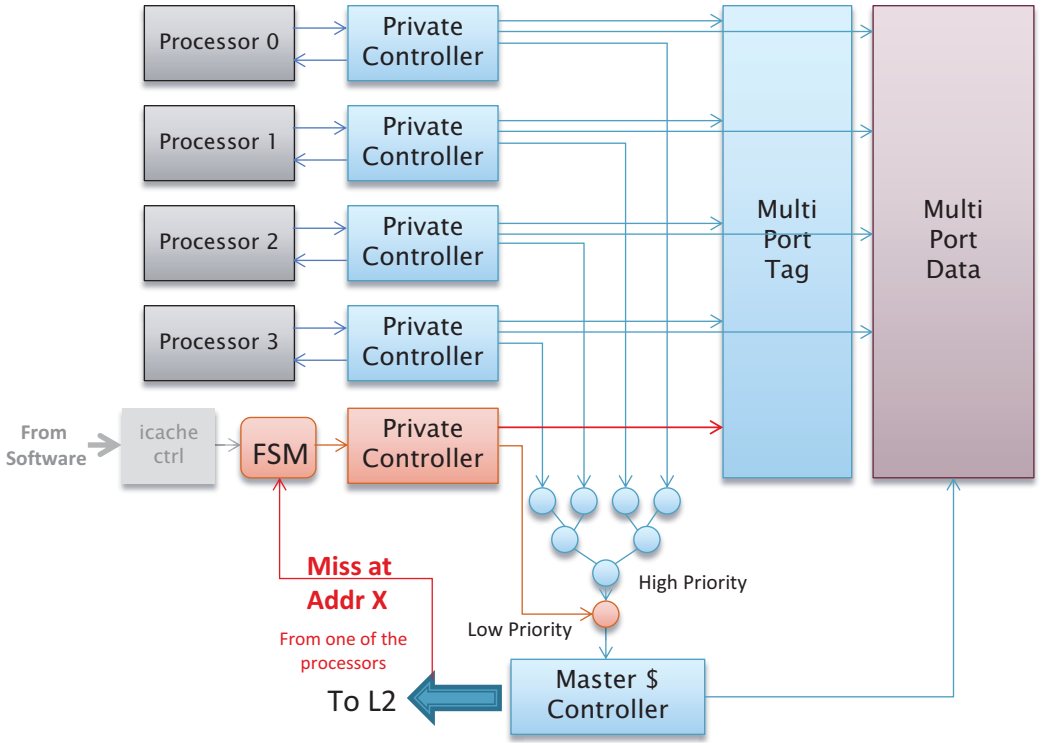

Figure 3.4: Hardware modifications for implementation of the next-line-prefetcher.

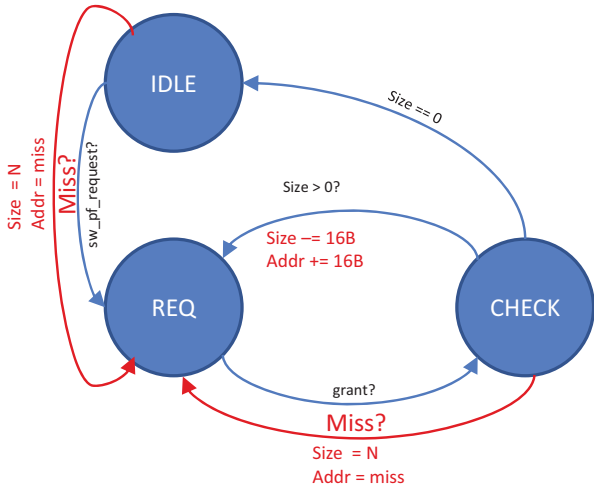

Figure 3.5: Modifications to the state-machine to build a next-line-prefetcher.

3.3 Extending the NLP to a Stream Prefetcher (STP)

One issue with NLP is that it operates on cache-misses. So it is only activated when a cache miss has already happened. For this reason, it cannot reach a hit-rate close

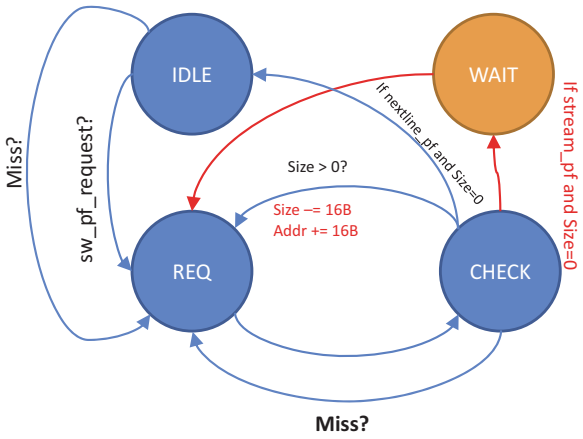

Figure 3.6: Modifications to the state-machine to support stream-prefetching.

to 100%. In this section we design a more intelligent prefetcher on top of the NLP: Instead of waiting for a miss to start a new prefetch request, we start upon completion of a previous prefetch request. However, since we do not want to issue too many prefetch requests and pollute the cache and L2 bus, we add a programmable number of wait-states to the FSM before starting a new prefetch request. This is shown in Figure 3.6. As can be seen, in the CHECK state if the previous prefetch command is finished we go to the WAIT state and after waiting for a specific number of cycles, we restart prefetching from the last prefetched address plus 16.

Of course, "preemption" is implemented in all states of the FSM to make sure we are not issuing useless prefetch commands. For example if inside the WAIT state a new miss happens, we start prefetching from that miss-address. Similar thing is true if in any of the states a software-prefetch request arrives. With this FSM we are able to perform software+hardware prefetching at the same time to take advantage of both of them. The obtained results for stream-prefetching are shown in section 4.3. Also, we will shown in chapter 4 that a combination of software+hardware prefetching can powerfully improve hit-rate and performance for most of the benchmarks.

Chapter 4

Effect of Prefetching on Performance

In this chapter we will show the performance impact of our designed prefetchers. For all experiments we use the baseline configuration, unless otherwise stated:

```
Cache Size:  1KB
Associativity:  2
Clock Period:  20ns
Replacement:  PLRU/PRAND
```

4.1 Software Prefetching Results

First we focus on the software prefetcher and try to show how this simple prefetcher can help hide the access latency of the processors to ICache. Figure 4.1 shows the source code of matrix-multiplication augmented with a single software prefetch command. This command has been carefully adjusted to be able to prefetch the complete loop. Figure 4.2-top shows the address pattern of this code when prefetching is disabled. We can see that in the first iteration many compulsory misses happen, but after that all access hit in the cache. But as Figure 4.2-bottom shows, when software prefetching is enabled almost all misses in the first iteration turn into hit, even though the prefetch distance is zero and prefetch command is placed right before the loop.

To understand better why this is happening we can take a look at Figure 4.3. Since the RISC-V processor is single-issue, it issues one fetch request to the ICache, waits to receive the instruction, then issues another request after that. In the first iteration of the loop, all ICache accesses result in miss so the core has to wait for

```
SET_PREFETCH_SIZE(SW_PF_SIZE);      Bytes to Prefetch
PREFETCH_ADDRESS(0x1c000290);       Start Address

for(i=0;i<SIZE;i++){
   for(j=0;j<SIZE;j++){
      matC[i][j]=0;
      for(k=0;k<SIZE;k++){
         matC[i][j]+=matA[i][k]*matB[k][j];
      }
   }
}
```

Figure 4.1: Source code of matrix-multiplication augmented with a single software prefetch.

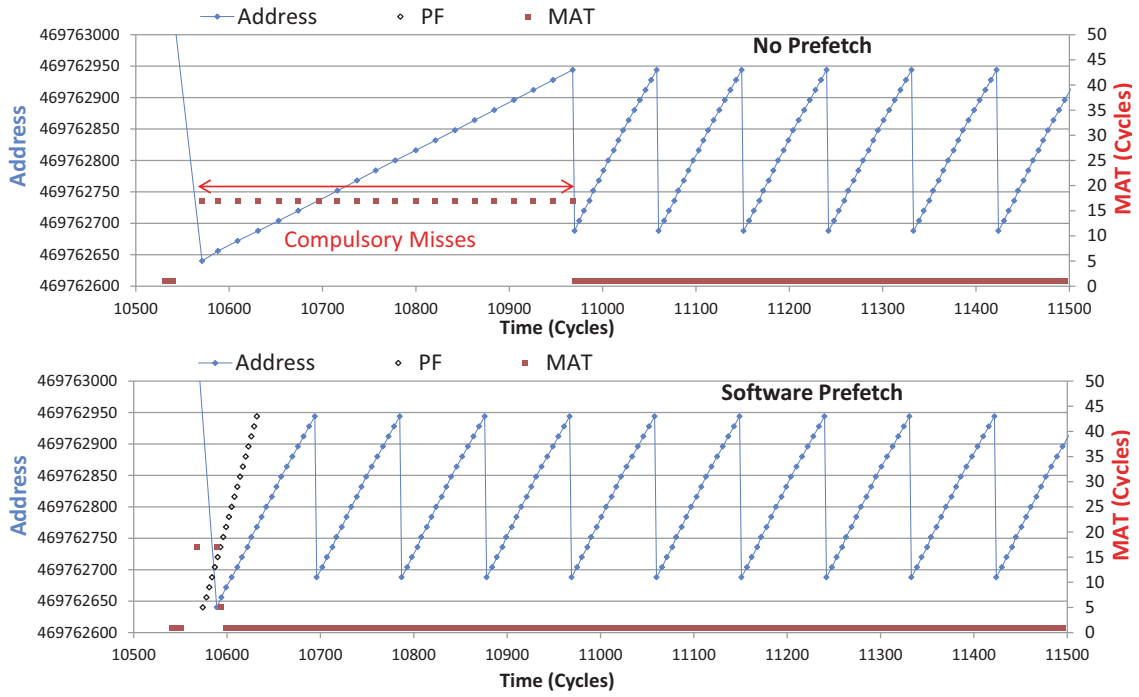

Figure 4.2: Execution of the code in Figure 4.1 once without prefetching (top) and once with one software prefetch (bottom).

the miss penalty every time it gives a request (See Figure 4.3 top). But when we use software prefetcher, we can give multiple small prefetch requests (or a large burst prefetch request) and fill N cache blocks in a short period. So prefetching allows for having multiple outstanding refill requests and it allows hiding the access latency. This

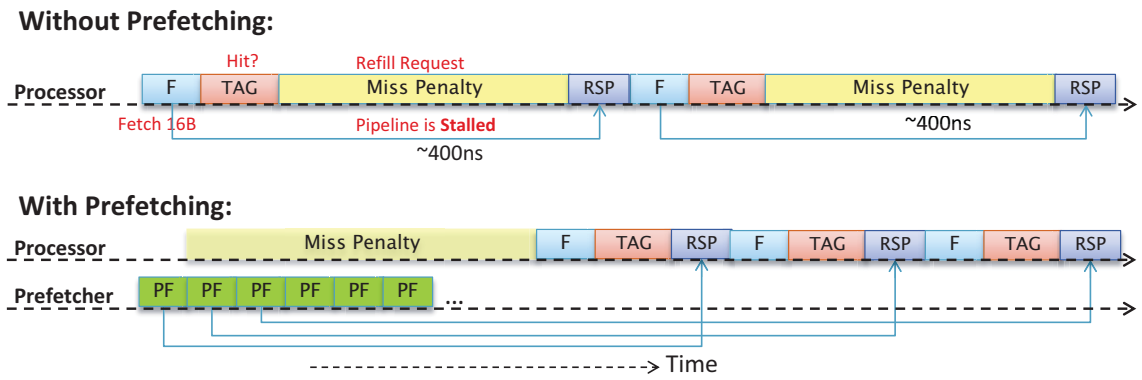

Figure 4.3: Demonstration of instruction request latency without prefetching (top), and with prefetching (bottom).

is the reason that in Figure 4.2 even with a prefetch distance of zero, still all misses except for the first ones turn into hit.

Next we choose 3 representative benchmarks (*strassen*, *lu-u*, and *fft-double*) from Group 1 and Group 2, and try to improve their hit-rate and performance with software prefetching. We have easily augmented *strassen* and *lu-u* benchmarks with software prefetch commands, but for *fft-double* this is a bit more difficult. Because it uses floating-point computations and our RISC-V cores do not have hardware any FPU. For this reason, a software floating point library is automatically linked with the final code and floating point operations are replaced with calls to the functions in this library. This makes software prefetching inside the C code more difficult because the function calls are implicit. One solution to this problem is to perform software prefetching inside the assembly code by the help of the compiler. This solution is very flexible and it can easily prefetch complicated function calls, but we leave it as a future work, as it needs changes to the compiler. In this work, we choose an alternative solution to overcome this problem. We use a software floating-point library [39] instead of the standard floating point library. We have replaced all floating point computations in the *fft-double* code with calls to the functions in this library (e.g. float64_add(), float64_sub()). This way we are able to put software prefetch instructions right before these functions and also inside them. Also, we perform a combination of SWP and NLP to achieve a better hit-rate.

Figure 4.4 shows the hit-rate and execution time for 4 different cases. *Baseline* represents the baseline configuration without any prefetching. *NLP* is the next-line-prefetcher with prefetch-size of 128Bytes (We will show the detailed results of NLP in section 4.2). *NLP+SW* is the combination of NLP and SWP, where they are both

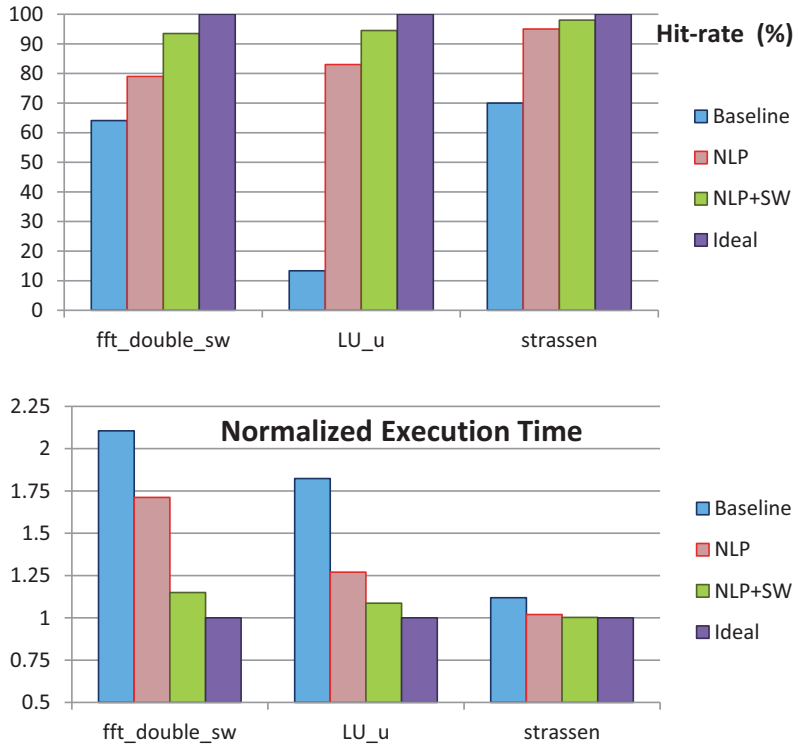

Figure 4.4: Effect of software prefetching on hit-rate and execution-time.

working. Finally, in *Ideal* the codes have been executed twice on a large cache (16KB) and stats are reported on second execution. This way hit-rate of the *Ideal* case is always 100%.

We can see that a combination of NLP and SWP can reach the hit-rate of all 3 benchmarks very close to 100% (96% on the average). Also, the average execution time is only 7% higher than the ideal case. The address plots for the *NLP+SW* case are shown in Figure 4.5. We would like to remind that, the rest of the remaining misses can also be removed by spending more effort on the software prefetcher and accurately tuning it, or by leaving the software prefetching job to the compiler. This shows the effectiveness of our proposed solution.

4.2 Next-line Prefetching Results

Now we show the results of NLP for all benchmarks and study the impact of different parameters on it. In the first experiment, we change the prefetch size of the NLP from 0Bytes (Disabled) to 288Bytes and measure performance. Figure 4.6-top shows the

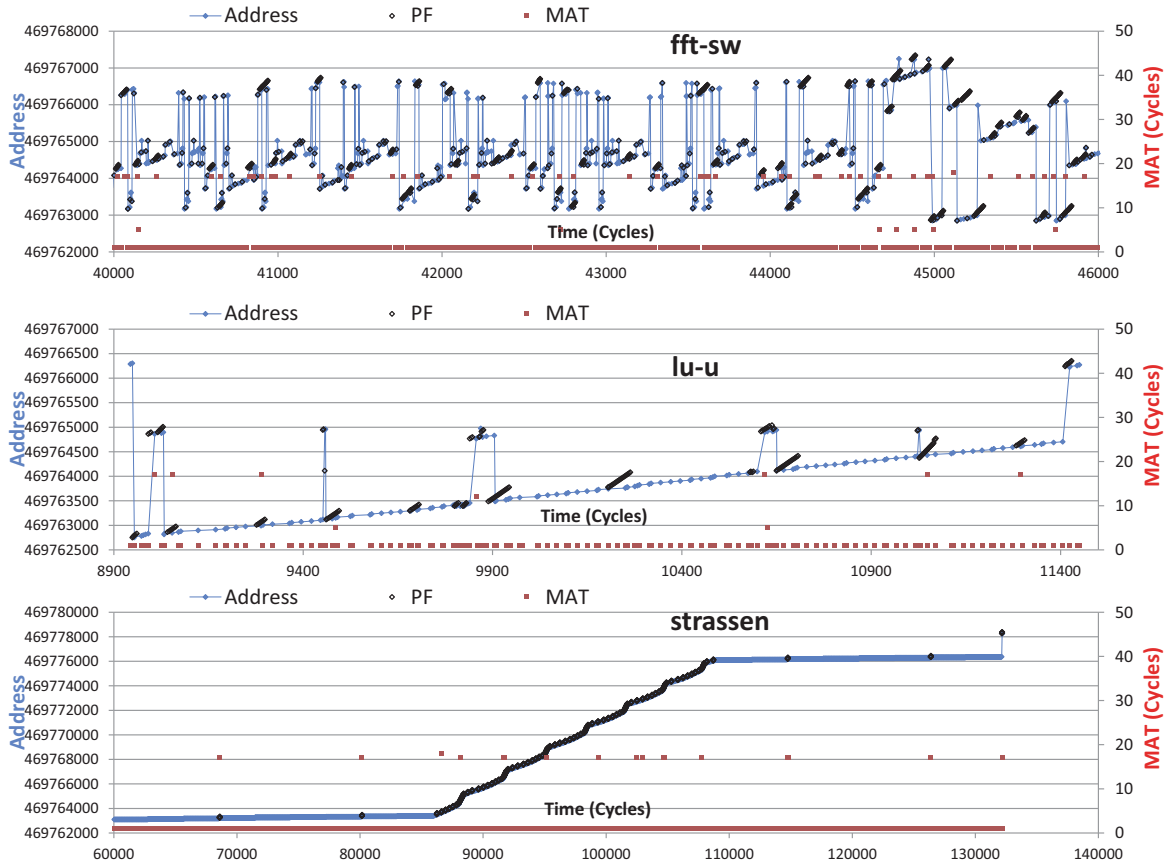

Figure 4.5: The address plots for 3 benchmarks with SWP and NLP enabled.

average hit-rate when PLRU is used and Figure 4.6-bottom shows the same experiment for PRAND replacement. First, it can be seen that the hit-rate of most benchmarks in Group 1 significantly improves (from an average of 30% to 93% on the average for both PLRU and PRAND). This is a very significant improvement and shows that NLP works very well for the Group 1 codes, because they mostly have large loops with few branch and function call instructions. It is also interesting to see that for Group 2 also some improvement is obtained (from 70% on average to 84% with PLRU and to 80% with PRAND). The codes in this group have many function calls or branches. For this reason NLP is not very effective for them, and a combination of NLP and SWP is needed to further improve their hit rate. This is what we studied in the previous section.

Another interesting point to observe is that for the *fast* benchmark NLP is only hurting performance instead of improving it. And this performance drop is worse with PRAND replacement than PLRU. As we described before *fast* has too many branch

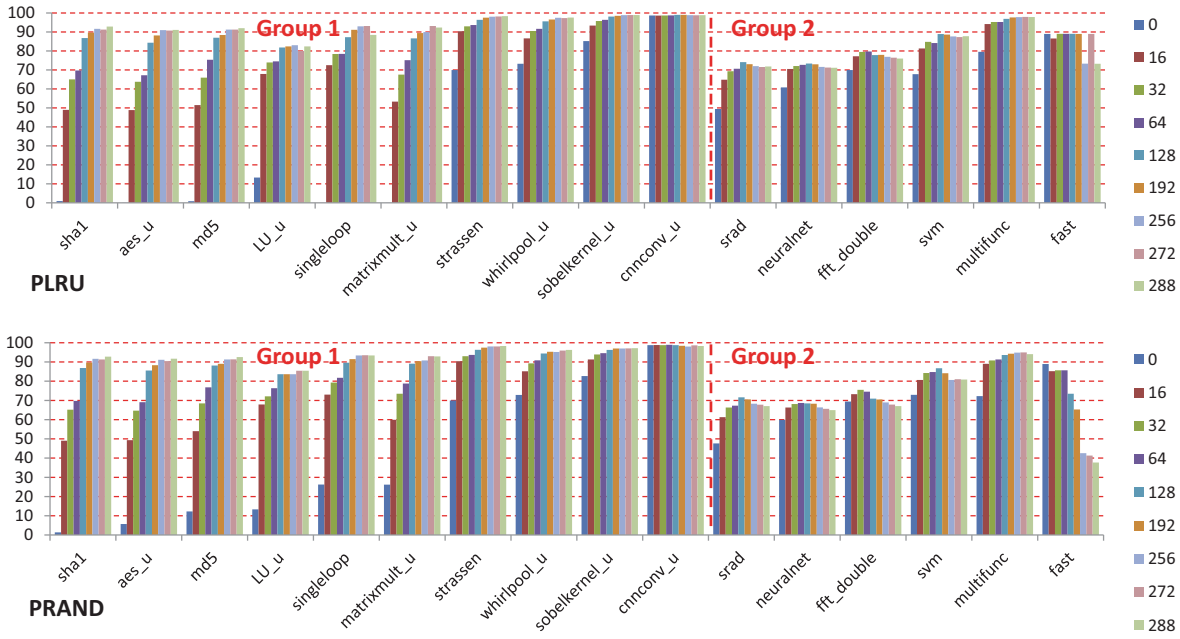

Figure 4.6: Effect of prefetch-size of NLP on the ICache hit-rate. Top: PLRU replacement, Bottom: PRAND replacement.

instructions and it is impossible for a simple NLP to improve its performance. Also, since PRAND randomly evicts blocks, hit-rate of *fast* drops more severely with PRAND compared to PLRU.

Figure 4.7 shows normalized execution-time for the previous experiment again for two cases: PLRU (top) and PRAND (bottom). As expected, for Group 1 benchmarks, execution-time improves significantly (by 1.8X for PLRU and 1.75X for PRAND replacement), while for Group 2 this improvement is smaller (1.25X for PLRU and 1.20X for PRAND). Again, it can be seen that *fast* is not getting any benefit from NLP, so it is better to disable it. Also, *cnnconv-u* does not benefit because it already has a hit-rate close to 100%.

In order to understand if our prefetcher is saturating the L2 bus or not we also plot the bandwidth of this bus (MB/sec) in Figure 4.8. The L2 bus has a data-width of 8Bytes and works with the same clock period as the cluster. So it can deliver a bandwidth up to 400MB/sec. We can see that, even with very large prefetch requests (288Bytes) still the L2 bus is not saturated and L2 bandwidth is always below 50MB/sec. So we can be sure that the L2 memory is not a bottleneck and performance is limited by the processor's performance and the hit-rate of the ICache. Also it is interesting to see for *fast* that, increase in the prefetch size only pollutes the L2 bus

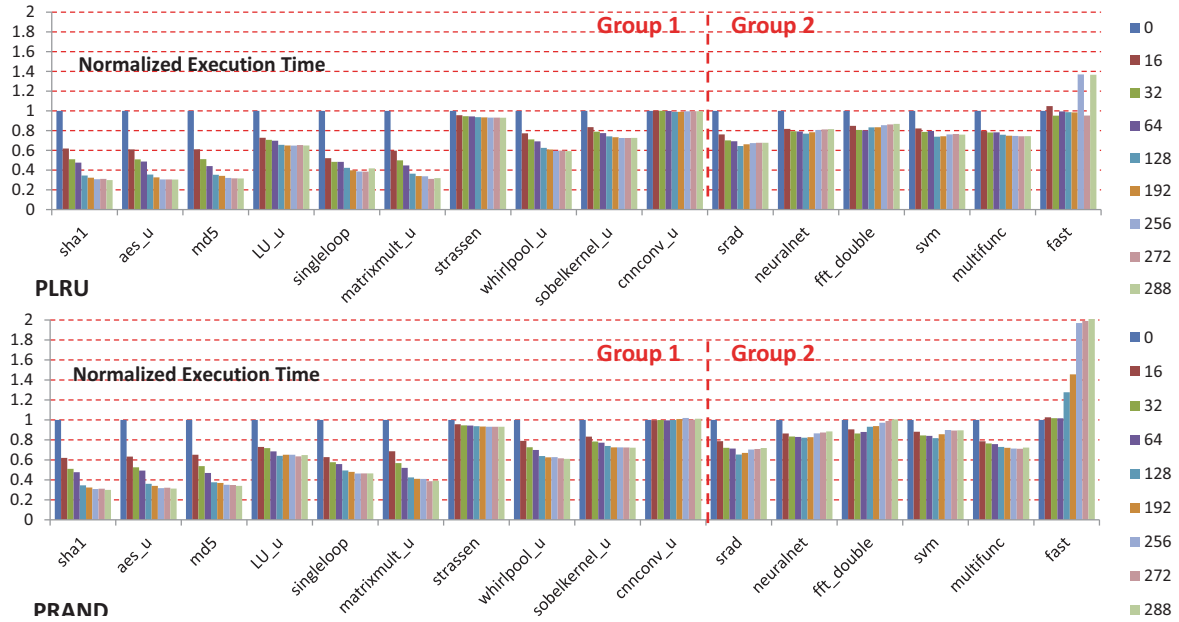

Figure 4.7: Effect of prefetch-size of NLP on the total execution time. Top: PLRU replacement, Bottom: PRAND replacement.

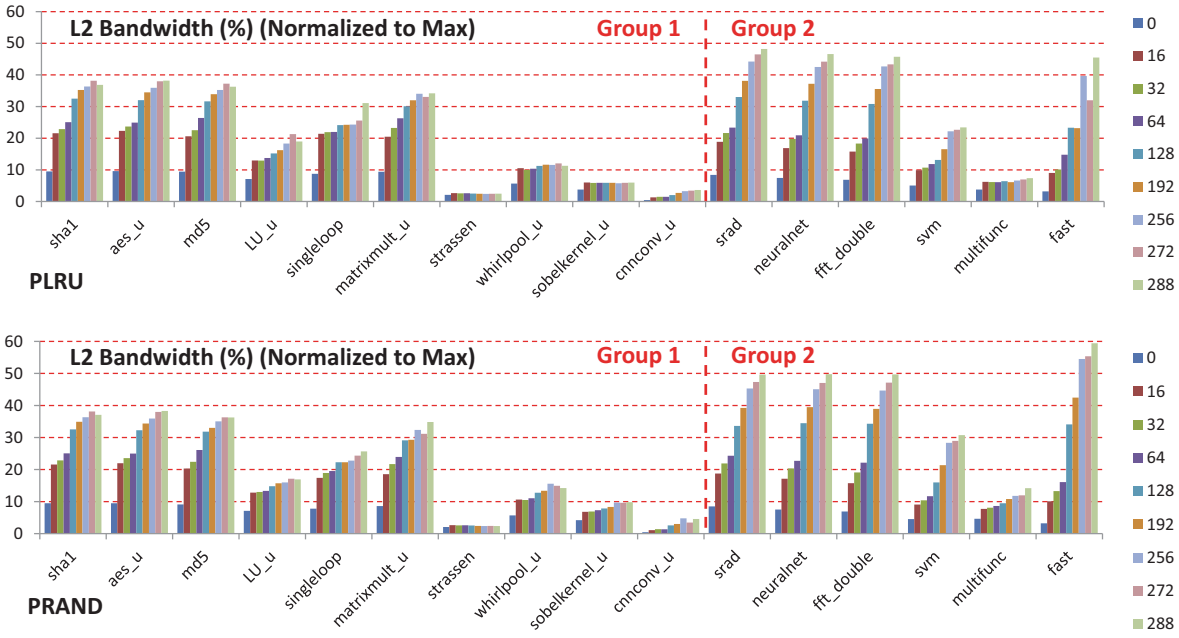

Figure 4.8: Effect of prefetch-size of NLP on the L2 bandwidth (MB/sec). Top: PLRU replacement, Bottom: PRAND replacement.

and does not give any benefit in performance or hit-rate (as we saw before).

Finally, Figure 4.9-left shows the best hit-rate which has been achieved with NLP

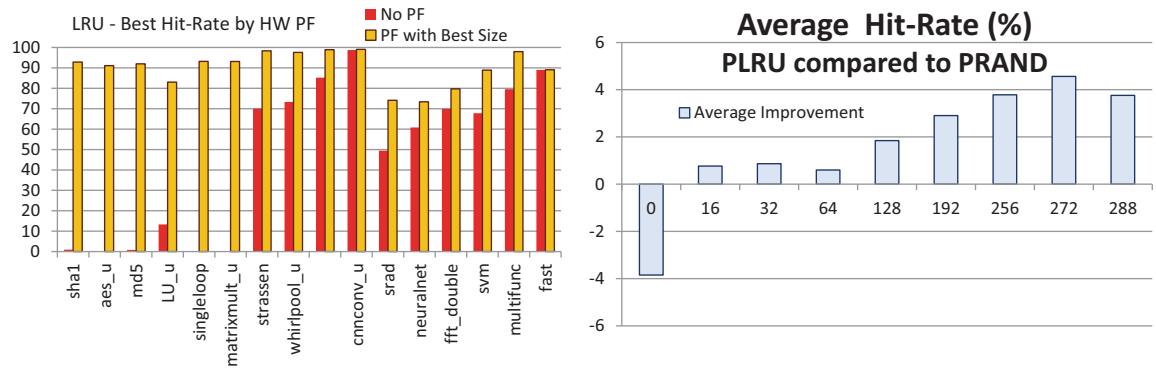

Figure 4.9: The best hit-rate achieved by NLP (left), and average hit-rate improvement of prefetching compared between PLRU and PRAND (right).

for different benchmarks. This plot is only shown for PLRU replacement. Again, we can see that for Group 1, NLP works perfectly and improves hitrate significantly. For the first 6 benchmarks, the hitrate is improved from an average of 2% to 91%. Also, to understand which replacement policy works better with prefetching we have averaged the results of Figure 4.6 and plotted the difference between PLRU and PRAND results in Figure 4.9-right. Interestingly, we can see that when prefetching is disabled, PRAND works slightly better than PLRU (about 4%). It was previously explained that for codes with very large loops PLRU works poorly, because it keeps evicting the LRU blocks while they are needed in the next iterations of the loop. This happens less in PRAND because some of the blocks remain in the cache and are later used. However, as we enable NLP and increase prefetch-size, we see that PLRU starts to work better than PRAND (up to 4%). This is because PLRU manages the prefetched blocks better and creates less cache-pollution. Also, the fact that we treat prefetched blocks differently from normal blocks is helpful in reducing cache pollution.

To understand better how NLP is working Figure 4.10 shows the address plot of 3 benchmarks with PLRU replacement and NLP with prefetch size of 256Bytes. It can be clearly seen in *sha1* and *md5* that all burst prefetches start only when a cache-miss happens. For this reason there are still many misses remaining in-between. We will show in the next section that a more intelligent stream-prefetcher can remove these misses, as well. Also, for *lu-u* we see that it has both a large loop and some function calls. The NLP is effective for the large loop, but not for the function calls. This is why a combination of NLP+SWP can be effective and remove both types of misses (as we saw in Figure 4.5 in the section 4.1).

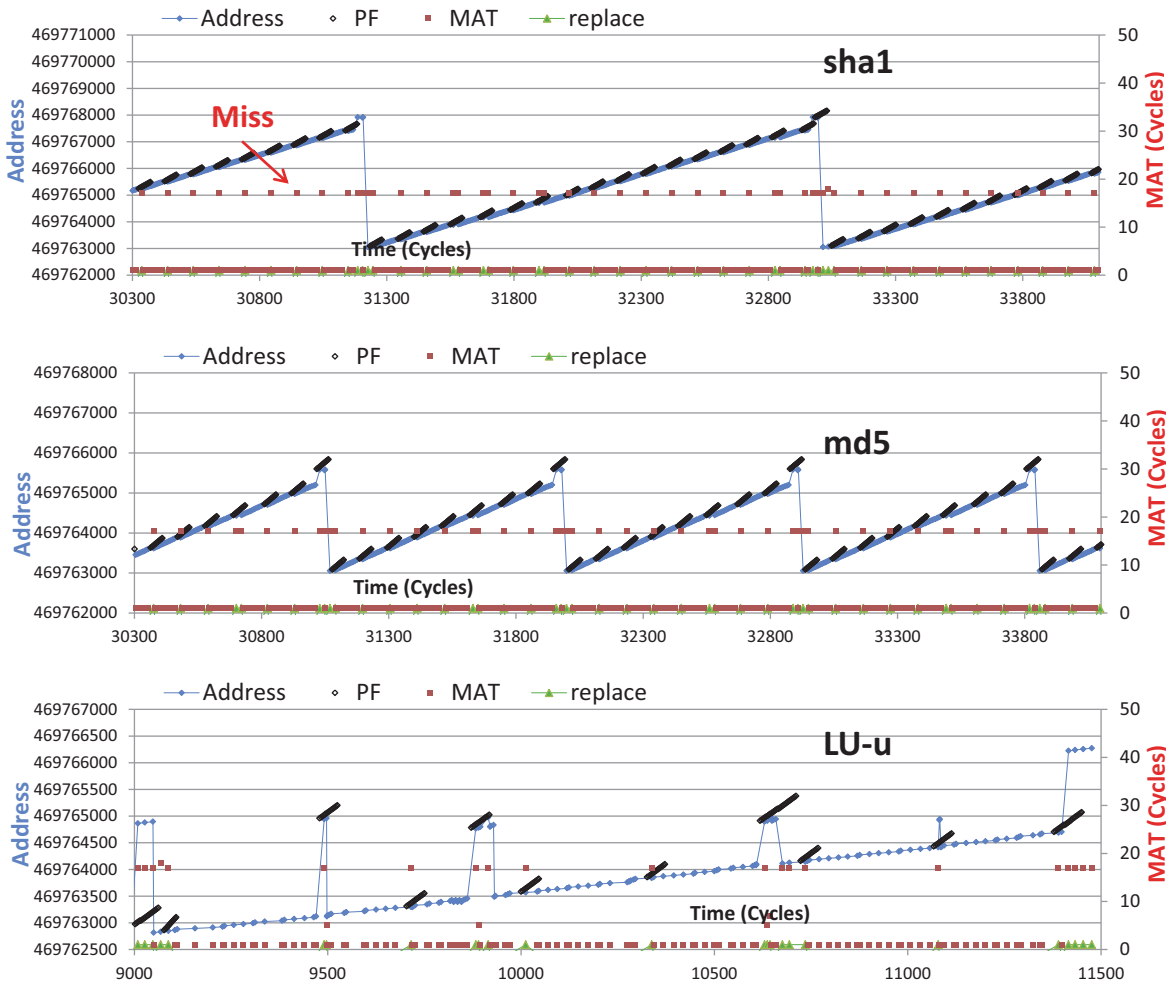

Figure 4.10: The address plot of three benchmarks with PLRU replacement and NLP with size 256Bytes.

4.3 Stream Prefetching Results

We saw in the previous section that NLP is not able to remove the misses between prefetch requests because it issues a new request only when a miss has already happened (See *sha1* and *md5* in Figure 4.10). In this section, we use stream-prefetching with a burst-size of 256Bytes and change the wait-cycles in its state-machine (see Figure 3.6) to see its effect on performance. First we show the address plot for 3 cases of stream-prefetcher in Figure 4.11. In *Wait=0* the prefetcher does not wait and starts the next prefetch immediately after one has finished. *Wait=30* waits for 30 cycles between different requests, and *Wait=60* waits for 60 cycles. It is interesting to see that *Wait=0* works poorly, because the rate of prefetching is much faster than the rate of execution and consumption of the blocks by the processor. For this reason the cache gets polluted

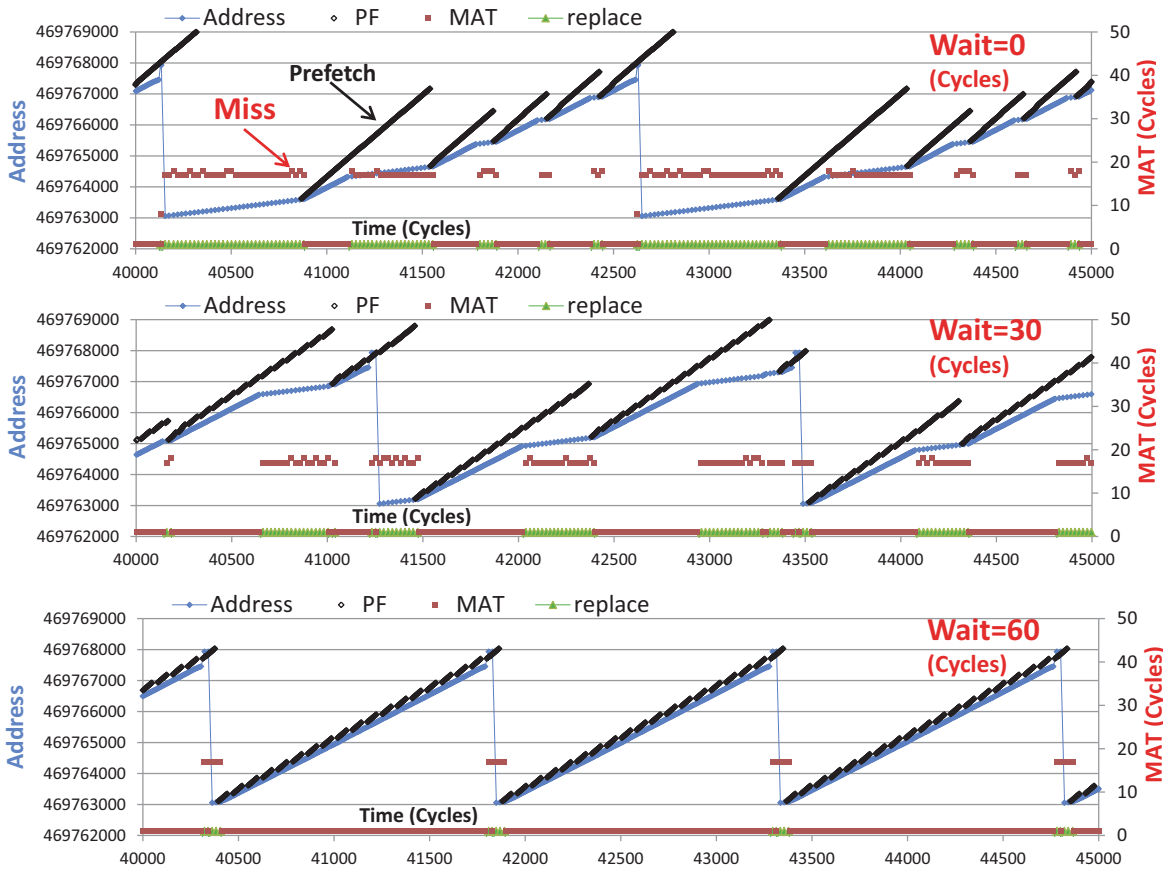

Figure 4.11: Address plots for three cases of stream-prefetcher with burst-size of 256Bytes and wait-cycles of 0, 30, 60 (Cycles).

and the hit-rate even decreases, compared to the case with NLP only. Please note that, the prefetcher can fill up the cache at the speed of 16Bytes per cycle, while the RISC-V core can consume these blocks at a rate less than 4Bytes per cycle, depending on how long the instructions take to complete. Similarly, in *Wait=30* the rate of prefetching is still faster that than the rate of execution. *Wait=60*, however, gives a reasonable hit-rate and is even able to remove all the misses between subsequent requests.

To understand the effect of wait-cycles better we can plot the normalized execution-time of the benchmarks in Figure 4.12, when wait-cycles is changed from 0 to 150. The first point in this plot (identified by "NO") indicates NLP only without any stream prefetching. As we saw before, zero wait-cycles works much worse than the baseline case with NLP only. For all studied benchmarks wait-cycles of 50 to 60 is found to be optimal, and the lowest execution time is achieved. This parameter can be preprogrammed in the FSM and these applications can benefit from it. Finally, in Figure 4.13-left the best hit-rate achieved by STP is plotted in comparison with the

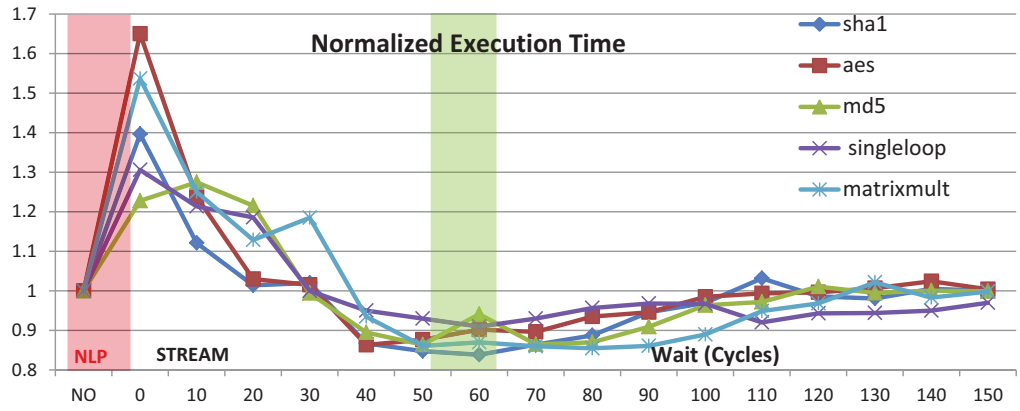

Figure 4.12: The effect of wait-cycles in STP on the execution time of different benchmarks.

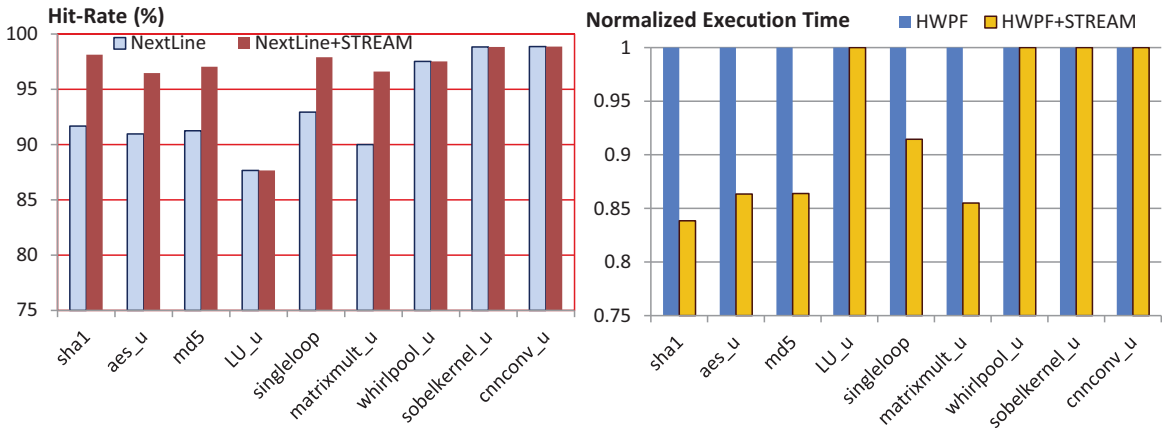

Figure 4.13: The best hit-rate achieved by STP in comparison with NLP (left), and the best execution time in the same experiment.

hit-rate of NLP, and in Figure 4.13-right the best execution time is plotted in the same experiment. Thanks to this stream-prefetcher, we can achieve a hit-rate of over 95% for almost all benchmarks. The only benchmark not gaining any benefit from STP is *lu-u*, but we already showed in section 4.1 that a combination of SWP and NLP can improve its hit-rate to about 95%.

In this work we proposed 3 easy to implement prefetching schemes: SWP, NLP, and STP. We showed that NLP and STP work very well for benchmarks with large loops (Group 1). Also for the benchmarks of Group 2 we showed that a combination of SWP and NLP can boost their performance. Next we will study the impact of the proposed methods on silicon area, power consumption, and energy using the state of the art technologies.

Chapter 5

Analysis of Power Consumption, Energy, and Silicon Area

In this section we study the impact of our proposed mechanisms on silicon area and power consumption of the processing cluster. For synthesis we use Synopsys Design Compiler (2014) in topographical mode. We export the post synthesis net-list and parasitics and feed them to Synopsys Primetime (2014) for power estimation, along with the switching activity from ModelSim. For synthesis we focus on one cluster and use the baseline configuration that we used throughout this work:

```
Number of RISCV processors:  4
I-Cache Size:  1KB
Associativity:  2
Replacement:  PLRU/PRAND
HW and SW prefetchers:  Enabled/Disabled
TCDM Size:  8 x 2K: 16KB
```

We used the following synthesis setup:

```
Technology:  FDSOI 28nm (STMicroelectronics) - RVT
Temperature:  125(C)
Voltage:  0.9V
Corner:  Slow-Slow
```

With this setup, we were able to achieve a clock period of 2.0ns, easily. For power extraction, we used the following setup:

```
Temperature:  25(C)
Voltage:  0.9V
Corner:  Typical-Typical
```

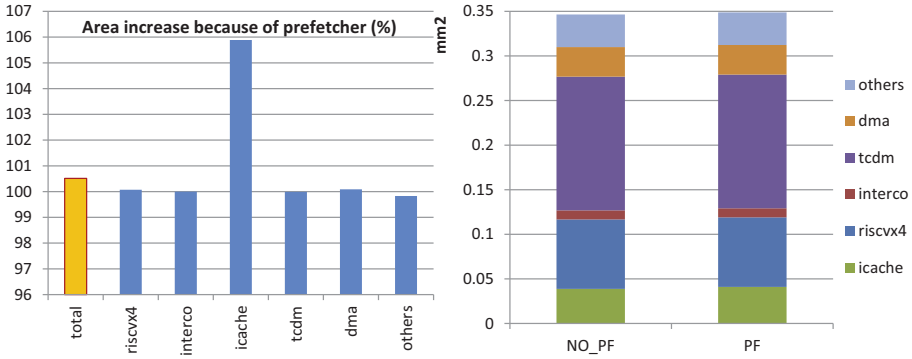

Figure 5.1: Percentage of area increase due to the prefetcher (left), and average area breakdown in the cluster with/without prefetching (right).

```
Benchmarks:  LU-u, md5, sha1, strassen
Clock Frequency:  500 MHz
```

First, we will study the effect of our proposed prefetchers on area. We run the synthesis twice once with both Hardware+Software prefetchers enabled, and once without prefetching. Figure 5.1-left shows the percentage of area increase when the prefetcher has been added to the cluster. It can be seen that the only component which is affected is the *icache* with an area increase of 6%. Interestingly, the total area increase in the cluster is only about 0.5% which is insignificant.

To understand this better, Figure 5.1-right shows the average area break-down in different components. *NO_PF* is without prefetching and *PF* is with prefetching. Again this plot shows that the percentage of change to the total area has been insignificant. These plots show that our prefetcher is implementable and realistic.

For power extraction, we run 4 representative benchmarks (*lu-u*, *md5*, *sha1*, and *strassen*) once with prefetching (prefetch size: 128B) and once without prefetching. Also, to estimate system-level power, we use a power model for the power consumption in the SoC (specifically in the L2 memory). Figure 5.2-left shows the total consumed power in the system (Cluster+SoC) without and with prefetching, and Figure 5.2-right shows power break-down in the whole system. On the average prefetching increases power consumption by 11%. Also, when prefetching is enabled, the power of both *icache* and *riscv* increase. This is because of higher hit-rate and higher utilization of the cores. As these plots show, the prefetching does not increase system's power significantly.

For the next experiment we study the effect of PLRU replacement policy on area

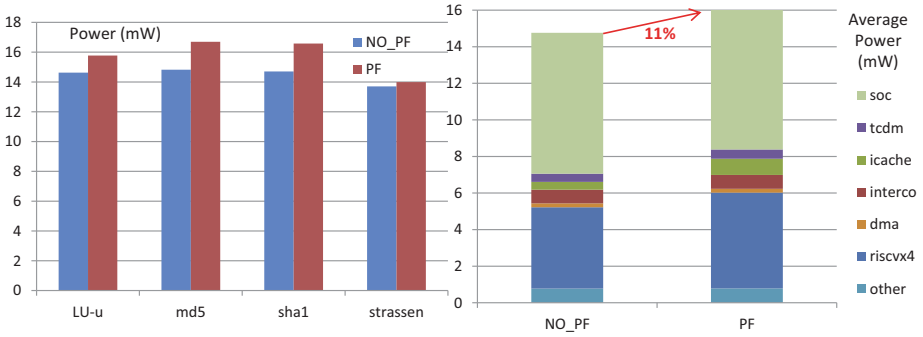

Figure 5.2: Total system power with/without prefetching for different benchmarks (left), average power break-down compared between the two cases (right).

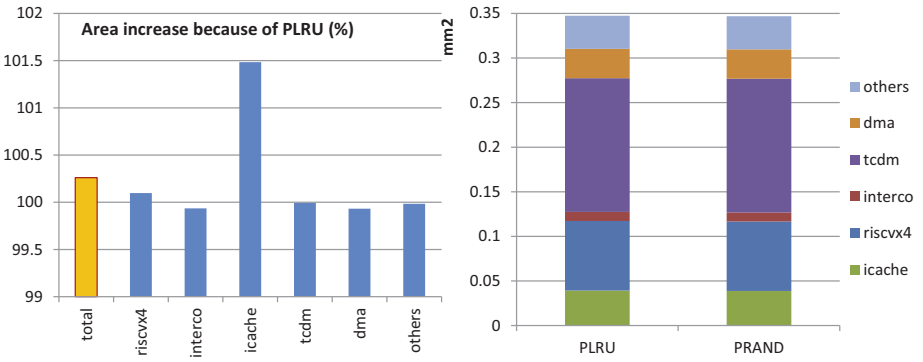

Figure 5.3: Percentage of area increaes in PLRU compared to PRAND (left), and average area breakdown in the cluster with PLRU and PRAND (right).

and power consumption. In the same baseline configuration, replacement policy is changed from PRAND to PLRU. Figure 5.3-left shows the percentage of area increase in different components. Again, it can be seen that only the area of the *icache* is affected with a minor increase of 1.5%. In total the amount of area increase in the cluster has been less than 0.3%. This is also shown in Figure 5.3-right where no significant difference between the two cases is observed.

Next, the effect of replacement policy on power consumption is studied. Figure 5.4-left shows the total system power in the two cases for different benchmarks, and Figure 5.4-right shows the average power break-down in different components. The power consumption in the ICache increases by about 12% in PLRU compared to PRAND, but the total system level power only increases by a small amount of 0.5%.

In chapter 4 we showed that addition of the prefetchers can lead to a significant

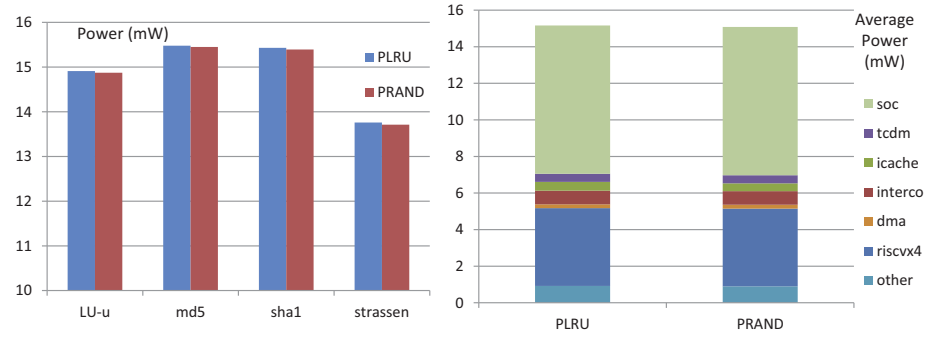

Figure 5.4: Total system power with PRAND/PLRU for different benchmarks (left), average power break-down compared between the two cases (right).

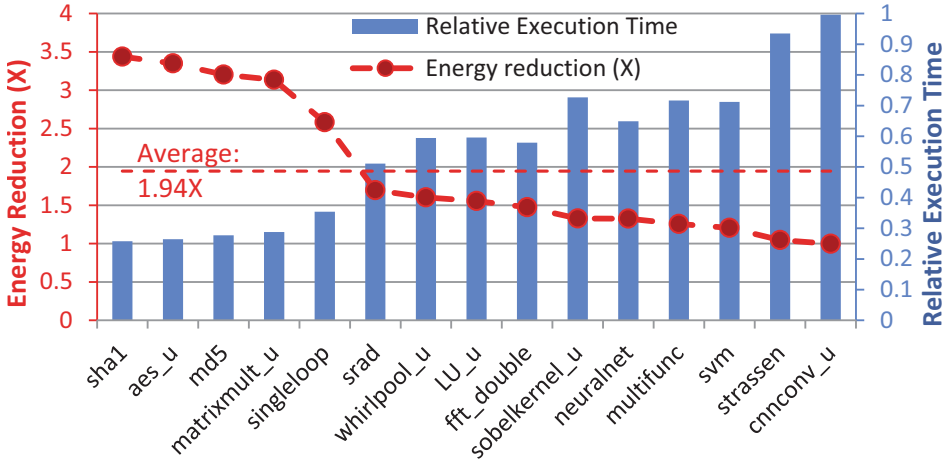

Figure 5.5: Total energy reduction thanks to the proposed prefetching mechanisms (left-axis), relative execution time when prefetching is enabled (right-axis).

performance boost for most of the studied benchmarks. Also in Figure 5.2 and Figure 5.4 we observed that the total power is not affected that much by the addition of the prefetchers. For this reason, we also expect a significant gain in the total consumed energy, thanks to the prefetchers. This is shown in Figure 5.5, where the left axis shows the total consumed energy (micro-joules) while running the benchmarks, and the rigth axis shows the amount of energy reduction (measured in the best prefetching configuration). As expected, an average energy reduction of about 2X is achieved thanks to the proposed prefetching mechanisms. Also, for the codes with very large loops (e.g. *sha1*, *aes-u*, and *md5*) even more than 3X reduction is achievable.

In this chapter we showed that our proposed prefetching mechanisms are implementable and realistic (area increase less than 0.5%), also we showed that the prefetchers increase power consumption only by 11%. This is while they can boost the hit-rate (up to 95%) and performance of most benchmarks significantly (up to 1.8X). Altogether, we showed that our proposed mechanisms can reduce the total energy consumption by an average of 2X compared to the case with no prefetching. For embedded systems, this can mean significantly higher battery life. Next chapter, finalizes this thesis and give our conclusions.

Chapter 6

Conclusions

In this thesis we proposed three simple and low cost instruction prefetching mechanisms (SWP, NLP, and STP) to be used in ultra low-power processing platforms. We studied a wide range of applications and grouped them into two categories: the ones with large computation loops, and the ones with many function calls. We showed that for most of the benchmarks in the first group NLP and STP are able to improve the ICache hit-rate to over 95% with an average execution time improvement of over 2X. While for the second group, we showed that a combination of SWP and NLP can be effective, again leading to a similar improvement in the execution time. By synthesizing our designs using the state-of-the-art technologies we showed that addition of the proposed prefetchers does not increase system's power significantly (less than 12%), and increase total area by less than 1%. Overall, our proposed prefetching scheme allow for an average energy reduction of 1.9X over the range of studied applications.

The future directions include studying the effectiveness of the proposed schemes for multiple threads and extending them to support multi-programming. Also, modifying the compiler to automatically insert software prefetch commands inside the code is another interesting future work.

Bibliography

[1] P. H. Wang, H. Wang, J. D. Collins, E. Grochowski, R. M. Kling, and J. P. Shen, "Memory latency-tolerance approaches for itanium processors: out-of-order execution vs. speculative precomputation," in *Proceedings Eighth International Symposium on High Performance Computer Architecture*, pp. 187–196, Feb 2002.

[2] "Out of order execution." https://en.wikipedia.org/wiki/Out-of-order_execution.

[3] J. Loew and D. Ponomarev, "Aggressive scheduling and speculation in multithreaded architectures: Is it worth its salt?," in *2008 20th International Symposium on Computer Architecture and High Performance Computing*, pp. 11–18, Oct 2008.

[4] K. Thangarajan, W. Mahmoud, E. Ososanya, and P. Balaji, "Survey of branch prediction schemes for pipelined processors," in *Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory (Cat. No.02EX540)*, pp. 324–328, 2002.

[5] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, (Washington, DC, USA), pp. 226–237, IEEE Computer Society, 1996.

[6] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," *SIGOPS Oper. Syst. Rev.*, vol. 30, pp. 222–233, Sept. 1996.

[7] S. Mittal, "A survey of recent prefetching techniques for processor caches," *ACM Comput. Surv.*, vol. 49, pp. 35:1–35:35, Aug. 2016.

[8] J. Pierce and T. Mudge, "Wrong-path instruction prefetching," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pp. 165–175, Dec 1996.

[9] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *IEEE Transactions on Computers*, vol. 48, pp. 121–133, Feb 1999.

[10] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, Mar. 1995.

[11] C. K. Tang, "Cache system design in the tightly coupled multiprocessor system," in *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, AFIPS '76, (New York, NY, USA), pp. 749–753, ACM, 1976.

[12] T. Ungerer, B. Robič, and J. Šilc, "A survey of processors with explicit multithreading," *ACM Comput. Surv.*, vol. 35, pp. 29–63, Mar. 2003.

[13] "Parallel ultra low power (pulp) processing platform." http://www.pulp-platform.org/.

[14] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, (San Jose, CA, USA), pp. 983–987, EDA Consortium, 2012.

[15] A. Heinecke, M. Klemm, and H.-J. Bungartz, "From gpgpu to many-core: Nvidia fermi and intel many integrated core architecture," *Computing in Science & Engineering*, vol. 14, no. 2, 2012.

[16] "Principle of locality." https://en.wikipedia.org/wiki/Locality_of_reference.

[17] A. Teman, D. Rossi, P. Meinerzhagen, L. Benini, and A. Burg, "Controlled placement of standard cell memory arrays for high density and low power in 28nm fd-soi," in *The 20th Asia and South Pacific Design Automation Conference*, pp. 81–86, Jan 2015.

[18] I. Loi, D. Rossi, G. Haugou, M. Gautschi, and L. Benini, "Exploring multi-banked shared-l1 program cache on ultra-low power, tightly coupled processor clusters," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, (New York, NY, USA), pp. 64:1–64:8, ACM, 2015.

[19] S. G. Berg, "Cache prefetching," Tech. Rep. UW-CSE 02-02-04, University of Washington, Seattle, WA, 98195-2350.

[20] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[21] S. P. V. Wiel and D. J. Lilja, "When caches aren't enough: data prefetching techniques," *Computer*, vol. 30, pp. 23–30, Jul 1997.

[22] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, (New York, NY, USA), pp. 176–186, ACM, 1991.

[23] J. E. Smith and W.-C. Hsu, "Prefetching in supercomputer instruction caches," in *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, (Los Alamitos, CA, USA), pp. 588–597, IEEE Computer Society Press, 1992.

[24] T. J. L. Ricardo Bianchini, "A preliminary evaluation of cache-miss-initiated prefetching techniques in scalable multiprocessors," 1994.

[25] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, pp. 7–21, Dec. 1978.

[26] C. Zhang and S. A. McKee, "Hardware-only stream prefetching and dynamic access ordering," in *Proceedings of the 14th International Conference on Supercomputing*, ICS '00, (New York, NY, USA), pp. 167–175, ACM, 2000.

[27] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, MICRO 25, (Los Alamitos, CA, USA), pp. 102–110, IEEE Computer Society Press, 1992.

[28] S. Verma and D. M. Koppelman, "The interaction and relative effectiveness of hardware and software data prefetch,"

[29] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "A study of integrated prefetching and caching strategies," *SIGMETRICS Perform. Eval. Rev.*, vol. 23, pp. 188–197, May 1995.

[30] S. Chung and Y. Kim, "Loop instruction processing using loop buffer in a data processing device having a coprocessor," Sept. 2005. US Patent 6,950,929.

[31] M. Jayapala, F. Barat, T. V. Aa, F. Catthoor, H. Corporaal, and G. Deconinck, "Clustered loop buffer organization for low energy vliw embedded processors," *IEEE Transactions on Computers*, vol. 54, pp. 672–683, Jun 2005.

[32] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, (New York, NY, USA), pp. 40–52, ACM, 1991.

[33] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 12, no. 2, pp. 87 – 106, 1991.

[34] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn’t, and why," *ACM Trans. Archit. Code Optim.*, vol. 9, pp. 2:1–2:29, Mar. 2012.

[35] C.-K. Luk and T. C. Mowry, "Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, (Los Alamitos, CA, USA), pp. 182–194, IEEE Computer Society Press, 1998.

[36] "Internet of things." https://en.wikipedia.org/wiki/Internet_of_things.

[37] "Loop unrolling." https://en.wikipedia.org/wiki/Loop_unrolling.

[38] "Inline functions." https://en.wikipedia.org/wiki/Inline_function.

[39] "Berkley softfloat library." http://www.jhauser.us/arithmetic/SoftFloat.html.