Investigating the viability of adaptive caches as a defense mechanism against cache side-channel attacks

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Thesis

INVESTIGATING THE VIABILITY OF ADAPTIVE CACHES AS A DEFENSE MECHANISM AGAINST CACHE SIDE-CHANNEL ATTACKS

by

SAHAN LAKSHITHA BANDARA

B.Sc., University of Moratuwa, 2015

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2019

Approved by

First Reader

Michel A. Kinsy, PhD
Assistant Professor of Electrical and Computer Engineering

Second Reader

Martin C. Herbordt, PhD
Professor of Electrical and Computer Engineering

Third Reader

Tali Moreshet, PhD
Senior Lecturer & Research Assistant Professor of Electrical and Computer Engineering

Acknowledgments

I would first like to thank my thesis advisor, Professor Michel A. Kinsy for his guidance throughout this endeavor. He always pushed me to put my best effort and aim for the highest target.

Second, I would like to acknowledge all the support I received from my wife. She was always ready to put her work aside and help me in every way possible. While being a busy Ph.D. student herself, she was willing to relieve me of all the house chores during the last month of writing this thesis. She even found the time to proofread my thesis at the end.

Next, I would like to thank my colleagues at Adaptive and Secure Computing Systems Laboratory. I received a lot of support from them, starting from building the BRISC-V platform which was used in this research, discussing implementation details, to proofreading my thesis.

I am thankful to Professor Martin C. Herbordt and Professor Tali Moreshet, for taking time off their schedules to serve on my thesis committee, and providing valuable feedback on the thesis.

Finally, I thank both mine and my wife's parents for the support they provided me, financially and otherwise, during the duration of the master's program at Boston University.

INVESTIGATING THE VIABILITY OF ADAPTIVE CACHES AS A DEFENSE MECHANISM AGAINST CACHE SIDE-CHANNEL ATTACKS

SAHAN LAKSHITHA BANDARA

ABSTRACT

The ongoing miniaturization of semiconductor manufacturing technologies has enabled the integration of tens to hundreds of processing cores on a single chip. Unlike frequency-scaling where performance is increased equally across the board, core-scaling and hardware thread-scaling harness the additional processing power through the concurrent execution of multiple processes or programs. This approach of mingling or interleaving process executions has engendered a new set of security challenges that risks to undermine nearly three decades' worth of computer architecture design efforts.

The complexity of the runtime interactions and aggressive resource sharing among processes, e.g., caches or interconnect network paths, have created a fertile ground to mount attacks of ever-increasing acuteness against these computer systems. One such class of attacks is cache side-channel attacks.

While caches are vital to the performance of current processors, they have also been the target of numerous side-channel attacks. As a result, a few cache architectures have been proposed to defend against these attacks. However, these designs tend to provide security at the expense of performance, area and power. Therefore, the design of secure, high-performance cache architectures is still a pressing research challenge.

In this thesis, we examine the viability of self-aware adaptive caches as a defense mechanism against cache side-channel attacks. We define an adaptive cache as a caching structure with (i) run-time reconfiguration capability, and (ii) intelligent built-in logic to monitor itself and determine its parameter settings. Since the success of most cache side-channel attacks depend on the attacker's knowledge of the key cache parameters such as associativity, set count, replacement policy, among others, an adaptive cache can provide a moving target defense approach against many of these cache side-channel attacks.

Therefore, we hypothesize that the runtime changes in certain cache parameters should render some of the side-channel attacks less effective due to their dependence on knowing the exact configuration of the caches.

Contents

List of Tables

List of Figures

List of Abbreviations

AES	....	Advanced Encryption Standard
ALM	....	Adaptive Logic Module
BRAM	....	Block RAM
FPGA	....	Field Programmable Gate Array
ISA	....	Instruction Set Architecture
LLC	....	Last Level Cache
LRU	....	Least Recently Used
MESI	....	Modified Exclusive Shared Invalid
PDF	....	Probability Distribution Function
RAM	....	Random Access Memory
SRAM	....	Static RAM

Chapter 1

Introduction

1.1 Background

Most modern processors use caches to overcome the “memory wall”, which is the widening gap between processor speed and memory access speed. Memories are orders of magnitude slower compared to a processor. To bridge this gap, multi-level cache hierarchies are implemented between the processor and main memory. Caches are small, fast memories implemented on the same die as the processor, used to store a small but frequently used portion of the main memory.

Typically, caches have a fixed organization. While different processors may use caches with varying sizes, associativities, etc., these parameters remain constant for a given design. Designing cache architectures which allow run-time changes of some cache parameters has been an active research field for a long time. The ability of a run-time reconfigurable cache to improve cache performance and power usage has been studied extensively. However, their ability to defend against cache side-channel attacks is a relatively unexplored research area.

Certain architecture features in modern processors allow transient states which are not specified in the Instruction Set Architecture (ISA) which is an abstract representation of the processor. Although these micro-architectural states do not impact the correctness of the computations, they can be leveraged to leak sensitive information regarding the processes running on the processor. Branch prediction, speculative execution, and caches are such performance oriented architecture features which have been exploited by various attacks. An attacker could exploit certain intrinsic details of a system to create a covert channel which will transfer sensitive information to the attacker.

Side-channel attacks are based on specific implementation details of a computer system rather than any weakness of the algorithms executed on the given system. A computer system could unintentionally leak certain information as byproducts of its normal operation. Measuring these byproducts can be used to uncover sensitive information of the processes executed on the system. Timing, electromagnetic radiation, heat dissipation, power usage, and even the sound produced during computation could be used for this purpose.

Caches are widely exploited in side-channel attacks because they are very effective as covert channels. This is mainly because caches are a shared resource and can be used to share information between processes. Every process running on a processor shares the primary caches. The Last Level Cache (LLC) is shared by processes running on different cores. Their high bandwidth, size and central position in a processor are other factors which makes caches effective side-channels. An important feature of a cache side-channel attack is that it can be software based and the attacker does not require physical access to the target computer system. Cache side-channel attacks are able to circumvent security measures such as privilege checks, address space layout randomization (Hund et al., 2013) (Evtyushkin et al., 2016), etc. and launch same core, cross-core, and cross-VM attacks (Zhang et al., 2012). Recent attack models Spectre (Kocher et al., 2018) and Meltdown (Lipp et al., 2018), both use caches as the covert channel which leaks sensitive information while targeting speculative execution and out-of-order execution respectively as the point of attack. Some of the side-channel attacks will be discussed further in section 4.1.

1.2 Motivation

Previous work has proposed defenses against cache side-channel attacks. These solutions include both software and hardware-based approaches. Generally, Software-based solutions are more specific to a particular attack while hardware-based solutions are more general. Most existing solutions have several limitations which prevent them from being used in real world designs. One major limitation in most of the proposed defenses is the negative impact they have on performance. Other limitations include high resource usage and requiring extensions to the Instruction Set Architecture (ISA).

This work is an attempt to utilize adaptive caches, which is conventionally used to enhance the performance of a processor, as a defense mechanism against cache side-channel attacks. The motivation behind this approach is to provide security against side-channel attacks while minimizing the impact on processor performance.

Cache side-channel attacks depend heavily on the knowledge of the cache organization in a particular processor. Therefore, run-time changes of some key cache parameters will impact the accuracy of these attacks. In this work, we explore whether that can be used as an effective defense mechanism against cache side-channel attacks.

1.3 Threat Model

We assume that the attacker is capable of running arbitrary malicious user space programs on the system. Attacker processes can only access the memory allocated for them and cannot directly access memory regions allocated for other processes. However, the attacker is capable of mounting a cache side-channel attack on either the primary or secondary caches depending on whether the attacker and victim processes are co-located on the same core or located on different cores. The attacker is fully aware of the various parameters of the cache such as size, associativity, and line size. This work does not consider attacks using cache flush instructions. Therefore in this work, it is assumed that the attacker does not have access to such instructions. However, the attacker does have access to high-resolution timers to perform the attack.

1.4 Research Outcomes

There are four major components of this work;

1. Design a cache architecture with run-time reconfigurability to achieve higher flexibility in terms of possible configurations compared to other published work
2. Design a self-aware cache capable of monitoring itself and reconfiguring its resources to achieve a given objective such as performance or low power
3. Investigate the level of defense provided by an adaptive cache against cache side-channel attacks
4. Explore design techniques which will enable an adaptive cache to detect and react to a potential cache side-channel attack

The following four chapters of this thesis are organized according to the above outcomes. The fifth chapter concludes the thesis.

Chapter 2

Reconfigurable Caches

Memory latency and bandwidth have been inherent performance bottlenecks of Von Neumann architecture. This is due to the growing gap between the processor and memory speeds. Caches play a vital role in current processors. They bridge this gap to ensure high performance.

Caches become bigger and slower as they get closer to the main memory. Main memory is extremely slow when compared to the caches and the processor. The processor accessing a piece of data, and the data being available in the cache is referred to as a cache hit. When the data is not present in the cache it is referred to as a cache miss. Cache miss penalty grows rapidly as the accesses move farther way from the processor in the cache hierarchy and can degrade the performance severely. Therefore, in order to achieve maximum performance, computer architects aim to achieve highest possible cache hit rates.

Memory access patterns of computer programs have subtle differences from one another. Cache hit rates are strongly influenced by the specific memory access pattern of an application and the actual organization of the cache. Since the caches are typically designed with a fixed organization, they cannot provide the best performance for all the different memory access patterns of programs unless they have very high associativity. But highly associative caches take up more die area and use more energy. If the caches were able to monitor the memory access patterns and reconfigure their organization, they could minimize cache misses and provide improved performance.

Caches handle memory contents at the cache line granularity. Depending on the exact architecture, the size of a cache line may vary. Line sizes among commercial processors are typically in the order of 10s of bytes. Many Intel processors use 64-byte lines. When a cache miss occurs, a whole cache line is brought into the cache from memory. This is to take advantage of spatial locality in memory. But if a given program does not have high spatial locality in data, the cache will be transferring data from memory unnecessarily, consuming power and bandwidth of the on-chip communication fabric. Usually, the data buses or network on chip has fixed bandwidth and time taken to transfer a cache block grows with the size of the cache block. Therefore, a program with low spatial locality will suffer a higher performance penalty due to how the caches operate. A run-time reconfigurable cache will alleviate this problem by reducing the size of the cache blocks for programs or sections of a program with low spatial locality and increase the size when there is high locality of data.

Cache access time is not the only factor a cache designer should be concerned about. The power density of processors has also been growing rapidly and caches account for a large portion of total processor power. For instance, the cache subsystem in ARM 920T microprocessor consumes 44% of the total power (Segars, 2001) while the instruction cache alone in the StrongARM SA-110 processor consumes 27% of the total power (Montanaro et al., 1997). Authors of (Vijaykrishnan et al., 2000) report that up to 90% of the total system energy could be spent by the memory system. Energy efficiency of caches will also benefit from the ability to reconfigure cache parameters to best fit the current application. With the run-time reconfiguration capability, a smaller cache would be able to provide similar performance to a bigger, highly associative cache. The smaller cache will consume less power since it has fewer SRAM blocks. Also, it would be possible to shut down part of the memory blocks if the current program is not affected by the reduced cache capacity/associativity.

It has been shown that a reconfigurable cache can provide improved performance over a large set of applications and a wide range of performance objectives such as hit rates, total execution time, energy for a particular cache level, main memory energy, total memory system energy, etc. Kadayif et al. in (Kadayif et al., 2001) explore the performance variations of different applications with cache size, line width, and associativity variations. Authors of (Kumar and Singh, 2016) analyze the effect of replacement policy on cache performance. In (Tao et al., 2008), the impact of changing cache associativity, cache block size, and total cache size on the L1 hit rate are analyzed across several applications. The authors show that different applications have different optimal cache configurations.

This work implements a new cache architecture which will provide higher flexibility in terms of the number of possible configurations compared to previous work. A key feature of the proposed architecture is the ability to explore different tradeoffs between the level of reconfigurability and resource usage at design time. The same architecture could be implemented with different levels of resource usage and the number of possible configurations.

2.1 Related Work

Activating and deactivating ways in a cache is the earliest form of dynamic reconfiguration employed (Albonesi, 1999). This is also the simplest in terms of modifications made to a non-reconfigurable cache architecture. Authors of (Malik et al., 2000) describe a unified instruction and data cache in which each way can be locked as "instruction only", "data only" or "instruction and data". This dynamically allocates the available cache ways between instructions and data. This cache architecture was implemented in M·CORE M340 processor, an embedded processor by Motorola, Inc. In (Zhang et al., 2003), the authors describe way concatenation which reduces cache associativity while utilizing the full cache by increasing the line width.

The Variable-Way (V-Way) cache proposed by the authors of (Qureshi et al., 2005), is capable of adjusting the number of ways in a cache set at run-time. In a V-way cache, different cache sets could have different associativity. This is achieved by decoupling the data and tag storage and allocating double the number of tag entries compared to a non-reconfigurable cache with the same configuration. Data is stored in a single direct mapped array. An index to access the data array is stored with every tag. With this structure, the number of ways in a set does not have to be a constant. It could vary according to the run-time requirements as long as there is free tag storage available for a given set. However, one drawback of this architecture is that tag comparison and data accesses cannot be done in parallel.

The "adaptive group-associative cache" described in (Peir et al., 1998) attempts to leverage the fact that recently used cache lines are not distributed uniformly across all cache sets. It provides better performance by dynamically partitioning cache sets into groups and allocating the underutilized cache sets to store recently used cache lines if they are evicted due to a conflict with another line. Therefore, a part of the cache sets is dynamically turned into a victim cache. Architectures described above provide limited reconfigurability due to the lack of flexibility in their structure. They rely on manipulating how ways are allocated as the means of reconfigurability.

Authors of (Ranganathan et al., 2000) discuss the benefits of dynamically partitioning the caches and using the different partitions for different processes. They describe two methods of dynamically partitioning the SRAM storage. The first is using cache ways as the partitions. The second method is overlapped wide-tag partitioning where the tag storage is extended to support different partition sizes. The authors also discuss how to maintain data consistency when the cache is being dynamically reconfigured. First method proposed is "cache scrubbing" where data is written back to lower levels of the cache hierarchy when a cache is reconfigured. An alternative approach is to maintain contents of the cache across reconfiguration and move data to correct partitions. But this requires storing additional state information with data.

Authors of (Tradowsky et al., 2016) describe a highly reconfigurable cache architecture based on modules referred to as “Tiles”, which comprise of memory to store cached data, tags and other meta-data and logic for read/write operations and tag comparison. Cache size, line width, and associativity are reconfigured by reallocating the Tiles to different sets or ways. Wider cache lines are emulated by the cache controller by concatenating lines from multiple tiles. This architecture also provides the ability to select a replacement policy at run-time. Although this provides better reconfigurability, there are instances where the resources are not utilized maximally. Because tag storage is allocated statically in each ‘Tile’, most of the tiles will have unused tag storage when multiple tiles emulate wider cache lines. When fusing multiple sets to emulate larger line width while not reducing associativity, additional hardware is required to make sure that a given cache line is stored in the same way across multiple sets.

Chen et. al. in (Chen et al., 2007) describes the main drawback of partitioning a cache based on set associativity; the number and granularity of partitions being limited by the associativity of the cache. The authors propose using multiple physically partitioned sub arrays of varying sizes with overlapped wide-tag partitioning. However, this architecture is still limited by the fixed number and sizes of the memory sub-arrays.

Amoeba-Cache described in (Kumar et al., 2012) explores the idea of eliminating the dedicated tag storage and treating the storage array as uniform and morphable between tag and data storage. This allows each cache block to have a variable size.

Tag and data are stored interleaved in the same SRAM array. One drawback of this approach is that this significantly complicates the cache reading logic as there is no dedicated place for the tags. Any given word in the storage array could either be a tag or a data word. To overcome this, an **N/2** number of tag comparators are used when the total number of words in a single line is **N**. Since the cache block sizes vary dynamically, the storage array is fragmented and deciding where to insert data is a challenge. Additional logic to search for empty words in the storage array or additional memory to keep track of empty words is required.

Maintaining cache coherence becomes challenging when the caches are reconfigured at run-time, especially if the cache line width is dynamically changing. This is due to the fact that cache coherence is usually managed at cache line granularity. When the cache line size is not consistent, there are three main challenges faced by the coherence protocol; (i) at which granularity the coherence states should be maintained? (ii) how to support variable granularity read sharing? (iii) what is the granularity of write invalidations? (Kumar et al., 2012)

In (Dubnicki and LeBlanc, 1992), the authors propose a coherence scheme for adjustable block size caches. In the proposed coherence scheme, caches pass additional information with the requests which are used to determine the size of the transfer. The reconfigurability of line width is limited to two possible configurations. Cache controller could either update the whole cache line or half of the line. Amoeba cache (Kumar et al., 2012) handles coherence with varying block sizes by dictating a fixed granularity at which cache coherence is managed. The coherence block size is large enough that any sub-block involved in a coherence operation always lies within an aligned coherence block. One drawback of this approach is unnecessary invalidations in caches which maintain blocks much smaller than the fixed coherence block size.

Other works discuss sub-block coherence protocols (Kadiyala and Bhuyan, 1995)

(Anderson and Baer, 1994). These work do not directly focus on reconfigurable caches, but attempt to improve performance by operating on sub-blocks of cache blocks unless it is necessary to operate on the entire block. Sub-block coherence protocols could be used to maintain cache coherence in a system with run-time reconfigurable caches. These schemes incur additional storage overhead to maintain the status of each sub-block.

2.2 Limitations of Previous Work

There are several recurring observations in previous work on reconfigurable caches.

- A consistent trend across most previous work is using cache associativity as the source of reconfigurability. This severely limits the level of reconfigurability because the number of partitions to be moved around and reconfigured is limited to the number of cache ways.
- Dedicated tag storage is largely underutilized in certain configurations of a reconfigurable cache.
- Reconfigurability at a cache line granularity requires a lot of additional logic and storage for housekeeping tasks. Such a level of reconfigurability may not be necessary. Therefore, it may be worthwhile to strike a balance between the granularity of reconfigurable elements and resource usage. But this is specific to the exact designs in question. Different designs may fit at different locations on a spectrum of possible configurations and resource usage.

2.3 Proposed Architecture

The proposed architecture is made of a number of small independent memory blocks. Reconfiguration is done by changing the logical organization of these blocks of memory. This design is largely inspired by the tile based architecture proposed in (Tradowsky et al., 2016). But instead of making blocks which consist of tag and data storage, and tag comparison logic, this work implements only the storage elements as tiles/blocks. The blocks are one word (4 Bytes) wide. The number of memory blocks and the size of memory blocks are configurable at synthesis time. The proposed architecture eliminated dedicated storage for tags (and other meta-data), and considers the storage array to be uniform and morphable to store either data or tags depending on the current configuration.

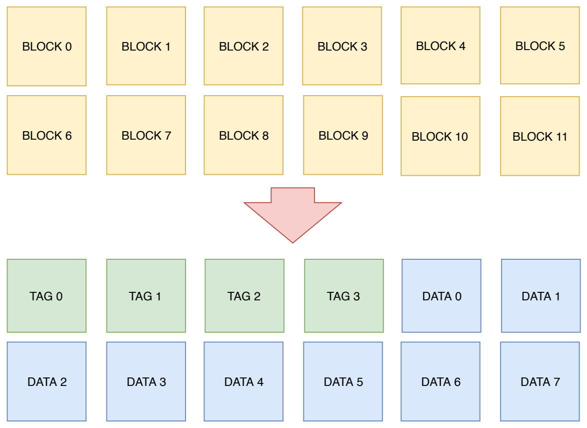


Figure 2·1: Memory block allocation for data and tag storage

Figure 2·1 depicts how a uniform set of memory blocks are designated as tag (tag and other meta-data) storage and data storage. This allocation translates to the logical organization depicted in figure 2·2, which has two ways, 8 bytes (2 words) wide cache blocks and $2 \cdot L$ cache sets (L represents the number of lines in a memory block). It can also translate in to the organization in figure 2·4, which is a direct mapped cache with 8 byte blocks and $4 \cdot L$ cache sets. The difference is made by how the cache controller interprets the logical organization of the blocks. Figures 2·5 and 2·3 depict how the same number of memory blocks (12) can be reorganized to have vastly different configurations. Figure 2·5 is a direct mapped cache with 16 bytes wide cache lines and $2 \cdot L$ cache lines. Such a structure is suited to an application with good spatial locality. Transfers bring more data in to the cache due to wide cache blocks. Because of good spatial locality, there is a high probability of the data fetched being used next. This improves the hit rate. Only 10 memory blocks are utilized under this organization. The unused blocks could be shut down using clock gating in order to save power. The high associativity of the 4-way set associative cache with 4-byte blocks and $(1 \cdot L)$ lines shown in figure 2·3 is suitable for an application with a memory access pattern which causes more conflict misses. This organization utilizes only 8 memory blocks out of the 12.

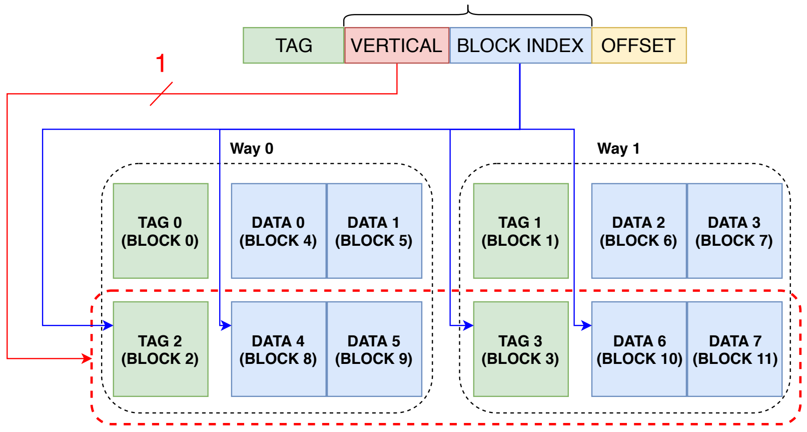


Figure 2·2: 2-way set associative cache organization with $2 \cdot L$ sets and 8-byte lines

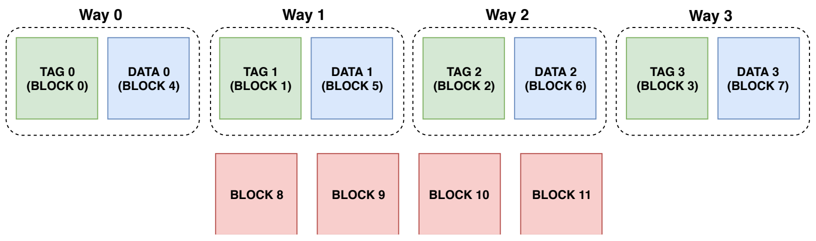


Figure 2·3: 4-way set associative cache organization with L sets and 4-byte lines

It is evident that the number of independent memory blocks determines the number of possible configurations for the cache structure. Figure 2·6 plots the number of

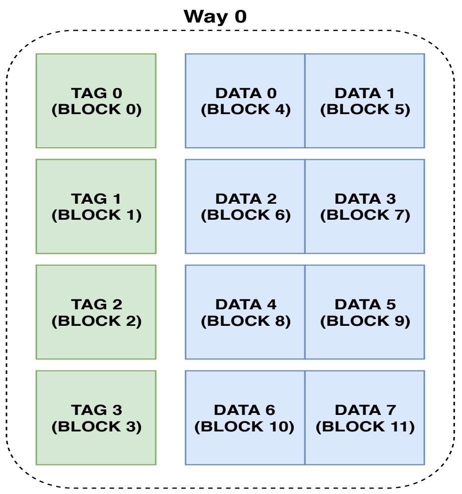


Figure 2·4: Direct mapped cache organization with $4 \cdot L$ sets and 8-byte lines

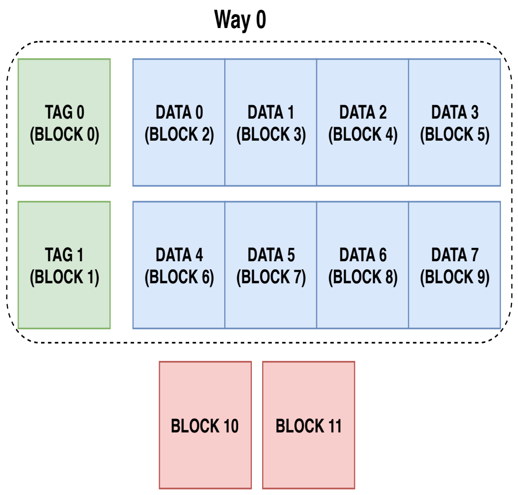


Figure 2·5: Direct mapped cache organization with $2 \cdot L$ sets and 16-byte lines

total configurations and number of configurations which utilize at least 50% of the memory blocks, against the number of memory blocks. Higher number of memory blocks means higher number of unique configurations. However, it is not feasible to indefinitely increase the number of blocks due to resource constraints. Impact of increasing block count on resource usage is discussed in section 2.5.

This architecture is based on a standard snoopy L1 cache. It is a blocking cache with 1 cycle pipelined access. It implements write-back with write allocate policy. True Least Recently Used (LRU) replacement and MESI cache coherence is supported. This will be referred to as the "baseline cache" hereafter. The basic components of the cache are depicted in figure 2·7. All the components were modified in varying degrees to achieve the run-time reconfiguration capability.

All the simulation results were generated using Mentor Graphics® 'ModelSim'. The shared bus based cache hierarchy was adopted from the BRISC-V platform (Bandara et al., 2019). The cache hierarchy has two levels of caches with private

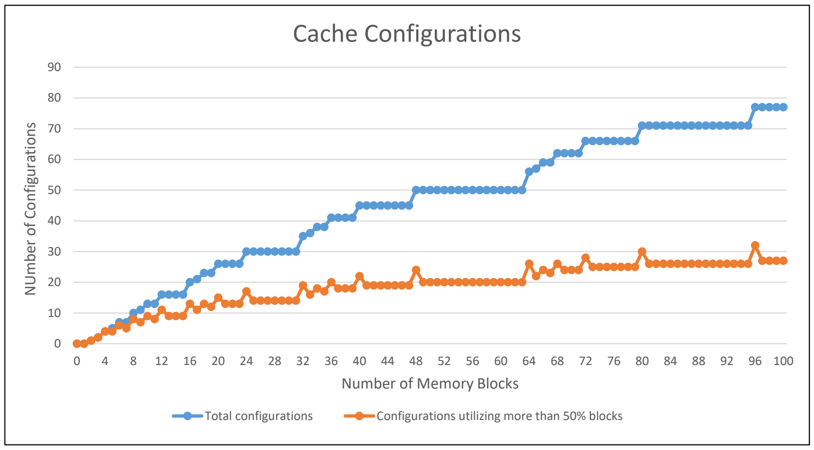


Figure 2·6: Number of possible configurations against the number of memory blocks

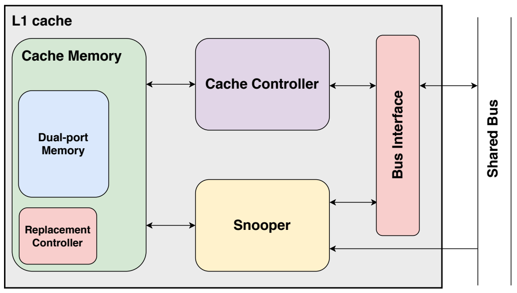


Figure 2·7: Baseline L1 cache design

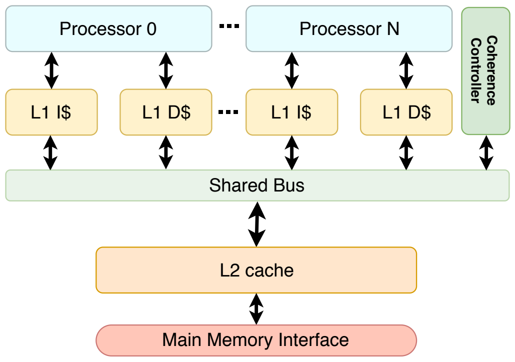


Figure 2·8: BRISC-V cache hierarchy

L1 instruction and data caches, and a unified L2 cache. The shared bus width was configured to be 4 Bytes for all simulations. All synthesis results provided here are based on a Cyclone V FPGA (Part number 5CGXFC9E7F35C8). Synthesis was performed using ‘Quartus Prime Lite Edition’ (version 18.0).

2.3.1 Cache Memory

The L1 cache used as the starting point of this design is implemented in a highly modular fashion. ‘Cache memory’ is the module which instantiates the block RAMs storing tags and data. Replacement logic is also implemented within the cache memory module. Cache memory receives read, write, invalidate signals along with data, tag and other meta-data from the cache controller. The ‘replacement controller’ module inside the cache memory tracks the empty lines and recently used lines for each set and implements Least Recently Used (LRU) replacement policy. Cache memory is the module which required most changes from the BRISC-V baseline design to achieve run-time reconfigurability.

The main challenge introduced by eliminating dedicated tag storage and treating the memory array as uniform is tracking what is stored in each block of memory. Two tables are used to track the contents of the memory blocks. These tables are referred to as ‘tag_registry’ and ‘data_registry’ from here onwards. Tag_registry is a 2D array of registers which tracks the memory blocks which stores tags and other status bits. Data_registry is a 3D array of registers and it tracks the memory blocks which stores data.

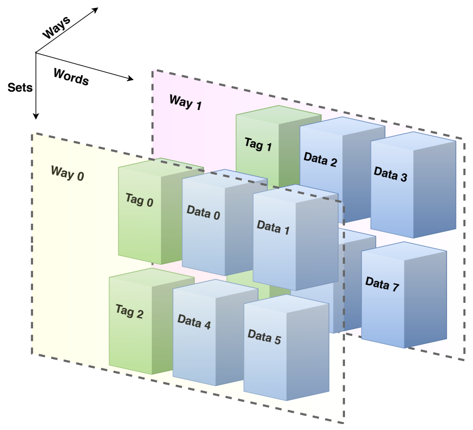


Figure 2·9: Cache memory as a 3D structure

We can visualize a cache as a 3D structure with the cache ways, cache sets, and words in a cache block as the three dimensions. The memory blocks fit into different positions on the 3D structure as depicted by figure 2·9. This is a conceptual visualization and does not represent the physical organization of the memory blocks. The tag_registry and data_registry structures track the positions of the memory blocks in the 3D structure of the cache. Since the tag blocks are located along only two dimensions of the 3D structure, a 2D array is used to track the location of tag memory. Since memory blocks are one word wide, multiple memory blocks fit in the same cache line. The 3rd dimension of our 3D visualization is along the cache lines. Therefore, a 3D array, 'data_registry' is used to track the location of each piece of data within the 3D structure. Note that tags and other status bits for cache lines share the same memory blocks. Each line in the memory block stores tag and status bits for the same cache line.

The memory block which would store a given piece of data depends on the current configuration of the cache. Therefore, data read from individual memory blocks should be routed to the proper destinations. Tag data should be routed to tag comparators while the data from the correct cache way should be routed to the data output port when there is a cache hit. Values stored in tag_registry and data_registry are used as the control inputs to the multiplexers which route the readouts of memory blocks to correct destinations.

While the tag_registry and data_registry structures enable routing the readouts of memory blocks, using the same structure to set the inputs to the memory blocks resulted in a significant impact to the resource usage. For certain configurations, resource usage was as high as 9 times the resource usage of the baseline cache. Setting the write enable signals of data storage blocks and routing correct data words to the inputs of memory blocks accounts for most of the additional logic. This is because each block of memory required logic which compares its own block number to the contents of the tag_registry and data_registry. Replicating this logic for each memory block uses a lot of FPGA resources.

This issue was solved by creating a distributed registry apart from the centralized data_registry and tag_registry. Each memory block has four registers associated with it. Those record the position of the memory block along each dimension of the 3D cache structure and whether the block stores meta-data (tag + status bits). These values combined with other signals are used to control the 'write enable' and 'write data' inputs of individual memory blocks. This method results in lower resource usage compared to using the tag_registry and data_registry for the same purpose.

The baseline cache used for this design implements true LRU replacement policy. The replacement policy is managed by the 'replacement_controller' module inside the 'cache_memory' module. Replacement_controller was extended to support a variable number of ways and still provide true LRU replacement. The Memory array which stores the states to determine the least recently used way is extended to support the maximum number of ways the cache could reconfigure itself into ('MAX_WAYS' parameter). This additional storage capacity is not utilized when the cache has less than maximum associativity. But the resource usage is not significant compared to the total BRAM usage. If the maximum number of ways is 'W', and maximum number of cache sets is 'S'; the BRAM usage for the LRU storage is $\Big(W \cdot log_2(W) \cdot S\Big)$. For instance, the BRAM usage for the LRU unit of a cache with a maximum of 64 sets, and maximum associativity set to 8 amounts to 3.63% of total BRAM usage. 40KB of memory was allocated for tag and data storage in this configuration. One can always use a replacement policy such as random replacement if the storage requirement for true LRU replacement deemed too high.

A cache typically uses a number of comparators equal to the number of ways in the cache to perform tag comparison for all the cache ways in parallel. There are cache architectures which reduce this resource usage by serializing access to different ways of the cache, but that is not the default configuration considered here. The maximum number of ways in the reconfigurable cache is controlled by the 'MAX_WAYS' parameter. If the proposed architecture was to follow the implementation strategy of the baseline cache, it would require 'MAX_WAYS' number of tag comparators to support all possible configurations. But the resource usage for routing stored tag values to the comparators is very high. Since the memory array is uniform and any given memory block could be storing either tags or data, there should be large multiplexer trees which allows routing the output of any memory block to tag comparators. A revised design introduced the 'MAX_TAG_BLOCKS' parameter which specifies the maximum number of memory blocks which will be used to store meta-data. This puts a restriction on the maximum associativity of the cache. By following the memory block allocation strategy depicted in figure 2·1, the resource usage was minimized. With this optimization, only the output of 'MAX_TAG_BLOCKS' memory blocks need to be routed to tag comparators.

Consider the following notation for different cache parameters; maximum number of ways = W, maximum number of blocks storing tags = B, and maximum number of tag bits = T. The cache requires W comparators, one each for the W ways. Each bit of the comparator requires a B:1 mux to route a bit of data from memory blocks to the comparator. In total, this is $(W \cdot T)$ B:1 muxes. This maps to a tree of cascaded multiplexers. These wide and deep multiplexer trees negatively impact both resource usage and operating frequency. To further reduce resource usage, an alternative approach is taken. Instead of routing data to different comparators, a comparator is instantiated for every memory block which could potentially store tags. The number of comparators is governed by 'MAX_TAG_BLOCKS' parameter. The upper bound for the number of comparators is N/2 where N is the total number of memory blocks in the cache. This is for a configuration where every other memory blocks store meta data. Meta-data will be stored for each word of data in this extreme scenario.

Table 2.1: Resource usage for comparators and wide muxes

Component	ALM usage
20-bit comparator	32
20-bit wide 8:1 mux	142
1-bit wide 8:1 mux	9

This new configuration requires B comparators, and W B:1 multiplexers to route the outputs of relevant tag comparators to a one-hot decoder in order to determine cache hit or miss. By comparing the resource usage of a comparator and a wide mux, it can be seen that the resource utilization can be reduced by trading off comparator count for mux count.

Table 2.1 includes the synthesis results for several configurations which depict the tradeoff discussed above. Tag width is assumed to be 20 bits. The cache is assumed to support up to 4 ways and there are 8 memory blocks which store tag bits. A 20-bit comparator uses 32 Adaptive Logic Modules (ALMs) in the Cyclone V FPGA. (ALMs in Cyclone 5 devices include an 8 input LUT and 4 registers.) If we are to use 4 comparators, we need a 20 bit wide 8:1 mux in front of every comparator. Since every mux tree uses 142 ALMs, the total resource usage will be 696 ($32 \times 4 + 142 \times 4$) ALMs. With the alternative approach used in this architecture, every memory block which could store a tag has a comparator accompanying it. It will also use 4 one bit wide 8:1 multiplexers. The total resource usage for this configuration will be 292 ($32 \times 8 + 4 \times 9$) ALMs. This is a 58% reduction in resource usage.

2.3.2 Cache Controller

One of the challenges in designing a reconfigurable cache is maintaining data consistency across reconfigurations. This work solves this challenge by writing back the dirty cache lines to lower level caches before starting the reconfiguration sequence. The cache controller was modified to perform this task. The controller state machine was extended to include states which reads every line in the cache and write back if the line is dirty. If the current number of ways is ‘w’ and current number of sets is ‘s’; the upper bound for the number of write back request is $(w \cdot s)$. But with real workloads, number of dirty lines in the cache at a given time is far less than the upper bound.

The baseline cache hierarchy used for this design uses a shared bus for communication between caches. In order to make the reconfigurations faster, a cache performing a reconfiguration is given full control over the bus. This speeds up the phase of reconfiguration in which the dirty cache lines are written back to the lower level caches. To obtain control over the bus, the cache controller first initiates a dummy read to the memory. Once it receives the response back from the L2 cache, instead of releasing control of the bus, it holds the bus and starts reconfiguration sequence. The dirty line encountered could now be quickly written back to the lower levels in the cache hierarchy without waiting for the control of the bus. Cache controller extensions also include states to obtain control over the shared bus and hold it until the end of reconfiguration.

One design challenge with a run-time reconfigurable cache is how to split the addresses into offset, index and tag components. When the cache block size, and/or number of sets change, the number of address bits allocated for offset, index and tag also must change. While this seems trivial, it consumes a lot of hardware resources if not implemented carefully. A separate sub module named ‘address_splitter’ is implemented to split the address into offset, index and tag based on the current configuration of the cache. A simple implementation using just the shift operations resulted in resource usage of 262 ALMs for a single address splitter module. But an optimized implementation which uses shifting and bit slicing inside multiple nested generate loops reduced the resource usage to 162 ALMs.

Reconstructing an address from offset, index and tag values also become a challenge when the number of bits in each of those components is not constant. The cache controller has to construct the addresses of evicted cache lines when a dirty line is evicted and has to be written back to the level below. Address is constructed by concatenating the evicted tag read from the tag storage, index accessed and a zero offset. With dynamically changing index and tag widths, those cannot be concatenated correctly since Verilog does not support variables as bit slice indices.

Therefore, an ‘address_builder’ module was implemented to construct the addresses correctly. The ‘address_builder’ module takes the current tag, index and offset as the inputs, and generates multiple signals by concatenating different bit slices from each of the three inputs. Then it outputs the correct concatenation of bits according to the current configuration of the cache.

2.3.3 Snooper

A snoopy L1 cache from BRISC-V platform is used as the baseline for this design. The snooper module performs different operations in order to maintain cache coherency across the multi-core system. The snooper module's major tasks are listening to the shared bus and performing cache line invalidations or write-backs depending on the current transaction on the bus and the current contents of the local cache. Cache coherence is handled at cache block granularity. When we have run-time reconfigurable caches, the block sizes could change at run-time and different caches in the same coherence domain could have different block sizes from one another. Therefore, read sharing and write invalidations could be relating only to sub blocks in a cache with larger cache blocks. On the other hand, a cache with smaller cache blocks might have to invalidate or write back multiple lines in response to a single request on the bus.

As described in the related work section, one approach to face this challenge is to maintain coherence state for sub-blocks and perform coherence operations on sub-blocks. This alone does not address the situation where a cache with smaller blocks having to invalidate/write back multiple lines. Another approach is to declare a common coherence block size across the coherence domain. This still does not directly resolve the issue faced by the cache with smaller blocks. Snooper state machine still requires modifications to operate on multiple cache lines per request. Even beyond that, the main drawback is unnecessary coherence operations due to the large coherence block size.

This work focuses on making modifications to the snooper module in order to work around the challenge introduced by variable block sizes. It utilized the modifications made to the bus interface and the shared bus described in sections 2.3.4 and 2.3.5. The snooper module receives information regarding the size of the block with every request on the bus. Snooper module receives another signal from cache memory regarding the current line width of the local cache. Snooper control FSM is extended to determine the number of coherence operations required based on the size difference between the request on the bus and the current configuration of the local cache. Then it performs the required coherence operations. If the snooper has to issue multiple write back requests, it is allowed to hold the bus control across those transactions.

2.3.4 Bus Interface

The bus interface module bridges the gap between the shared bus width and the cache line width. The bus interface of the baseline cache was also modified in this work to support bus transactions with variable block sizes. Current value of 'OFFSET_BITS' parameter for the cache lines is given as an input to the bus interface. Using that information, the bus interface generates the necessary number of bus requests to send or receive data words to/from the level 2 cache. Bus interface also puts the current width information on the bus when issuing any request.

2.3.5 Shared Bus

The shared bus in the baseline cache hierarchy only supported transactions of fixed sized cache blocks. This was a reasonable assumption since the caches all shared the same line width. But in order to enable dynamic and heterogeneous line widths, the shared bus was also modified. Additional lanes are added to the bus to send information regarding the number of words related to the current request. Every request on the bus is accompanied by information regarding the number of words. When issuing a request, a cache also puts the current value of 'OFFSET_BITS' parameter on the bus. This allows other caches snooping on the shared bus to determine how many words should be invalidated or written back. It also informs the level 2 cache about how many words to be returned to the requesting cache in case of a read request or how many data words it will receive in case of a write back request.

2.4 Results and Evaluation

This section provides simulation and synthesis results for the run-time reconfigurable cache and evaluates its performance. Six programs from SPEC and PARSEC benchmark suits were used to evaluate the performance. Since the RTL for the reconfigurable cache architecture was implemented as a part of this work, a cycle accurate RTL simulation was done instead of a software simulation of the cache architecture. Memory request traces for the benchmark programs were collected using Intel PIN tool (Reddi et al., 2004). Due to the long simulation times of cycle accurate RTL simulations, only the first 1,000,000 memory requests for each program were simulated.

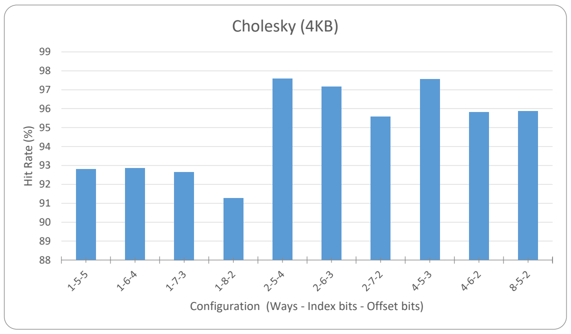
Firstly, in order to understand the different cache configurations that provide the best performance for each benchmark program, the memory traces were simulated on different configurations of the baseline cache. A simple cache hierarchy with L1

Table 2.2: Cache configurations used for simulations

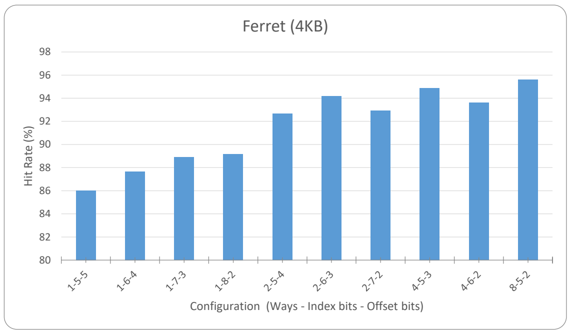
Parameter	Values
Associativity	1, 2, 4, 8, 16
Index bits	5, 6, 7, 8
Offset bits	2, 3, 4, 5, 6

instruction and data caches and unified L2 cache was used for all the simulations. Only the requests to data cache were simulated. Since the instruction cache receives no requests, it does not initiate any bus transactions. This allows the data cache to exclusively utilize the bus with no contention. A large L2 cache with 1024 sets and 16 ways was used to minimize the effects of the L2 cache capacity on the performance of the primary cache. Memory requests were fed continuously to the cache directly from the testbench. Therefore, the total cycles count provided in the results is the number of cycles spent on memory operations alone. Every benchmark memory trace was simulated on 100 different cache configurations, varying the number of cache ways, number of cache sets and cache block size. The different values used for each parameter are presented in table 2.2. A limited number of interesting observations analyzed in the following section.

2.4.1 Performance Comparison



(a)



(b)

Figure 2·10: Hit rates for different configurations of a 4KB cache: (a) Cholesky; and (b) Ferret

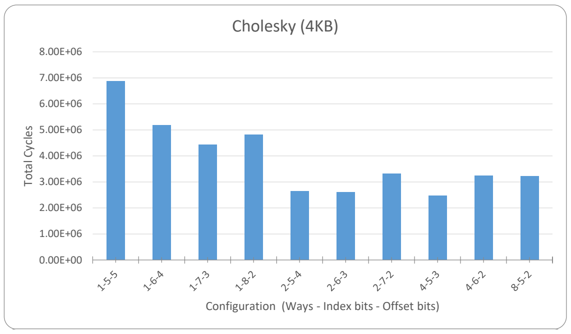
Throughout the remainder of this thesis, cache configurations will be referred to in the following format.

(Number of ways - Number of index bits - Number of word offset bits)

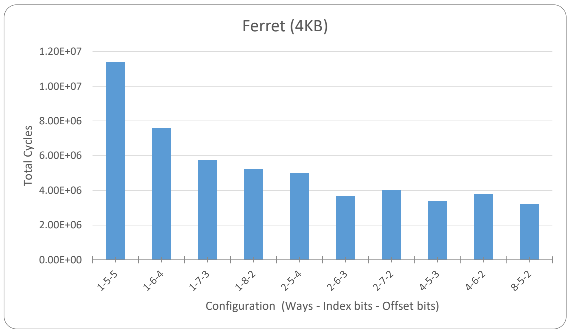
Index bits and offset bits translate to the number of sets and number of words in a cache line (line width) as follows;

$$Number\,of\,sets = 1 << Index\,bits; \qquad Line\,width = 1 << Offset\,bits$$

The first scenario analyzed is for a 4KB cache. Figure 2·10 shows the hit rates for different configurations of a 4KB cache for benchmark programs “Cholesky” and “Ferret”. As one can clearly see, the best configuration for Cholesky is a 2-way set associative cache with 32 sets (5 index bits) and 16-word (64-byte) cache lines (4 bits for word offset). This configuration is referred to as (2-5-4) from here onwards. The optimal configuration for ferret is an 8-way set associative cache with 32 sets and 4-word (16-byte) cache lines (2 bits for word offset). This configuration will be referred to as (8-5-2) hereafter. It is clear that the two programs need vastly different configurations to achieve their highest hit rates.



(a)



(b)

Figure 2·11: Total cycles for memory operations for different configurations of a 4KB cache: (a) Cholesky; and (b) Ferret

Figure 2·11 shows the total cycle count for memory operations with different cache configurations for the two programs. We can calculate the total number of cycles required to run the two programs back to back. For the (2-5-4) configuration, the two programs will take 7,617,567 (2,649,283 + 4,968,284) cycles for memory operations. For the (8-5-2) configuration, the total cycles will be 6,415,074 (3,219,825 + 3,195,249). But if a run-time reconfigurable cache was used, both programs would run under optimal cache configurations. This would give a total cycle count of (2,649,283 + 3,195,249 + reconfiguration cycles). Which is the same as (5,844,532 + reconfiguration cycles). Without considering the cycles spent on reconfiguration, the optimized cache configurations have reduced the total number of cycles by 570,542 cycles (8.89%). As long as the cycles required to reconfigure the cache is less than 570,542, the reconfigurable cache has a positive impact on the performance of the system by reducing execution time. The same scenario was simulated using the reconfigurable cache RTL to determine the total number of cycles required to run the two programs one after the other, including the cycles spent on reconfiguring the cache.

The first experiment was running 'Cholesky' and 'Ferret' back to back starting with (2-5-4) configuration which is the optimal configuration for 'Cholesky' program. After the first program finishes executing, cache reconfigures to (8-5-2) configuration, which is the optimal configuration for 'Ferret' program. This setup takes 5,845,371 cycles to complete. The cycles taken for reconfiguration is just 839. It needs to be noted that bus contention from other caches is not taken into account in this simulation. Even if the bus contention is taken into consideration, the number of cycles saved by using the optimal cache configuration for each program is still several orders of magnitude larger than the cycles taken to reconfigure the cache. When the programs were executed in reverse order starting with (8-5-2) configuration and reconfiguring to (2-5-4) configuration for the 'Cholesky' program, the total cycles will be 5,847,817 cycles with 3285 cycles spent on reconfiguration. The difference in the number of cycles spent on reconfiguration is due to the write-back step taking place at the beginning of the reconfiguration sequence. Cache controller first writes back all the dirty lines to the L2 cache before reconfiguration is done. Since different programs have different memory access patterns, the number of dirty lines in the cache at the end of execution is different from one to another.

An observation which could be made from the hit rate and total number of cycles graphs for different cache configurations is that highest hit rate does not always result in the lowest cycle count. This is due to two reasons. First is, wider cache lines tend to provide good hit rates in most applications unless the application has extremely low spatial locality. Second is, transferring wider cache blocks from memory/L2 cache takes more cycles compared to smaller cache blocks. Up to a certain extent, higher hit rates give lower number of total cycles since the time to fetch data from memory/lower level caches is saved due to higher hit rates. But after a certain cache line width, cycles taken to transfer the cache lines becomes higher than the number of cycles saved by the higher hit rate. Reason for the transfers taking different number of cycles is that the shared bus has a fixed width. For these simulations, a bus width of 4 bytes has been used. Total cycles for 'Cholesky' benchmark depicted in figure 2·11 (a) provides such a scenario. We have based our decision to select configuration (2-5-4) for 'Cholesky' program on the highest hit rate achieved for the same cache size as shown in figure 2·10 (a). But when we analyze the total cycles chart, we can see that the (4-5-3) configuration actually gives a lower cycle count.

When comparing two configurations with wider and narrower cache lines, one with wider cache lines might provide a higher hit rate, but will use the bus for more cycles to transfer the cache lines. Therefore, it might use more total cycles compared to a configuration with slightly lower hit rate, but smaller cache lines. If the target is to minimize execution time, the second configuration might be preferable even though it has a slightly lower hit rate. But execution time is not the only factor a designer might want to optimize. Power usage is another factor. Lower hit rates mean that the rest of the memory hierarchy and the communication medium will be called upon to service memory requests from the cache with lower hit rate more often compared to a cache with a higher hit rate. Therefore, we cannot ignore hit rate and focus only on total cycles for memory accesses. To find the best cache configurations, we must consider both the hit rate and the total cycles. In order to do so, we can consider the hit rate normalized by the total number of cycles as a metric for cache performance. Figure 2·12 plots the ratio between hit rate and total cycles (multiplied by 1,000,000 to avoid extremely small numbers) for different configurations. Once the hit rate is normalized by the total cycles, we can see that configuration (4-5-3) is the best configuration for the 'Cholesky' program.

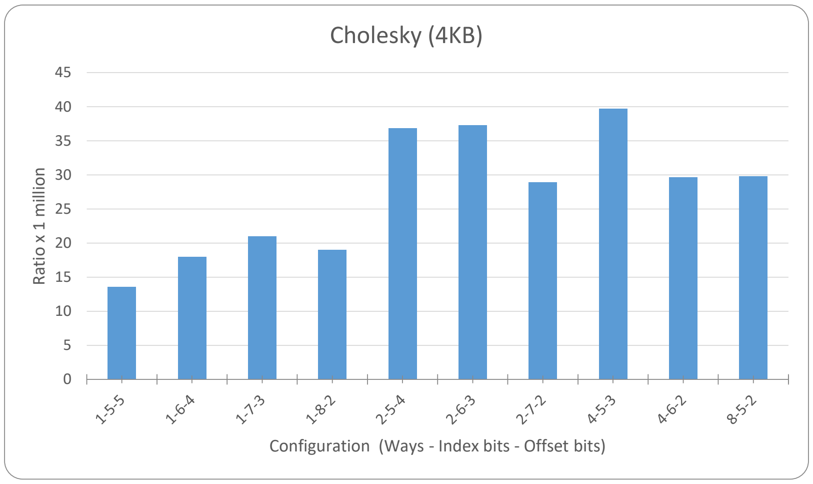


Figure 2·12: Hit rate normalized by total cycles for Cholesky benchmark program

While the total cycle count for program execution is not available at run-time for an optimization logic to decide optimal cache parameters, it is possible to use performance counters built in to the cache to estimate the time taken to complete a given number of memory accesses. There should be a minimum number of memory accesses after a reconfiguration, and before checking whether another reconfiguration is required. If the cycle count since the last reconfiguration is available, it can be used to normalize the hit rate and make the measurement more comparable with previous hit rates or predefined threshold hit rates.

2.4.2 Synthesis Results

Table 2.3 presents the synthesis results for a non-reconfigurable cache as a baseline for comparisons. The cache considered is a 4-way set-associative cache which has 256 cache sets and 16-byte cache lines. Table 2.4 presents the synthesis results for a reconfigurable cache with same memory usage. Results for the cache memory is provided for two different configurations; i) 20 memory blocks, and ii) 40 memory blocks.

Table 2.3: Synthesis results for individual components of the non-reconfigurable cache

Component	BRAM bits	BRAMs	Registers	ALMs	F_{max}(MHz)
cache controller	0	0	604	884	146.43
cache memory	159744	37	63	877	104.14
snooper	0	0	405	427	216.31
bus interface	0	0	486	729	211.06

Table 2.4: Synthesis results for individual components of the reconfigurable cache

Component	BRAM bits	BRAMs	registers	ALMs	F_{max}(MHz)
cache controller	0	0	724	1239	136.72
cache memory(20 blocks)	161792	41	305	2745	111.07
cache memory(40 blocks)	164352	82	539	5842	103.7
snooper	0	0	418	737	181.85
bus interface	0	0	711	739	187.65

Resource usage for the reconfigurable cache is higher than the non-reconfigurable cache for every component. Cache memory is the module which shows the highest increase in resource usage. Since most of the modifications from the baseline design were made to the cache memory module, increase in resource usage was expected.

Also as expected, the resource usage for the cache memory increases with the number of memory blocks. Compared to the non-reconfigurable cache, Adaptive Logic Module (ALM) usage increases by 213% and 566% respectively for the 20 memory block and 40 memory block configurations of the reconfigurable cache.

An interesting observation is that the estimated maximum operating frequency (F_{max}) for the 20 memory block configuration of the cache memory module is higher that the non-reconfigurable implementation of the same module. This is due to the reconfigurable design using smaller memory blocks compared to the baseline design. Since the non-reconfigurable variant of the same module uses large monolithic memory blocks, the synthesis tool has to use multiple block RAMs on the FPGA to emulate these large memory blocks. With the reconfigurable implementation, the smaller memory blocks map much better to the actual BRAMs on the FPGA. The total number of blocks (20) is small enough to avoid the impact of added routing congestion on operating frequency. For the 40 memory block implementation of a cache memory module of same size, F_{max} is lower than the non-reconfigurable design. While this uses even smaller memory blocks, routing the signals to and from more memory blocks result in complex logic and longer combinational paths. The added pressure on the place and route tool due to the higher number of memory blocks results in longer wires. The result of the above is lower operating frequencies.

2.5 Configuring the Level of Reconfigurability

One distinct feature of this work is the ability to change the level of reconfigurability of the architecture. This architecture was designed in such a manner that the number of possible configurations depend only on the number of independent memory blocks in the architecture. Since this architecture reconfigures itself by reorganizing the logical organization of the memory blocks, more memory blocks means more possible configurations.

However, it is not possible to indefinitely increase the number of memory blocks due to resource constraints. There is a limit on how much area on a chip could be allocated to caches without running out of area to implement the rest of the design. It is possible to keep the total size constant and decrease the size of the individual memory blocks. At a certain point, this strategy also fails due to high routing congestion. Since more memory blocks need to have their inputs and outputs connected, the routing congestion increases when using a large number of small memory block compared to a few large memory blocks. Therefore, it is clear that a designer should strike a balance between resource usage and the level of reconfigurability. There may not be any general optimal setting for this since it is a decision specific to each and every design. A certain design might spend a huge portion of its chip area on a cache structure with high degree of reconfigurability in order to provide high performance across a range of different applications. On the other hand, another design might use a lower level of reconfigurability because the memory access patterns of all the applications expected to run on the processor are well known at design time.

Figure 2·13 depicts the variations in resource usage against the number of memory blocks while maintaining the total memory capacity constant. The total number of BRAM bits remain virtually the same. This is expected since we keep the total memory size constant. However, the number of block RAMs used grows with the number of memory blocks. When the memory is implemented as large monolithic memory blocks, the synthesis tool is able to use the block RAMs more efficiently. But when the cache memory is implemented as a large number of smaller memory blocks, the synthesis tool has to map the memory blocks to individual block RAMs on the FPGA. The resource utilization influenced the most by the changes in the number of memory blocks is the Adaptive Logic Modules (ALM) usage. As the number of memory block increases, routing resources required to connect all the memory blocks also increases. This is the major factor which cause the ALM usage to increase by almost 100% when the number of blocks is doubled.

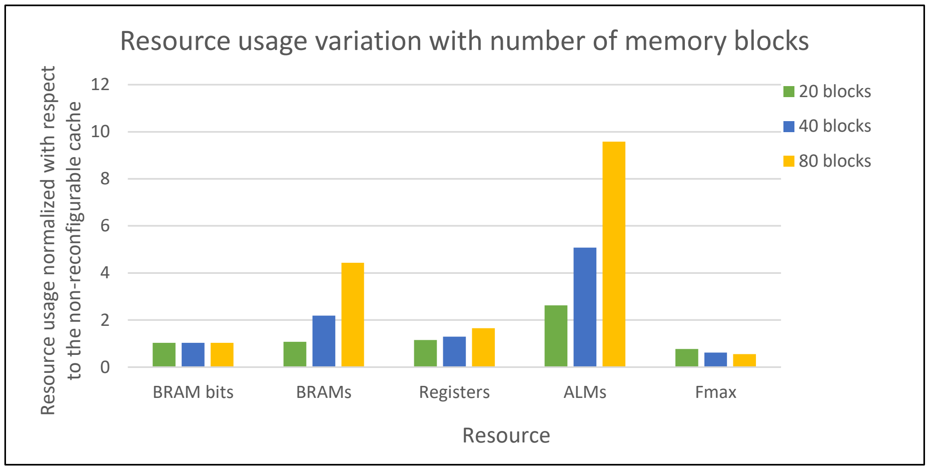


Figure 2·13: Normalized resource usage and F_{max} for the reconfigurable cache keeping the total memory size constant

2.6 Comparison with Previous Work

This section compares the level of reconfigurability of this work against some previous published work. Some of the numbers for previous work are estimates based on the level of detail in the relevant publications.

Way shutdown proposed in (Albonesi, 1999) provides the lowest level of reconfigurability. Let us consider a 4-way set associative cache with way shut down capability. At most it could have 4 different configurations. Let us assume that each line of the cache stores 4 words (16 bytes). With the tag storage also considered this is equivalent to 20 memory blocks in the proposed architecture. For the same amount of memory resources, the proposed architecture could provide 26 different configurations in total and 15 different configurations which utilizes at least half of the memory resources. If we discount the tag storage, we end up with 16 memory blocks. This gives us 20 total configurations and 13 configurations which utilize above 50% of the memory blocks available.

Way concatenation described in (Zhang et al., 2003) provides slightly better flexibility compared to way shutdown. Let's consider the same cache from previous paragraph. Let's also assume the cache has both way concatenation and way shutdown capability. This would give us a maximum of 6 different configurations. This is still far less than the number of configurations possible with the proposed architecture. The V-way cache (Qureshi et al., 2005) provides very high flexibility in terms of the associativity. It allows different cache sets to have different associativity. But it cannot dynamically change the two other cache parameters, cache sets and line width. While the proposed architecture can match the maximum associativity provided by the V-way cache, it does not support variable associativities across different cache sets. But unlike the V-way cache, the proposed architecture could dynamically change the line size and the number of cache sets.

The adaptive cache architecture proposed in (Tradowsky et al., 2016) provides very high flexibility. Since the architecture proposed in this work is based on a similar concept, the number of different configurations provided by the two architectures are similar. But due to the fact that the architecture proposed in this work has no dedicated tag storage and could use the same memory block to either store tags or data, this new architecture will provide more configurations for the same amount of memory resources.

Amoeba cache (Kumar et al., 2012) eliminates dedicated tag storage and considers the storage array to be uniform. It allows high flexibility in associativity and block sizes. It also supports different associativities among sets and different block sizes within the same set. Architecture proposed in this work does not support different line sizes within sets or different associativities for different sets. However, the proposed architecture can change the number of sets at run-time while Amoeba cache has a fixed number of sets.

Chapter 3

Self-tuning/Adaptive Caches

3.1 Background

Cache architectures described in section 2.1 provide different levels of reconfigurability, but no logic is implemented to determine the optimal cache configuration or when to perform the reconfiguration. Instead, the application developer must make cache configuration decisions. This imposes a requirement on the application developer to be familiar with the cache architecture in order to achieve best performance for the application. Since the majority of application developers may not be well versed in the micro-architectural details of a processor, they may not be able to utilize the reconfigurability of the caches. As we are spending a considerable amount of hardware resources in building a run-time reconfigurable cache, it is desirable to have it benefit most of the applications running on the system regardless of the application developer's understanding of the cache architecture.

Allowing the application developer to reconfigure the cache also requires the ISA to be extended with new cache reconfiguration instructions. This requires any existing binaries to be recompiled in order for those to enjoy the benefits of the new cache structure. Also the additional instructions in applications for reconfiguring the caches reduce the portability of application binaries.

Tuning the cache at run-time involves two tasks. First is determining when it is suitable to reconfigure the cache. This could be done by monitoring some selected cache performance metrics such as the hit rate. Second is to determine the optimal cache configuration which will improve the cache performance. Here, improving performance could mean multiple things. The obvious target is to improve the cache hit rate and reduce the time an application has to wait for memory accesses. If the primary caches have low hit rates, the memory requests have to be served by lower level caches or the main memory more often. Level 1 caches usually have latencies of a couple of cycles whereas the L2 caches have latencies of tens of cycles. Main memory has an even higher latency; the processor typically does a context switch when data has to be fetched from main memory. As discussed in chapter 2, different programs have different memory footprints and they have optimal cache configurations which result in highest hit rates. The task for the decision making logic is to determine the cache configuration which results in the highest hit rate for the current program.

Hit rate is not the only performance objective for a cache. Another obvious objective is to reduce the power consumption. Caches take up a large amount of chip area in modern processors, and account for a majority of the power consumption. With reconfigurable caches, it is possible to shut down unused resources and reduce power consumption. Whichever the performance objective may be, it is the responsibility of the decision making logic to select the next cache configuration based on current configuration, available resources and performance objectives.

3.2 Related Work

Authors of previous work have taken varying approaches to perform the two tasks discussed above in reconfiguring a cache at run-time. Zhang et al. in (Zhang et al., 2004) describe a heuristic to identify the optimal cache configuration. The heuristic-based self-tuning mechanism is implemented on top of the reconfigurable cache architecture described in (Zhang et al., 2003). The search space of possible configurations is constrained by the number of configurations supported by the reconfigurable cache. (3 settings each for cache size, line size, and associativity.) Cache parameters are ranked based on their effect on cache power dissipation. Cache is tuned by changing the parameters one by one and monitoring metrics such as cache misses. This work does not include methods to determine when to perform the reconfiguration. Reconfigurations takes place periodically according to a pre-determined set of cache configurations.

Authors of (Gordon-Ross and Vahid, 2007) take the cache and tuning heuristic from (Zhang et al., 2004) and describe a method inspired by feedback control systems, to determine the optimal tuning period. Tuning is focused on minimizing energy dissipation. In (Gordon-Ross et al., 2009), Gordon-Ross et al. propose a heuristic-based method to configure a two-level cache system. Reconfigurable cache described in (Zhang et al., 2004) is used as the level 1 cache. A cache similar to the one described in (Malik et al., 2000) is used as the unified level 2 cache. It is capable of specifying each way as unified (instruction and data), instruction only, data only or even shut down the way completely. Because of the level 2 cache, this work explores a significantly larger space of configurations compared to previous work. The authors demonstrate that trying to tune the two cache levels sequentially does not result in optimal configurations. Instead, they propose an interlaced exploration order for each parameter.

In these works, the possible solution space is limited by the capabilities of the reconfigurable cache architecture. The caches support a limited set of configurations depending on the base configuration. Self-tuning mechanisms depend on applying a set of possible configurations and monitoring the cache performance to determine the optimal configuration. The configurations are applied not based on any run-time metric but a pre-determined order. This means that sub-optimal configurations will also be attempted during the process of deciding the optimal configuration. This temporarily affects the energy efficiency and performance.

The authors of (Gordon-Ross et al., 2007) propose the “One-Shot Configurable-Cache Tuner” which is a hardware module to determine the optimal cache configuration in a non-intrusive fashion which does not negatively impact the performance. Unlike previous work, this method does not depend on applying different configurations and monitoring performance to determine the optimal setting. Instead it collects address traces and determines the optimal cache configuration for the current address trace. If the predicted configuration is different from the current configuration, a cache reconfiguration is performed. The hardware implementation described uses a stack structure to record the address trace and a multi-layered table structure to tally predicted hit counts under possible configurations. This means that when more possible configurations are considered, the resource usage goes up. The high resource usage makes it difficult to adopt this approach in a real design. Another drawback of this approach is due to the high latency in processing one address, this design cannot take the full address trace into consideration when making the decision on the optimal configuration. Instead it samples the address sequence. This reduces the accuracy of the prediction since the sampled range of addresses may not be representative of the complete memory footprint of the application. Another concern is whether the decision making logic takes the current performance of the cache into consideration or bases the reconfiguration decisions completely on the output of the tuning logic. In the latter case, cache will be reconfigured if the tuning logic predicts a configuration different from the current configuration regardless of the performance of the current configuration. However, it may not be necessary to perform a reconfiguration if the performance under the current configuration is acceptable.

Authors of (Dhodapkar and Smith, 2002) propose working set signatures to identify different phases of a program and reconfigure any hardware unit which supports multiple configurations according to the current phase. A phase in a program is the maximum interval in which the working set of the program remains more or less constant. Working set signatures are highly compressed representations of the working set of a program. In this paper, the authors propose an interesting alternative to dynamic resource tuning. Rather than tuning the configurations each time the phase changes, the working set signature could be used to load a set of configurations obtained through the reconfiguration algorithm the previous time the same working set was used. This will reduce the time to reach the optimal configuration for phases previously encountered and improve overall performance.

Authors of (Peng et al., 2007) propose a “phase-based self-tuning algorithm” to configure reconfigurable caches at run-time. Authors argue that since the working set remains unchanged, it is unnecessary to reconfigure the cache in the middle of a phase and unnecessary reconfigurations will hurt the performance. The authors propose detecting the phase changes in the program using the working set signature described above and reconfiguring the cache to fit the resource requirements of the new phase. The actual self-tuning algorithm implemented in (Peng et al., 2007) is for a cache with a lower level of reconfigurability and it is not useful for the proposed architecture.

3.3 Proposed Solution

Analyzing prior work shows that several factors contribute to the efficient use of a reconfigurable cache. The first challenge is determining when to reconfigure the cache. Periodically applying reconfigurations does not provide good results due to the fact that there is no check performed to determine whether reconfiguration is required. Indicators of varying complexity could be used to determine when to perform a reconfiguration. It could be as straightforward as the cache hit rate or as complex as monitoring addresses accessed and determining when a certain phase of the application executing has ended (Peng et al., 2007). An ideal solution will take into account both the performance metrics of the cache and the program behavior in terms of address traces and access timing.

In this work, program memory access patterns are not considered in determining whether or not to reconfigure the cache. This is due to high memory usage associated with such approaches. This work focuses only on current performance metrics of the cache. Several hardware counters implemented inside the cache monitor multiple metrics which could be used for determining both when to reconfigure and the best cache configuration to be selected. Current implementation considers the hit rate as the main indicator for reconfiguration. Two hardware counters in the cache track the number of requests and the number of cache hits to be used to calculate the hit rate (Not the exact hit rate but an estimation). These counters are reset after a reconfiguration to track the performance for the current configuration only. Another counter keeps track of the total cycles since the last reconfiguration. As discussed in section 2.4.1, hit rate and total cycles taken to service the memory requests considered together gives better indication of cache performance rather than hit rate alone. After a predetermined number of memory requests (N) are serviced by the cache, the tuning logic will read the counter values and perform a set of calculations to determine if the performance in terms of hit rate and total cycles is less than some predetermined level. If the performance is above or on par with the expectation, self-tuning logic will remain inactive for another N memory requests. After the N requests are serviced, the same check will be performed again with new counter values. The number of memory requests between tuning logic activations and the expected level of performance is determined at design time. A future improvement would be making it possible for an application developer to reconfigure these two parameters. Again, this would require certain ISA extensions. 'N' should be large enough so that the effects of the last reconfiguration are dominant and can be measured with high degree of certainty. Since the cache contents are flushed at reconfiguration, it is critical to allow enough accesses to fill up the cache before checking the hit rate. Since the cache is empty after a reconfiguration, the hit rates will be below average until the cache fills up.

If the self-monitoring logic determines that the cache performance is below the expected level and a reconfiguration is required, another piece of logic is activated to determine the next configuration to improve performance. Next configuration should be selected based on current configuration, types of misses encountered and cache utilization with the current configuration, and a predetermined importance of cache parameters in terms of the effect each has on the metric to optimize, such as the hit rate or the power usage. Section 3.4 describes different options in determining which cache parameter should be given priority in tuning.

3.3.1 Miss Classification Table

Miss classification table described by Collins and Tullsen in (Collins and Tullsen, 1999) is implemented to be used in determining the optimal cache configuration. This structure is capable of classifying cache misses as capacity misses and conflict misses. Compulsory misses are bundled into the capacity miss category. This uses a memory block with the same number of lines as the number of sets in the cache to store part of the tag bits from the line last evicted from each set of the cache. When a cache set encounters a miss, tag bits from the missed address is compared to the value stored in the table described above. If the tag bits are the same, the miss is classified as a conflict miss. This makes sense because if the cache had higher associativity, last evicted line would still be in the cache and the current access will be a hit. However, this classification is not perfect because only one evicted tag is stored. This makes sense for a direct mapped cache but for a cache with higher associativity, classification accuracy is not high. Authors provide the prediction accuracy of 87% for a direct mapped cache, but do not provide accuracy for caches with higher associativity. While it is possible to store multiple evicted tags for a cache set, the storage requirements grow fast with additional tags stored. Storing multiple evicted tags is the same as maintaining an additional tag storage apart from the usual tag and data storage and may not be an acceptable use of memory resources. However, it is possible to store only a part of the evicted tags. The tradeoff between storage resources and accuracy needs to be explored further.

3.3.2 Utilization Table

One important factor to be considered in optimizing the cache configuration is the cache utilization. For instance, if half of the sets in the cache have low utility, those memory blocks could be used to increase the associativity or block size of the cache and improve hit rate. Similarly, if the cache has larger than required associativity for the memory access pattern of the current application, some of the ways will have very low usage. Memory blocks for the ways with low usage can be reconfigured to increase the number of sets or the line width. To track the cache usage, two simple tables are implemented. First table tracks the set usage. It is implemented using a BRAM of depth equal to the maximum number of sets in the cache. Since the cache is reconfigurable, number of sets could change, but the table is allocated sufficient storage to handle the maximum number of sets in the cache. If the maximum associativity supported by the cache is 'W', each entry in the utilization table acts as a counter which counts up to 'W'. Therefore, the table entries should be $\Big(log_2(W)+1\Big)$ bits wide. During the reset sequence for the cache, each line of the table will be set to zero. Whenever a line is brought into the cache from memory and installed in a particular set, the table entry for that cache set is incremented by one. When a line is evicted from a cache set, invalidated by the coherence protocol, or invalidated during the reconfiguration sequence, the table entry for the corresponding set is decremented.

This enables keeping track of usage for each cache set. It is possible to tradeoff utilization tracking accuracy for storage resources by using one table entry for multiple cache sets.

Current implementation includes separate logic to cycle through the table and count how many sets are not utilized. Since the cache tuning logic operates parallel to the normal operations, this delay for going through the whole table is not a performance issue. It just impacts the time taken to make the reconfiguration decision. Since reconfiguration decision making is not done very frequently, this delay is acceptable. Adding logic to the utilization table to determine cache set utilization using less number of cycles than the number of sets in the cache is an interesting improvement which will speed up the reconfiguration decision making.

A similar table is implemented to track the way utilization of the cache. The way utilization table only has **(maximum number of ways in the cache)** entries. If the maximum number of sets is 'S', each entry in the table counts up to 'S'. Therefore, each entry is $\Big(log_2(S) + 1\Big)$ bits wide. Every time a new cache line is written to a way, the corresponding counter will be incremented. When an entry in a particular way is invalidated or evicted, the counter will decrement. The current values of table entries provide the number of lines utilized in each cache way.

Run-time reconfigurable caches could provide both high hit rates and lower power usage. This could be achieved by clock gating parts of the cache which are not being utilized. The memory arrays could be easily shut down since they are independent blocks separate from the control logic. Control logic is a bit more complex to be shut down. While utilization information is useful in determining how to optimize utility and therefore performance, it also is the source of information for the power efficiency oriented reconfigurations. Because the proposed architecture is based on a large number of small memory blocks, there is more flexibility in shutting down part of the memory blocks. For example, a cache with way shut down capability could perform optimizations at the granularity of cache ways. To shut down a whole way, it should have very low utility. But with the architecture proposed in this work, the optimizations could be done at a smaller granularity. Unlike the case with the way shut down, an optimization could be triggered when a certain way is only utilizing half of its sets, or three quarters of its sets. However, the amount of flexibility depends on the number of independent memory blocks used. Changes in the level of flexibility with the number of memory blocks is discussed in section 2.5.

While there are tables implemented to track set and way utilization, no logic is implemented to check how well the cache block size is utilized. Here utilization means how many words in a cache block inserted to the cache were actually used by the processor. If the application has high spatial locality, using wider lines acts as having a prefetcher. Most of the words in a cache line will be used in the execution. If the application has poor spatial locality, wider lines are detrimental to performance. Fetching wider lines from memory or a cache further away from the processor takes longer compared to smaller lines. If most of the words in the wide cache line are never referenced until the line is eventually evicted, the additional clock cycles spent fetching the wide line get wasted. A naive approach to track usage is to maintain status for each word on a cache line. But the high memory requirement makes this approach impractical. For the current implementation, utilization of the cache block size is not tracked.

Apart from the tables described above, several hardware counters are implemented to capture various metrics which will be used in both determining when to reconfigure the cache and determining the optimal cache configuration.

- Requests – Number of memory requests served
- Hits – Number of cache hits

- Coherence operations – Number of coherence operations performed
- Wait for bus – Number of cycles waiting to acquire the shared bus
- Wait for memory – Number of cycles waiting for memory to return data (Counting starts after coherence operations on the bus are completed and the cache is truly waiting for the memory to send data)
- Wait for shared writes – Cycles waiting to write to a shared line (Waiting for other caches to invalidate the line in question)
- Write-backs – Number of write backs
- Wait on misses – Total cycles waiting for required cache lines to be fetched
- Reconfiguration cycles – Number of cycles spent reconfiguring the cache (Reset after each reconfiguration)

Some of the metrics tracked by counters and tables discussed above have clear implications on the reconfiguration decisions. Other metrics are a little obscure and need to be examined further. Cache hits and misses are straightforward metrics indicating how well the cache is servicing the memory requests. When considered together with the total cycles counter, hit/miss rates could be derived and used directly in self-tuning logic. Write-backs is another straightforward metric. It indicates the number of dirty lines evicted from the cache and written back to L2 cache. Higher number of evictions indicate that there are conflict misses. This means that a cache reconfiguration may be required. The 'reconfiguration cycles' counter tracks the number of cycles spent on reconfiguration sequence the last time the cache reconfigured. This could be used as a reasonable estimate for the number of cycles which will be required for the next reconfiguration.

The number of cycles waiting for the memory to return data is affected by the cache line width. Higher number of cycles spent waiting could indicate that the cache lines are wider, and possibly too wide for the current application to achieve the highest hit rate. Wait for memory counter considered with total cycles counter can provide an idea about what portion of the total time is spent on waiting for the data to be fetched from lower levels in the cache hierarchy or main memory. The 'wait on misses' counter counts the total number of cycles spent waiting when there is a cache miss. This includes the number of cycles waiting for the coherence operations on the bus to finish, which is not captured by 'wait for bus' and 'wait for memory' counters.

The 'coherence operations' counter tracks the number of invalidations and write-backs performed by the snooper module. The 'wait for shared writes' counter tracks the number of cycles spent waiting to write to a shared cache line. Both these counters could indicate how many of the lines are shared with other caches. Large values in these counters could point to the fact that the cache lines are too wide and using smaller line sizes might decrease the coherence traffic and therefore improve cache performance. Since the snooper module operates independently of the main controller, coherence operations do not have a direct impact on normal cache operations. However, if the cache lines are too wide, there will be false sharing between caches and cache lines will be invalidated unnecessarily. This would increase the number of misses. False sharing could create a situation where a certain cache line keeps jumping back and forth between two caches, with both caches having to repeatedly invalidate and fetch the line again each time it is accessed. False sharing is the situation where two processors do not truly share any data, but they share a cache line. Since coherence is handled at cache line granularity, the cache line is transferred between the caches of the two processors, for every access to the cache line by the processors. This could be avoided by either tracking coherence information at lower granularity or using smaller cache lines. While wider cache lines generally improve the cache hit rates, a situation such as described above could significantly impact the cache performance due to false sharing. In such a situation, using a smaller line size may give better results.

3.4 Determining the Priority of Parameters

Even with all the performance and utilization metrics in hand, there could be instances where there is more than one cache parameter you could change. For instance, if half of the cache sets are underutilized, the cache could be reconfigured to better utilize the memory blocks underutilized. The question becomes whether to use the blocks to increase the cache associativity or the block size. Both are viable options, and run-time metrics may not be sufficient to prioritize one over the other. In such situations, rather than making a random choice, the parameter with highest impact on the objective function should be selected. Here the highest impact means that changing a particular parameter results in the largest variation of the metric such as hit rate or power usage which we wish to optimize. Therefore, to maximize the hit rate, parameter which has the highest impact on the hit rate should be tuned first. Similarly, if the objective function was power, another parameter may be given priority since it will have the highest impact on power usage.

3.4.1 Linear Regression

Determining the priority for each parameter could be done with different approaches. A naive approach would be to take the simulation results discussed in Chapter 2 and perform a linear regression considering hit rate (or some other objective function) as the dependent variable, and cache associativity, number of sets and block size, which are the tunable cache parameters as independent variables. This approach is very simple and straightforward, but has obvious issues. This assumes that the relationship between the cache parameters and hit rate could be represented by a linear model. It also does not directly consider a memory access model. Therefore, it ignores A priori knowledge we have on memory access patterns such as temporal and spatial locality. While the memory traces used for simulation might display these properties, it is not guaranteed that the memory traces used for simulations are a good enough representation of general workloads. By running a linear regression on simulation data for different cache configurations, following results were obtained.

Multiple Regression analysis was carried out using “Minitab 18” software with ‘hit rate’ as the response variable and (i) number of ways, (ii) offset bits, and (iii) index bits as the predictor variables. Multiple linear regression established that the predictor variables ways, index bits and offset bits statistically significantly predicted the hit rate. 61.84% of the variation in the response variable ‘hit rate’ can be explained by the regression model.

The regression equation is:

$$\begin{aligned} HitRate =\ & 39.44 + 3.151 \cdot W + 8.11 \cdot I + 7.637 \cdot B - 0.07342 \cdot W^2 - 0.330 \cdot I^2 \\ & - 0.3822 \cdot B^2 - 0.1723 \cdot W \cdot I - 0.1078 \cdot W \cdot B - 0.3834 \cdot I \cdot B \end{aligned}$$

Where W,I, and B represent ways, index bits, and offset bits respectively.

The predictor variables that contribute the most to the model variations of hit rate are; Ways > Offset bits > Index bits based on increased R^2 percentage. R^2 is a goodness of fit measure for linear regression models. It indicates the percentage of variation in the response that is explained by the model. Figure 3·1 shows the incremental impact of predictor variables.

3.4.2 A Statistical Model of the Cache Behavior

A better approach to decide the importance of each cache parameter is by developing an analytical model of cache behavior. Let us consider hit rate as the objective

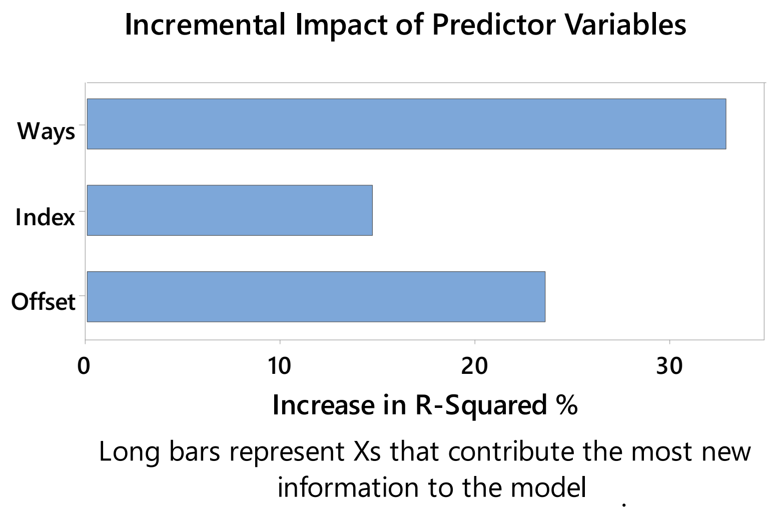


Figure 3·1: Incremental impact of predictor variables

function of choice. If we could model the hit rate using a model of memory access patterns and different parameters, it would be possible to derive which parameter has the highest effect on the hit rate. There are several previous publications on building analytical models of cache behavior (Agarwal et al., 1989) (Harper et al., 1999) (Suh et al., 2014).

We do not attempt to derive an accurate model here. Instead, the following section demonstrates how the cache parameters could be incorporated into a model for hit rate. A simple model can be derived in terms of probabilities. We have a reasonable understanding of the probability distribution for the next memory address given the current address, $P\left(Addr_{t+1} \mid Addr_t\right)$. Here t represents the discrete event of memory accesses rather than the absolute time. Due to temporal and spatial locality, the probability density function (PDF) should be one with more weight centered around the address currently accessed.

Given the above, let us assume that we can define the conditional probability of a particular address ‘A’ being accessed at time t such that $t < T$, given the address accessed at time T. Let us denote the probability of above by, $P_{acc}\big(A_t\big)$. Also let $P_{hit}\big(A_T\big)$ denote the probability of address 'A' accessed at time 'T' resulting in a cache hit and $P_{miss}\big(A_t\big)$ denote the probability of a cache miss for address 'A' accessed at time 'T'.

In order infer the importance of each cache parameter to maximize the hit rate, we should incorporate the cache parameters in the the analytical model. Let's start with a simplified model of a cache. In this model, each location in the cache stores only one byte (assuming a byte addressable memory) and the cache is as big as the entire memory space. Every address has a unique location in the cache which it will get mapped to. Therefore, in this model, the only requirement for the current access to be a hit is that the same address has been accessed at some time t prior to current time T. For this model the probability of a cache hit could be expressed as;

$$\begin{aligned}
P_{hit}\big(A_T\big) &= 1 - P_{miss}\big(A_T\big) \\
&= 1 - P_{acc}\big(B_{T-1} \mid (B \neq A)\big) \cdot P_{acc}\big(B_{T-2} \mid (B \neq A)\big) \cdot \\
&\quad P_{acc}\big(B_{T-3} \mid (B \neq A)\big) \cdots\cdots \\
&= 1 - \prod_{t=0}^{T-1} P_{acc}\big(B_t \mid (B \neq A)\big)
\end{aligned}$$

However, this is not a realistic model of a cache. Since the caches are much smaller compared to the memory space, multiple addresses map to the same index in the cache. Caches use some of its lower order bits to index into the cache. Assume the cache has 's' sets. $(A \bmod s)$ gives the set which address 'A' will get mapped to, where 'mod' is the modulo operator. When we take into account the fact that multiple addresses could map to the same location in the cache, out formula for the probability of hit should be updated to incorporate this new constraint. For address accessed at time T to be a hit, the necessary conditions are,

1. The same address has been accessed at some time $t < T$.

2. No address which maps to the same location in the cache was accessed since.

These conditions could be represented as,

$$\begin{aligned} P_{hit}\big(A_T\big) =& P_{acc}\big(A_{T-1}\big) + P_{acc}\big(A_{T-2}\big) \cdot P_{acc}\big(B_{T-1} \mid (B \mod s) \neq (A \mod s)\big) + \\ & P_{acc}\big(A_{T-3}\big) \cdot P_{acc}\big(B_{T-2} \mid (B \mod s) \neq (A \mod s)\big) \cdot \\ & P_{acc}\big(B_{T-1} \mid (B \mod s) \neq (A \mod s)\big) \\ & + ... \\ P_{hit}\big(A_T\big) =& P_{acc}\big(A_{T-1}\big) + \sum_{t=0}^{T-2} \left(P_{acc}\big(A_t\big) \cdot \prod_{s=t+1}^{T-1} P_{Acc}\big(B_s \mid (B \mod s) \neq (A \mod s)\big) \right) \end{aligned}$$

Now we have incorporated one cache parameter to the equation. To incorporate the other parameters, we should keep adding more constraints and make the model more realistic.

Real caches do not store one word per line. A cache with one word lines will only exploit temporal locality. Fetching multiple words which are stored in adjacent memory locations to the address currently accessed enables a cache to exploit spatial locality. Therefore, each line in the cache stores multiple words. The number of words in a line is referred to as the cache block size or line size. The least significant bits of the memory address are referred to as offset bits and are used to select between the multiple word in the line. If the number of offset bits is denoted by 'b' and >> represents the right shift operation, the index calculation now looks like,

$Index = (Address >> b) \mod s$

The updated expression is,

$$
\begin{aligned}
P_{hit}\big(A_T\big) = & P_{Acc}\Big(C_{T-1} \mid \big((C >> b) \bmod s\big) = \big((A >> b) \bmod s\big)\Big) + \\
& \sum_{t=0}^{T-2} \Bigg(P_{Acc}\Big(C_t \mid \big((C >> b) \bmod s\big) = \big((A >> b) \bmod s\big)\Big) \cdot \\
& \prod_{s=t+1}^{T-1} P_{Acc}\Big(B_s \mid \big((B >> b) \bmod s\big) \neq \big((A >> b) \bmod s\big)\Big)\Bigg)
\end{aligned}
$$

Now both the number of sets and line size is incorporated to the expression. In order to incorporate the cache associativity, we need to consider the replacement policy. If we assume least recently used (LRU) replacement, the necessary conditions for address A accessed at time T to be a cache hit are;

1. Address 'A' or some other address in the same cache line has been accessed at some time $t < T$.

2. The number of addresses which mapped to the same cache set since time t is less than 'w', which is the associativity of the cache.

At this point, we have incorporated all three parameters to the cache model. Such a model could be used to determine the importance of each parameter in optimizing a certain objective function.

3.4.3 Learning-based Methods

A third approach to deriving the relative importance of the three tunable cache parameters in maximizing hit rate is based on learning methods. Previous work has shown that it is possible to use machine learning methods for optimization problems in computer systems. Authors of (Hashemi et al., 2018) use LSTMs (Long Short Term Memory) which is a Recurrent Neural Network architecture to improve memory prefetching performance. They show that the learning based approach consistently outperforms traditional prefetchers in SPEC2006 benchmarks. In (Sakr et al., 1997), three trainable prediction techniques are used to predict memory access patterns in order to optimize the configurations of dynamically reconfigurable interconnection networks. The authors test the performance of of i) a Markov predictor, ii) a linear predictor, and iii) a Time Delay Neural Network (TDNN) to predict the memory access patterns. A similar approach could be taken in predicting which cache parameter to tune first in order to get the maximum impact on hit rate.

Chapter 4

Cache Side-channel Attacks and Run-time Reconfiguration as a Defense Mechanism

4.1 Cache Side-channel Attacks

Cache side-channel attacks can be used for many purposes such as stealing cryptographic keys (Osvik et al., 2006), spying on keyboards/mouse, breaking kernel address space layout randomization (Hund et al., 2013) (Evtyushkin et al., 2016), violating browser sandboxing (Oren et al., 2015), among others. Cache side-channel attacks exploit intrinsic characteristics of cache systems such as cache lines being mapped to sets, inclusive property, hit/miss time difference and cache coherence. The timing difference between a cache hit and a miss is the most commonly exploited characteristic. Caches are used to hide the memory latency by keeping a small subset of recently used memory contents on the processor die itself and avoiding accessing slower main memory on every memory access. When there is a cache miss, the requested data should be brought to the cache from the main memory or disk. This takes significantly longer to provide the requested data to the processor compared to a cache hit where the requested data is available in the cache and can be sent to the processor immediately. This timing difference could reveal certain information regarding the contents of the cache, and if the cache is shared by more than one process, information regarding the memory access patterns of the processes sharing the cache.

There are three main techniques used in cache side-channel attacks;

1. Evict: Access memory until a given address is no longer cached
2. Prime: Place known addresses in the cache
3. Flush: Remove a given address from the cache using cache flush instructions provided in certain ISAs

These techniques are combined with timing to devise different cache side-channel attacks. Some of the major attack models are;

4.1.1 Prime + Probe

'Prime+probe' attacks consist of three steps;

1. Prime a cache set to contain known addresses
2. Wait for victim activity
3. Time accessing the known addresses from step 1

If the accesses in step 3 takes longer than a certain threshold, the attacker infers that the victim accessed an address which maps to the same cache set which was primed in step 1. An attacker using 'prime+probe' technique can correctly infer the cache sets accessed by the victim. Therefore, the attack is said to have cache set accuracy.

4.1.2 Evict + Time

'Evict+time' attacks work in a different fashion. It needs to be able to execute a certain function such as a crypto function. The steps involved are;

1. Execute the function and time it
2. Evict a cache set
3. Execute the function again and time it

If step 3 was slower than step 1, the attacker can deduce that the function uses an address which maps to the cache set which was evicted. Similar to 'prime+probe', this attack also provides cache set accuracy.

4.1.3 Flush + Reload

This attack makes use of the cache flush functions provided in the ISA (clflush in x86). The attacker should have access to a shared library used by the victim. The attack depends on the fact that the kernel same page merging will will map this library to the same pages accessed by both the processes. This shared mapping allows the attacker to directly manipulate the addresses which may be accessed by the victim. The attack is carried out using the steps listed below.

1. Flush shared address from the cache
2. Wait for victim
3. Access the address flushed in step 1 and time the access

If step 3 was fast (cache hit), attacker can infer that the victim has accessed an address which is in the same cache line as the flushed address. 'Flush+reload' has the highest accuracy since it provides cache line accuracy.

There are several other attack models such as 'flush+flush', and 'prime+abort'. Although the mechanics may differ, all cache side-channel attacks are based on the same principle. That is, the attacker and victim processes sharing cache resources, and the attacker's ability to manipulate those shared resources. The cache acts as a covert channel the attacker can use to monitor the victim process' activity. This is done by monitoring the effects of the victim's activity on the shared cache resources. The general flow of a cache side-channel attack is (i) set the cache contents to a known state, (ii) allow the victim program to run, and (iii) check the state of the cache and try to deduce information regarding the victim's access pattern from the current state of the cache contents. Most of the early cache side-channel attacks depended on the co-location of the attacker and the victim processes. That is the two processes are executing on the same core. However, this is not common with high core counts in modern processors and service providers actively avoiding process co-location in cloud environments. To counter these trends, the latest attacks focus on the Last Level Cache (LLC) to mount cross-core and cross-VM attacks. Authors of (Liu et al., 2015) & (Yarom and Falkner, 2014) describe such attacks on the last level cache.

4.2 Previously Proposed Defenses

Previous works have proposed several defenses against cache side-channel attacks. There are two classes of defenses proposed. First is the set of software based defenses. These include solutions ranging from application level solutions such as rewriting applications to reduce information leakage to system level solutions such as monitoring processor performance counters and using machine learning to predict potential attacks and terminate suspicious programs (Chiappetta et al., 2016).

Cryptographic applications are the main target of cache side-channel attacks. The vulnerability of these applications arises from memory access patterns that are correlated with the secret information which the attacker attempts to retrieve. For instance, Advanced Encryption Standard AES (Daemen and Rijmen, 1999) has been shown to be vulnerable to cache side-channel attacks (Osvik et al., 2006) (Bernstein, 2005). While AES can theoretically be implemented completely with arithmetic and logical operations in the processor, real implementations use lookup tables to store the outputs for every possible input for certain steps of the algorithm. Accesses to these tables depend on the secret key and the plaintext to be encrypted. Since all memory accesses go through the cache system, this correlation between secret data and memory accesses create a cache side-channel. The first type of defenses is rewriting the applications in a way which minimizes the information leakage. The drawback of this approach is that the defenses are very specific to the applications. A generic solution which provides a defense for a range of applications is desirable compared to an application specific solution.

Cache side-channel attacks depend on the timing difference between cache hits and misses. To measure these differences, an attacker needs access to high resolution timers accessible through the operating system. One proposed defense is to restrict access to high resolution timers. However, recent works have shown that these attacks can be carried out even without access to high resolution timers (Mcilroy et al., 2019). Another defense targeting 'flush+reload' and 'flush+flush' attacks is restricting access to cache flush instructions. An obvious drawback of both these approaches is the fact that benign applications also will be denied access to these features and it will impact their performance and correct execution. In the case of restricting cache flush, this is a change to the ISA and requires recompiling and in certain cases even rewriting all applications. This may not be an option with certain legacy applications.

The core challenge faced by software based defenses if that the side-channel attacks are not based on any particular weakness of the software or the algorithms, but certain features of the hardware on which the software is executed. Therefore, there is no straight forward defense against side-channel attacks on the software level. While software based defenses could successfully defend against very specific attacks, those cannot provide any general guarantees about a whole class of attacks or several different threat models. This is due to the specific nature of the software based solutions, and the lack of control the software has over the underlying hardware. In contrast if the defenses against side-channels were implemented in hardware, it can prevent attacks on more than one application and will prevent the need to recompile/rewrite every application with security concerns in mind.

The second class of side-channel defenses are hardware based solutions. This section gives priority to defenses related to cache architecture. Wang and Lee in (Wang and Lee, 2007) describe the Partition-Locked cache (PLcache) and Random Permutation cache (RPcache). The PLcache allows a process to lock cache lines to prevent other processes from evicting it. This essentially creates a private partition inside the cache. PLcache requires the ISA to be extended with a new set of load/store instructions with a lock/unlock sub-operation. The RPcache attempts to randomize the memory to cache mapping by adding another level of indirection in addressing cache sets. A 'Permutation Table' (PT) stores an alternate mapping of the index bits to cache sets. The process which has to be protected will use the PT while the other processes will not. This randomizes the set mapping and minimizes the attacker's ability to infer useful information about which cache sets accessed by the victim process.

Authors of (Wang and Lee, 2008) propose an architecture similar to RPcache with a dynamic mapping from memory to cache. They propose a direct mapped implementation with a longer index than what corresponds to the number of cache lines available. The 'remapping table' and the replacement policy map a larger logical cache to the physical cache while retaining the most recently used lines in the cache. The authors of (Domnitser et al., 2012) propose Non-Monopolizable caches, which restrict the maximum number of ways in a set a given process can use. This reduces the ability of the attacker to evict the victim's data from a set and the amount of useful data the attacker could infer. This approach does not require extensions to the ISA or support from the operating system. However, it requires additional hardware to keep track of threads using each way in a set. This also could have a performance impact on benign processes by restricting the effective amount of cache memory which can be used.

Dai and Adegbija in (Dai and Adegbija, 2017) explore the use of reconfigurable caches as a defense mechanism against cache side-channel attacks. This work utilizes a highly configurable cache similar to the architecture described in (Zhang et al., 2003) which allows total cache size, line width, and associativity to be dynamically configured. In this work the authors statically determine a set of cache configurations, targeting a particular application or an application domain, which would satisfy two conditions; (i) Selected configurations provide sufficient variability among themselves rendering the information gathered by the attacker less effective; and (ii) Average energy consumption of the set of configurations is below a predefined energy threshold. A hardware cache tuner applies the set of reconfigurations corresponding to the current application, periodically at run-time. Reconfiguration period is statically determined based on a known attack model. Since a predetermined set of configurations are applied in a predetermined order, it may still be possible for an attacker to design an attack which overcomes the limitations introduced by this method. Having a predetermined reconfiguration window also reduces the effectiveness of this method since that allows the attacker to modify a given attack to fit the window of time available. The reconfiguration takes place regardless any other run-time factors. Therefore, even if there is no attack taking place, the cache will reconfigure itself periodically. This will result in a performance degradation as some of the cache configurations may not be optimal for the current application.

Previous works include two different approaches to hardware based defenses against side-channel attacks. First is isolation which limits the ability of one process to manipulate the data belonging to another process. Cache line locking, and preventing one process form monopolizing the cache fall under this category. The second approach is obscuring certain information regarding the cache structure and introducing noise to the information collected by the attacker. This approach is based on the fact that cache side-channel attacks require intimate knowledge of the exact cache structure. Without an understanding of the cache parameters such as cache size, associativity, block size, replacement policy, cache line mapping function, etc. it is extremely difficult to mount a successful cache side-channel attack. ‘RPcache’ described in (Wang and Lee, 2007), and the defense proposed in (Dai and Adegbija, 2017) fall under this category. Solution proposed in this thesis also follows the second approach in providing defense against side-channel attacks.

All the architectures described above provide different levels of protection against side-channel attacks at the expense of performance, area, and power. They also pose different levels of difficulty to be adopted in designs due to requiring ISA extensions and/or operating system support, performance degradation, etc.

4.2.1 Performance Impact

Different defenses against cache side-channel attacks have varying levels of impact on the performance. Continuous monitoring in software to detect side-channel attacks has high impact on performance since the monitoring takes CPU cycles from the user applications running on the system. Rewriting applications to limit information leakage typically results in a performance overhead, either because of dummy calculations performed to obscure information or due to avoiding performance improving alternatives such as using pre-computed lookup tables instead of computations on the processor. Methods based on detecting suspicious applications usually terminate the suspect application. If the detection was not accurate and a benign application was incorrectly flagged, such defense mechanisms affect not only the performance but the execution correctness of innocent programs.

Partition locked cache (Wang and Lee, 2007), and ‘Non-Monopolizable’ caches (Domnitser et al., 2012) restrict the maximum amount of cache resource which can be utilized by a given application. This could clearly result in a performance degradation for certain applications. Cache architecture described in (Wang and Lee, 2008) and the ‘RPcache’ (Wang and Lee, 2007) dynamically changes the address mapping to cache sets. The additional functionality will have an area overhead and an impact on operating frequency. Reconfigurable cache based approach in (Dai and Adegbija, 2017) applies cache reconfigurations periodically without paying any attention to currently executing application or whether there is an attack taking place. Since certain cache configurations may not be optimal for certain applications and because reconfiguration also takes up time, there will be a performance impact from this approach as well.

4.3 Utilizing Adaptive Caches as a Defense Mechanism

Given that the success of cache side-channel attacks depend on the understanding of the actual structure of the cache, if the cache structure is dynamic, mounting an attack becomes more challenging. When the dynamic range increases, difficulty of mounting an attack also increases. This is the rationale behind using run-time reconfigurable caches as a defense against side-channels. The expectation is that due to the dynamic nature of the cache structure, data collected by the attacker is highly noisy that accurate information cannot be retrieved.

Cache side-channel attacks are performed by initializing the cache contents to a known state and monitoring the effect of victim’s activity through the changes in the cache contents. Attack models such as ‘flush + reload’ and ‘flush + flush’ utilize cache flush instructions provided in certain ISAs to manipulate the cache. Other attack models make multiple accesses to a set of addresses which maps to the same cache set. A set of addresses which map to the same set of a cache is called an ‘eviction set’ (Vila et al., 2018). The success of side-channel attacks depends on the attacker's ability to find eviction sets efficiently. To find an eviction set the attacker should be aware of the number of sets in the cache, the size of cache blocks and the associativity of the cache. An attacker should also know the replacement policy and how the cache sets are indexed. Lower level caches are usually physically indexed and physically tagged (PIPT) while primary caches are virtually indexed to avoid address translation latency. Primary caches can be virtually or physically tagged. If the attacker is to mount an attack on a secondary cache, an understanding of virtual to physical address translation is also required. If an attacker fails to identify an eviction set, the success of the attack is unlikely. Motivation behind this work is dynamically changing some of the cache parameters the attacker needs to identify eviction sets and mount an attack.

Consider a 'prime and probe' attack on a W-way set associative cache using LRU replacement policy. In the prime step, the attacker program should prime the cache by accessing addresses known to map to certain cache sets. To prime a cache set with 'W' ways, the attacker should make at least W accesses to the same cache set in order to evict current content of the set and fill it with known values. To do this the attacker should first find an eviction set of W addresses. Assume that the cache was a run-time configurable one and it contains logic to detect the high number of evictions for the same set. The cache control logic interprets the evictions as a result of insufficient associativity and reconfigures to have associativity of $2 \cdot W$. At this point, the eviction set of W words is useless since it only evicts half the words in the set. As the cache does not have unlimited resources, doubling the associativity would mean halving the number of sets or the block size. Changing the number of sets or cache block size changes the mapping from memory to the cache. With the modified line mapping, some of the addresses in the eviction set may not map to the same set anymore. This makes the original eviction set even more ineffective, and therefore rendering the attack unsuccessful.

If the reconfigurable cache has a static reconfiguration period and if the attacker is aware of it, then the attacker could modify the attack to fit within the time window between two reconfigurations. Although the time constraint limits the amount of information retrieved, the information collected will be accurate. The attacker can still mount a successful attack, despite taking more time and attempts. Reconfiguration process takes a certain number of cycles which could have been used for useful computations. Performance impact of reconfiguration increases with the frequency of reconfigurations. Therefore, the frequency of reconfiguration should remain low in order to reduce the performance impact, which makes the time window for the attacker to mount an attack longer. Therefore, an ideal solution should not perform reconfigurations with the same period, but either reconfigure when a potential attack is detected or at least at random time intervals.

If the reconfigurable cache supports a very low number of possible configurations, an attacker may still be able to work around the reconfiguration and complete the attack. An attacker can detect a cache reconfiguration by the excess delay in the cache servicing requests from the processor. Then the attacker can perform a simple preprocessing step to determine the current configuration of the cache. Once it is found, the attacker performs the attack until the next reconfiguration occurs. However, determining the cache configuration becomes difficult when the cache has a higher number of possible configurations. The attacker's attempts to determine the current cache configuration will take longer due to the large number of potential cache configurations. This will reduce the amount of information gathered by the attacker between two reconfigurations, because more time is spent on determining the cache configuration. Therefore, a reconfigurable cache which supports a higher number of configurations will be more successful in thwarting a cache side-channel attack.

4.4 Proposed Solution

This work proposes reconfigurable caches as a defense against cache side-channel attacks. Since the success of cache side-channel attacks depend on the knowledge of the cache configuration, run-time changes in the cache configuration will reduce the accuracy of information gathered through a cache side-channel attack. The reconfigurable cache should be complemented by an attack detection mechanism which will detect potential cache side-channel attacks. A detection mechanism is important because periodic reconfigurations; (i) are easier to be countered when mounting an attack, and (ii) increase the negative impact on performance. The attack detection mechanism could be software based or hardware based.

Reconfigurable caches provide performance improvements over non-reconfigurable caches of the same size. The performance gains from the optimized cache configurations are expected to offset some of the performance degradations due to security related cache reconfigurations and attack detection overhead (in case of software based detection). Some of the modifications to the cache architecture proposed in previous works, create a performance limitation in the cache architecture itself. For instance, partitioning the cache among different processes limit the amount of cache memory accessible by one process. Since the memory access patterns of programs are not uniform, some are bound to experience performance degradations due to cache partitioning. Complex cache mapping schemes such as 'RPcache' (Wang and Lee, 2007), reduces the operating frequency and possibly the latency of the cache. Run-time reconfigurable cache implementation proposed in this work has the same latency as the non-reconfigurable cache. However, the maximum operating frequency is slightly lower than the operating frequency of the non-reconfigurable cache. The architecture proposed in this work can optimize the cache structure to fit the exact requirements of the current program when it does not suspect a side-channel attack. Therefore, on average the proposed architecture should provide better performance compared to previous work while providing strong defense as shown in section 4.5.

When a potential threat is detected, instead of terminating the suspicious application, a cache reconfiguration will take place. This will render the information collected by the attacker mostly useless as the side-channel attacks depend on the knowledge of the cache organization. Since the suspicious application is not terminated, even if a benign application was incorrectly flagged by the detection mechanism, the application will only experience a performance degradation. Therefore, the program completion will not be affected even if the attack detection mechanism is not highly accurate.

However, if the timing is critical to the correct execution of a program, the proposed method could cause such programs to fail. For example, a real-time process might fail if a cache reconfiguration takes place in the middle of execution. We make the assumption that the timing critical programs are well written and do not perform operations which will be flagged by the detection mechanism.

4.4.1 Flush + Reload Attacks

The defense proposed in this work is directly targeting cache side-channel attacks which does not use the cache flush instructions provided by the ISA. Those are the attacks such as 'flush+reload' and 'flush+flush'. These attacks are based on identifying which addresses were accessed by the victim program rather than attempting to identify which cache lines were accessed by the victim. Attacks such as 'prime+probe' depend on identifying which cache lines were accessed by the victim. Since cache reconfigurations could change cache set mappings, it becomes increasingly difficult to mount an attack which depends on identifying cache sets accessed by the victim. On the other hand, a 'flush+reload' attack could still succeed even with the cache dynamically reconfiguring because regardless of the exact configuration of the cache, the cache flush instructions allow an attacker to flush an address from the cache hierarchy.

However, the dynamic reconfiguration could still have an effect on the success of cache flush based attacks. There is no certainty that a ‘flush+reload’ or a ‘flush+flush’ attack will fail due to this method. But, it could still greatly reduce the accuracy of information collected by the attacker. Let’s first recall how a ‘flush+reload’ attack is mounted. The attacker first flushes certain addresses from the cache. These addresses are usually from a shared library accessed by both the victim and the attacker. The attacker expects to infer some sensitive data regarding the victim process by analyzing the accesses to the shared library. For instance, the shared resource could be an AES lookup table in a cryptographic library. The attacker then yields and allows the victim process to run. Once the attacker program regains the processor, it reloads the addresses which were flushed previously. Attacker also measures the latency of the memory requests, being served by the caches. If the latency is below a certain threshold, the attacker assumes that the requested data was accessed by the victim process. More precisely, the victim has accessed some address which maps to the same cache line as the address flushed by the attacker. Attacker uses this information to infer sensitive information regarding the victim.

Assume that a cache reconfiguration takes place in the middle of a ‘flush+reload’ attack. Since all the data in the cache is written back to the cache in the level below, the cache is empty when it is ready to service memory requests again. Depending on the timing of the reconfiguration in relation to the execution timing of the attacker and victim processes, the reconfiguration could write back the address which the attacker is monitoring, after it was accessed by the victim. Now the attacker might deduce that the address was not accessed by the victim even though it was, because when the attacker attempted to reload the address, it was not returned immediately by the level one cache. However, Reconfiguration sequence does not flush addresses in the cache back to the main memory. Dirty cache lines are only written back to the lower level caches. A flush instruction used in 'flush+reload' attacks usually flushes the address from the whole cache hierarchy. Therefore, the attacker may still be able to detect the difference in timing between fetching data from L2 or L3 cache and main memory. In conclusion, the ability of the proposed method to thwart 'flush+ reload' attacks depends on a host of factors related to the caches such as the sizes, and contents at the time of reconfiguration. It also depends on the implementation details of the side-channel attack, such as the latency threshold considered when reloading and timing, etc. Therefore, the proposed method does not provide a reliable defense against 'flush+ reload' attacks, but impacts the accuracy of such attacks as a side-effect.

4.4.2 Selecting the Next Configuration

If the cache is reconfigured to improve the hit rate, there is a clear way to determine what the next configuration should be. It is the configuration which could potentially improve the hit rate the most. When the cache detects a potential side-channel attack, a new configuration has to be selected. The selection criteria could be different from the one for performance optimization.

We should first analyze how a side-channel attack works in order to define a set of rules which will enable us to determine the next cache configuration that will deter the side-channel attack. We only focus on 'prime+probe' type attacks which do not utilize cache flush instructions. Cache flush instructions can manipulate the cache contents regardless of the current configuration of the cache. In the first phase of the attack, the attacker forms eviction sets which are groups of addresses which map to the same cache set. Addresses in eviction sets are accessed to evict the current content of the cache and set the cache contents to a known state. Let us analyze the effect of changing each cache parameter on the attacker's assumptions about the cache organization.

Cache associativity has the most obvious effect on the attacker's actions. The size of the eviction set is based on the associativity of the cache. For instance, if the cache is four-way set associative, the attacker needs to find at least four addresses that map to the same cache set in order to put the cache set in a fully known state. If the associativity was eight, eviction set size should also be eight. If the cache detects a side-channel attack and increases the number of ways, the eviction sets formed by the attacker becomes ineffective. Since there are more ways than the number of addresses in the eviction set, the cache set will not be in a fully known state. Even if the victim accesses an address which maps to a set primed by the attacker, it will not replace any of the addresses in the attacker's eviction set. This is because the set was not fully primed by the attacker. When the attacker probes the cache set, all the addresses from the eviction set will still be there and will result in short access times. The attacker will conclude that the victim has not accessed that particular cache set even though it is not the case.

Intuitively, reducing the cache associativity should have the opposite effect. It should make the eviction sets even more effective because the associativity is smaller than the size of the eviction set. However, if the attacker is unaware of the cache reconfiguration, they keep accessing every address in the eviction set. The later accesses end up evicting the previously accessed addresses of the eviction set from the cache due to insufficient associativity. When the attacker probes the cache, part of the addresses the attacker expects to be in the cache results in cache misses. Attacker's accesses keep evicting the addresses of the eviction set and this causes more misses. Because of the cache misses, the attacker assumes that the cache set was accessed by the victim. The loss of accuracy in the data collected by the attacker depends on several factors including; i) the exact implementation of the attack, and

ii) whether the cache reconfiguration takes place during the prime phase or the probe phase of the attack and how much of the respective phase was completed before the reconfiguration. Both increasing and decreasing the cache associativity reduce the success of a side-channel attack. Section 4.5 provides experimental results on the effectiveness of changing cache associativity as a defense against a side-channel attack.

Effect of the cache line size is less obvious compared to the associativity. Changes in line size modifies the cache set mapping. An address mapped to a particular index under a certain line size can map to another index under a different line size. The caches in commercial designs use hash functions to map addresses to cache sets rather than simply using the index bits from the address. Usually these mapping functions are not documented, and the attacker has to experimentally find the eviction sets. Therefore, some of the addresses in the eviction set could map to different words in the cache line. Changes in line size could change the sets which some of the addresses in the eviction set map to. This also reduces the effectiveness of the eviction set. Because the effect is not guaranteed as with associativity, cache line size changes get a lower priority in the list of parameters to change when an attack is detected. While decreasing line size gives the effect described above, increasing line size does not. Therefore, if it is not possible to increase the cache associativity, controller will decrease the line size when an attack is detected.

Similar to the cache line size, the effect of changing the number of cache sets is also not guaranteed. By changing the number of sets, the number index bits taken from the address to index into the cache changes. This could result in some of the addresses in the eviction set being mapped to different cache sets. However, depending on the set index and the addresses in the eviction set, it may have no effect at all. Based on the reasoning above, we can derive a list of configuration changes which would affect the success of side-channel attacks.

1. Increase/decrease the associativity

2. Decrease the line size

3. Increase/decrease the number of sets

The reasoning behind selecting above configurations was based on the assumption that each of the cache parameters can be changed independent of the other two. However, this is not a realistic assumption due to the fact that the cache is of finite size. Unless most of the memory blocks in the cache are currently powered down due to underutilization, it is not possible to increase the cache associativity, set count or line width without decreasing one or both of the remaining parameters. Decreasing the associativity, line width or set count can be done without changing the other parameters. However, it does not make sense to power down the unused memory blocks and reduce the size of the cache. The performance penalty introduced by the smaller associativity, set count or line width, can be reduced by using the memory blocks to increase one or both of the remaining parameters. For instance, if the associativity is reduced, the memory blocks in the unused ways can be reconfigured to increase the number of sets or the line width. Therefore, more than one cache parameter will change during a reconfiguration for a typical implementation where we expect maximum utilization of resources. Changing multiple parameters increases the noise introduced to the data gathered by the attacker, and improves the defense provided by the reconfigurable cache.

4.5 Results

The success of a side-channel attack is determined by the amount of information retrieved through the attack. Therefore, the success of a defense mechanism could be measured through the reduction in the amount of data retrieved when the defense mechanism is in place compared to when there is no defense mechanism. To evaluate the effectiveness of the proposed defense mechanism, the same principle is applied. First a simplified model of a cache side-channel was implemented and simulated on a RISC-V core adopted from BRISC-V platform (Bandara et al., 2019). The simulation is based on two programs. The first is the victim program which accesses a set of random addresses. The second is the attacker program which attempts to correctly infer the cache lines accessed by the victim program. Attacker success is measured by the percentage of cache lines correctly identified as the ones accessed by the victim. Each scenario was simulated 100 times with different random address sequences accessed by the victim. The attacker takes a simple 'prime+probe' approach in this attack.

After establishing a baseline for the attacker success, the level 1 caches in the BRISC-V system were replaced by the run-time reconfigurable caches and the impact of run-time cache reconfigurations on attacker success was observed. The same test was performed for attacks taking place at level 1 caches and level 2 caches. For the level 1 case where the two processes are co-located, two different scenarios were simulated. First is a hardware multi-threaded core where the two processes run on separate hardware threads. The second scenario is where the core only supports a single thread and the processes are scheduled by a scheduler. This was used to create a worst case scenario where the attacker gets full control over the cache right before and right after the victim process executes. Therefore, the attacker program gets to prime the cache with no interference from other processes before the victim starts executing. After the victim has completed execution, the attacker gets to probe the cache without any interference from other processes. Since attacker and victim are the only processes scheduled to the core, there is no cache pollution by other processes.

Among the different scenarios simulated, the one where attacker and victim processes are manually scheduled one after the other, results in the highest success for the attacker. Attacker success is expected because we simulate an artificial best case scenario for the attacker. In reality, it is not guaranteed that a scheduler will never preempt these processes in the middle of execution. And it is also unrealistic to assume that the only processes running are the attacker and the victim. However, since the experiments are carried out to analyze the impact of run-time reconfigurations on the accuracy of the information collected by the attacker, best case scenario for the attacker is used as the baseline for the analysis.

Figures 4·1, 4·2 and 4·3 show the baseline attacker success for 4-way set associative, 2-way set associative and direct mapped caches when there are no run-time cache reconfigurations. Other cache parameters were kept constant with values of 256 sets and 16-byte cache lines. We can see that the percentage of cache sets correctly predicted by the attacker does not change significantly depending on the associativity of the cache. Figures 4·4 and 4·5 show the percentage of sets correctly predicted for a couple of 4-way set associative caches with 128 and 64 sets respectively. Cache line width is 16-bytes. While the smaller caches provide slightly better results for the attacker, it is not a significant improvement. Therefore, we can conclude that the number of sets in the cache also does not impact the accuracy of the attack significantly. From these results, we can see that as long as the attacker is aware of the correct values of all the cache parameters, an attack could be carried out with more or less the same accuracy regardless of the exact values of parameters.

Figures 4·6 and 4·7 presents the accuracy of attacks when the attacker does not know the actual associativity of the cache. In both scenarios where the attacker assumes a higher associativity and a lower associativity respectively, the accuracy of the information gathered drops drastically. Accuracy of attacks when the attacker

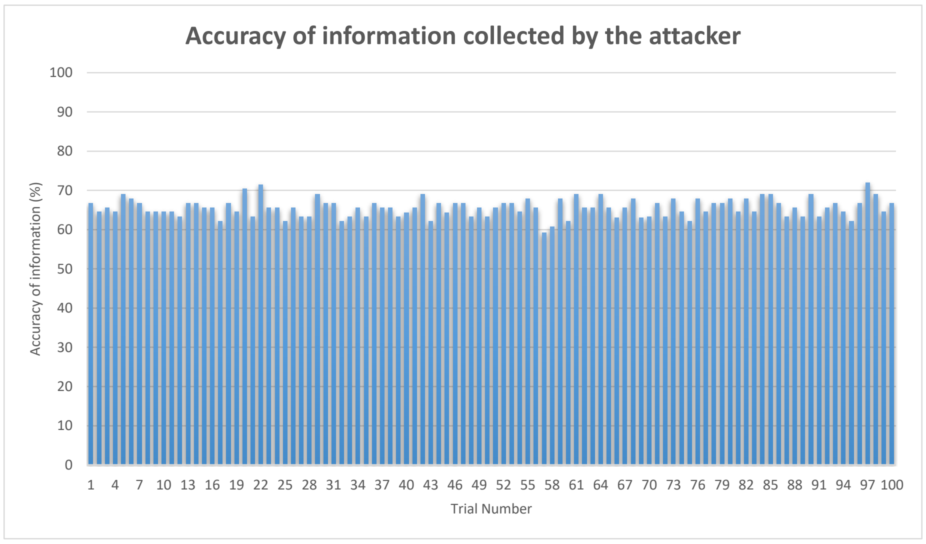


Figure 4·1: Percentage of cache sets correctly inferred by the attacker (4-way set associative cache)

assumes an incorrect number of cache sets is shown in figures 4·8 and 4·9. Similar to associativity, assuming larger or smaller set count than the actual count reduces the accuracy significantly. From these results, we can conclude that assuming a wrong value for even one cache parameter results in extremely low accuracy of the data collected through cache side-channel attacks.

Next, we analyze the effect of run-time cache reconfigurations on the accuracy of

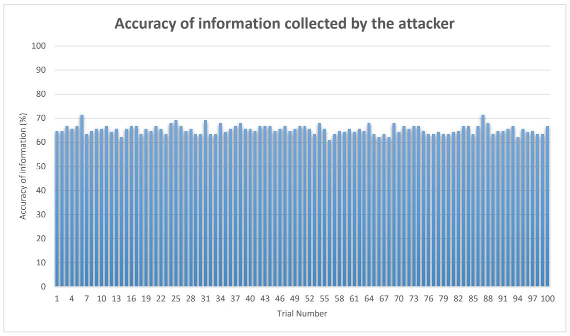


Figure 4·2: Percentage of cache sets correctly inferred (2-way associative cache)

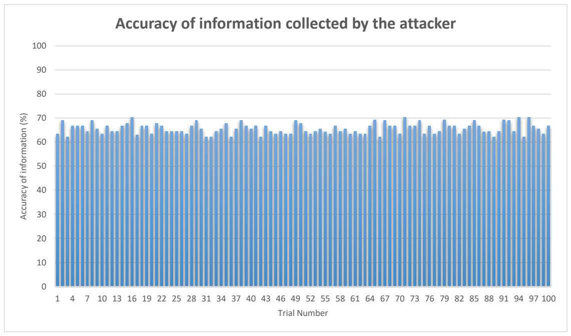


Figure 4·3: Percentage of sets correctly inferred (Direct mapped cache)

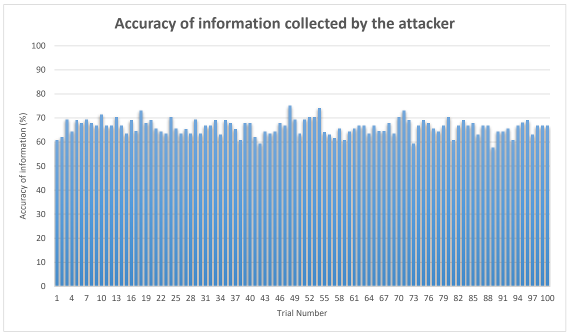


Figure 4·4: Percentage of sets accurately inferred (128 cache sets)

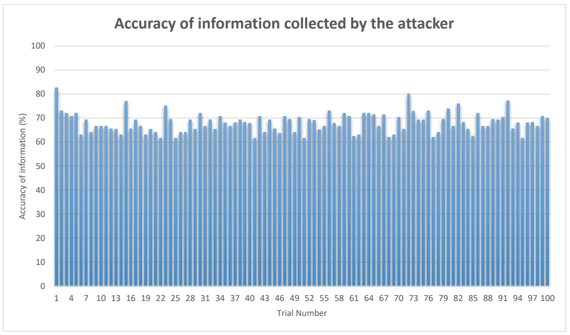


Figure 4·5: Percentage of sets accurately inferred (64 cache sets)

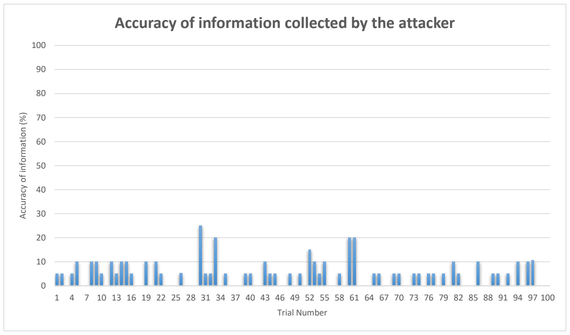


Figure 4·6: Percentage of sets accurately inferred (Underestimate associativity)

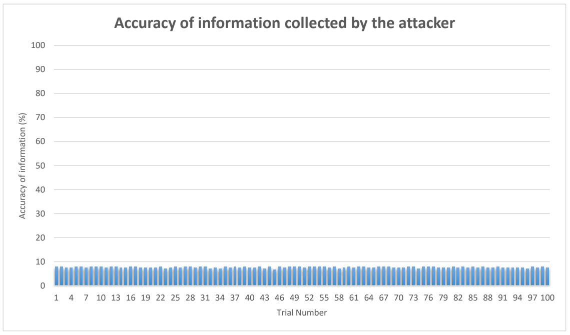


Figure 4·7: Percentage of sets accurately inferred (Overestimate associativity)

the data collected by the attacker. We assume that the cache detects the attack and reconfigures itself. For these simulations, it is assumed that there is an oracle which predicts an attack taking place.

Since the changes in cache associativity has the highest impact on the accuracy of the attack, we first test run-time changes in cache associativity. Figures 4·10, 4·11, and 4·12 depict the percentage of cache sets correctly inferred by the attacker when the associativity of the cache is doubled via a reconfiguration, during the probe phase of the attack. The reconfiguration takes place respectively after 25%, 50%, and 75% of the prime phase is completed. We can observe that if the cache reconfiguration takes place during the prime phase, the attack accuracy is dropped below 10% on

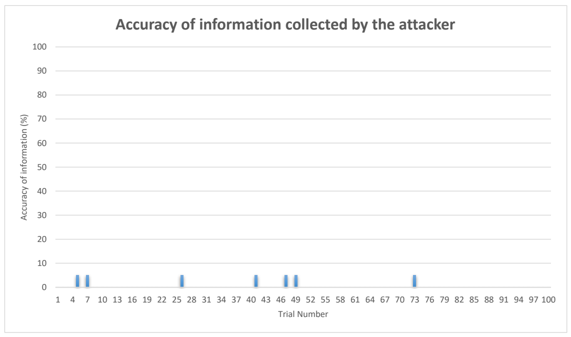


Figure 4·8: Percentage of sets accurately inferred (Underestimate set count)

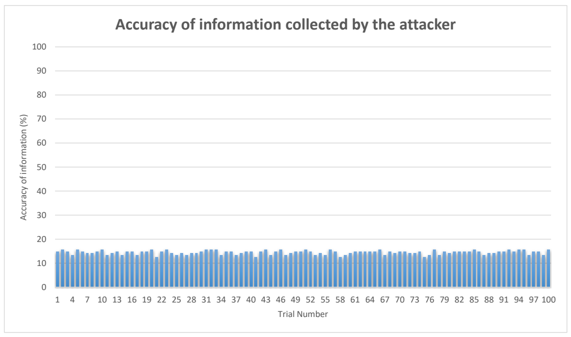


Figure 4·9: Percentage of sets accurately inferred (Overestimate set count)

average. Cache reconfiguration has two effects on the attack. First, it writes back all dirty lines and clears the cache. This means the cache sets already primed by the attacker are no longer primed. After the reconfiguration, the attacker cannot properly prime the cache sets without knowing the new configuration. Therefore, the accuracy of the attack is very low.

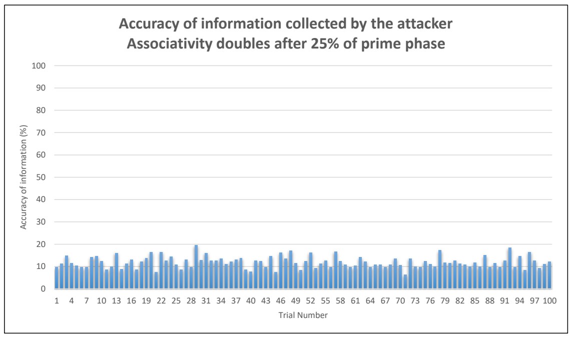


Figure 4·10: Associativity doubles after 25% of the sets are primed

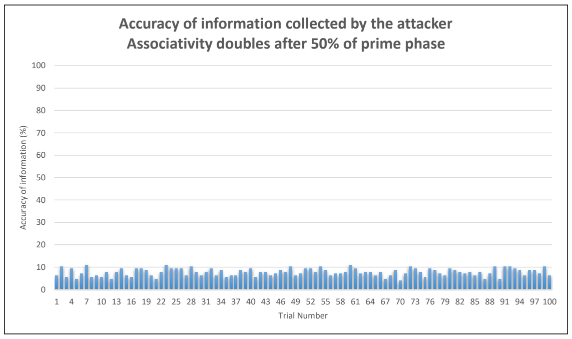


Figure 4·11: Associativity doubles after 50% of the sets are primed

Figures 4·13, 4·14 and 4·15 show the accuracy of the attack when the number of ways is doubled during the probe phase of the attack. We can clearly see that the accuracy increases with the detection delay. This is obvious because with more time before the cache reconfiguration, the attacker is able to probe more sets and gather

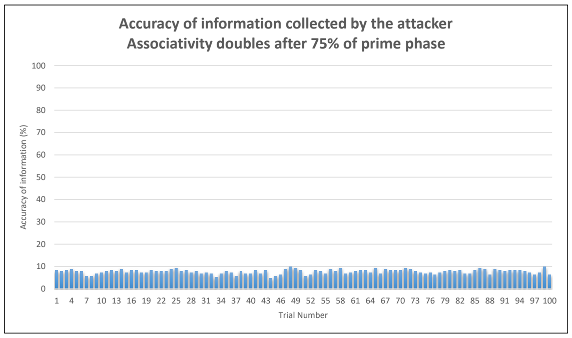


Figure 4·12: Associativity doubles after 75% of the sets are primed

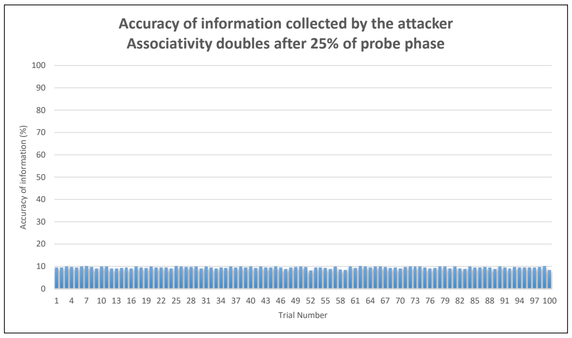


Figure 4·13: Associativity doubles after 25% of the sets are probed

more information. From this observation, we can conclude that the detection speed is important to the success of the defense.

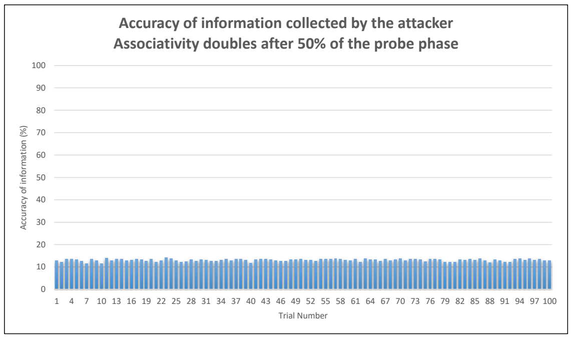


Figure 4·14: Associativity doubles after 50% of the sets are probed

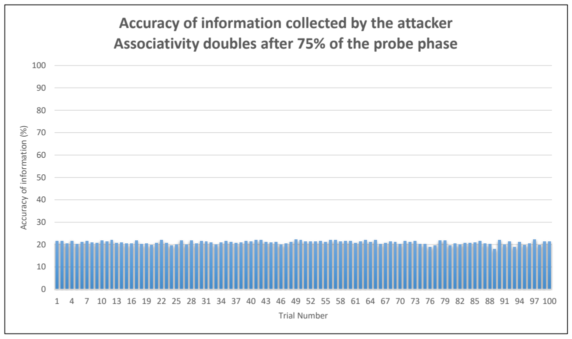


Figure 4·15: Associativity doubles after 75% of the sets are probed

Next, we test the effect of changing the other two cache parameters on the accuracy of the side-channel attack. Changes during the prime phase resulted in constantly low accuracy for the attacker. But reconfigurations during the probe phase showed some interesting variations. Therefore, only the results for reconfigurations during the probe phase are analyzed here. Figures 4·16 and 4·17 show the accuracy achieved by the attacker when the cache line width is respectively doubled and halved after 50% of the cache sets are probed. It is worth noting that in the scenario depicted in figure 4·17, the line width was halved without increasing the other parameters. The new configuration does not use almost half of the memory blocks in the cache. In the case of increasing the line width, number of sets was halved because the cache has a fixed number of memory blocks. Changes in two parameters versus one parameter could play a part in the lower attack accuracy in figure 4·16 compared to figure 4·17.

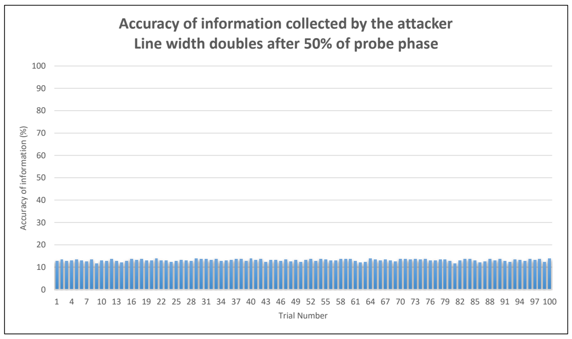


Figure 4·16: Line width doubles after 50% of the sets are probed

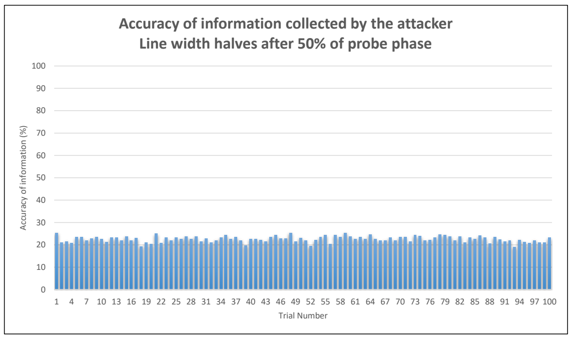


Figure 4·17: Line width halves after 50% of the sets are probed

Finally, we analyze the effect of the number of cache sets. Similar to previous two experiments, the number of cache sets was doubled and halved after 50% of the probe phase was completed. The results are shown in figures 4·18 and 4·19. Here, both the reconfigurations involved changes to set count and associativity in order to maintain a high memory block utilization. Decreasing the number of sets seems to impact the attack accuracy more than increasing the number of sets.

4.5.1 Analysis

The scenarios with no run-time reconfigurations and other programs in the system yields the highest success for the attacker. This was expected because there is no noise polluting the data collected by the attacker. The simplicity of the attack scenario and the cache hierarchy used in these simulations also result in high success rates for the attacker. There is no virtual memory or complex cache line mapping schemes

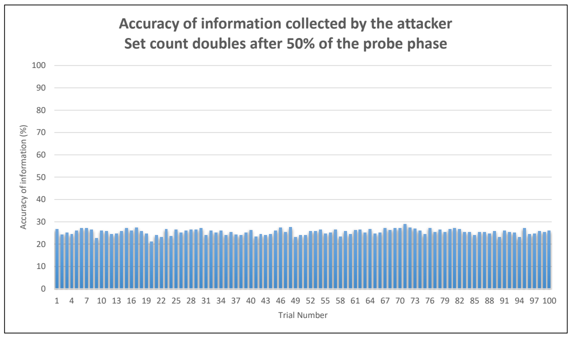


Figure 4·18: Set count doubles after 50% of the sets are probed

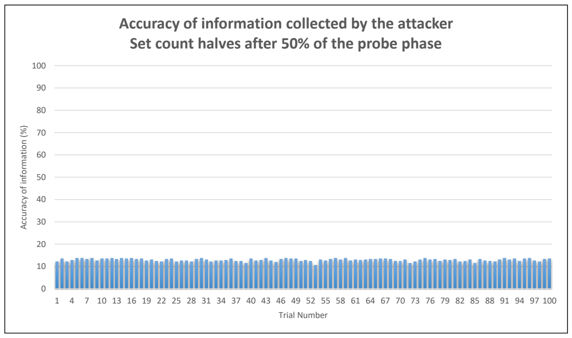


Figure 4·19: Set count halves after 50% of the sets are probed

the attacker has to work around. All the caches use physical addresses to map cache lines. The processor is running bare-metal code.

Adding other benign processes to the system reduces the success of the attacker. This is due to caches now being shared by all the other processes. The attacker cannot distinguish the accesses from the victim and the other processes not targeted by the attack.

Run-time reconfigurations drastically reduce the attacker success. Even a change in one cache parameter renders the information gathered by the attacker more or less unusable. This is due to the fact that the attack is designed assuming a certain cache configuration. Once the cache configuration changes, some of the assumptions made in designing the attack becomes invalid. Therefore, the information collected by the attacker becomes unreliable. A single reconfiguration was shown to be sufficient to reduce the accuracy of the attack. However, these experiments were conducted using a simple implementation of a cache side-channel attack which does not attempt to identify a cache reconfiguration and the new cache configuration after a reconfiguration. If a sophisticated implementation of the attack was used, multiple reconfigurations may be necessary.

Chapter 5

Detecting Cache Side-channel Attacks

5.1 Background

5.1.1 Importance of Detection

In chapter 4, we have shown that run-time cache reconfigurations significantly reduce the amount of secret data inferred through an attack. As a result, the probability of attack success is reduced. However, if the attacker is aware of the reconfiguration, they can perform a pre-processing step to determine the current cache configuration before launching the actual attack. Performing periodic reconfigurations can introduce performance degradations due to sub-optimal configurations and time spent on unnecessary cache reconfigurations. If the attacker could infer the reconfiguration period, they can break the attack into smaller steps which fit within the reconfiguration period. After each reconfiguration, the pre-processing step will identify the current cache configuration. Attack code will be executed until the next reconfiguration. This method could still work successfully while the only drawback being the longer time taken to complete the attack. It is also not feasible to make the reconfiguration time period very small due to performance considerations. Therefore, it is reasonable to assume that if periodic reconfiguration is performed, the reconfiguration period will be long enough for the attacker to perform both pre-processing and the actual attack.

Due to these considerations, it is important to have the ability to detect potential attacks and carry out the reconfiguration only if a potential attack is detected. It will eliminate the need for unnecessary periodic cache reconfigurations. Moreover, the system can react quickly when an attack is detected.

5.1.2 Prior Work

A major challenge in defending against cache side-channel attacks is real-time detection. If a side-channel attack cannot be detected when it is taking place, there is no way to prevent the attack. Previous works have explored different approaches for cache side-channel attack detection.

CloudRadar (Zhang et al., 2016) is a system to detect cache side channel attacks in a cloud environment, where multiple virtual machines (VM) from different users can be co-located in the same processor package. Although cloud providers prevent VM co-location on the same core, the same is not true for different cores in the same package. While the private caches of the cores are not shared with other VMs, the last level cache (LLC) will inevitably be shared among different VMs executing on different cores. This gives the attacker an opportunity to mount a cache side-channel attack across VMs. CloudRadar utilizes processor performance counters to detect potential side-channel attacks while one of the VMs is executing a cryptographic operation. First, the system detects whether any of the VMs are running any cryptographic applications using signature based detection methods. If a VM is running a cryptographic application, it monitors the cache activity of other VMs co-located on the same package. It uses anomaly detection techniques to detect suspicious cache activity from other VMs.

The authors of (Chiappetta et al., 2016) propose three schemes for real-time detection of cache side-channel attacks which use the 'flush+reload' technique. All three methods leverage processor hardware performance counters to detect potential attacks. The first method, which is the only non-machine-learning solution, is to find a correlation between performance counter data for the victim and attacker processes. The intuition behind this is that when a side-channel attack is taking place, the victim and the attacker will have largely similar memory access patterns. Total accesses to the LLC is identified as a good indicator by the authors. The second method is to train a neural network to detect potential attacks. The third approach is based on semi-supervised anomaly detection. Both methods 2 and 3 need profiled data of malicious programs to train the machine learning model. They both use the following events tracked by hardware performance counters as features; i) Total instructions, ii) total CPU cycles, iii) L2 cache hits, iv) L3 cache misses, and v) L3 cache accesses. While the simplest method proposed in this work, which is the correlation based method, shows good detection accuracy, the authors argue that machine learning based detection schemes are required because of the presence of false positives where benign applications are falsely flagged as malicious. If these suspicious applications are terminated by the system, false positives will affect the execution correctness of innocent applications. The authors claim that the machine learning base methods will increase the confidence in detection.

Intel Cache Monitoring Technology (CMT) and hardware performance counters are used in (Bazm et al., 2018) to detect cross-VM side channel attacks on the LLC. This work uses Gaussian anomaly detection to identify potential side-channel attacks. LLC references, LLC misses, iTLB-cache-misses and iTLB-r-accesses are the events used as features for the model. In (Yu et al., 2013) the authors use k-means clustering to distinguish the behavior of side-channel attacks from legitimate applications. This also targets side-channel attacks taking place in a cloud environment across VMs. The authors have identified three features as good indicators for side-channel attacks; i) long cache miss times due to the attacker continuously accessing memory addresses or flushing in order to evict addresses from the cache, ii) Low CPU usage because most of the time is spent waiting for memory/caches, and iii) virtual memory utilization changing constantly.

By analyzing prior approaches we can observe few trends;

- All proposed methods perform software-based attack detection.
- Almost all of the methods monitor processor performance counters and other information provided by Intel cache monitoring technology.
- Most of the methods are based on machine learning techniques.

These observations make the approaches taken in prior works non-ideal for this research. The target for this work is to have a detection mechanism confined to the reconfigurable cache itself. Software based detection is not preferred because of the added complexities of extensions to enable software to directly control the cache behavior including run-time reconfiguration. Also a hardware based approach will eliminate the performance impact with low cost monitoring. Neural network based approaches were too complex given the time frame for this work and the expertise of the author. Therefore, this work attempts to device a simple hardware based detection mechanism within the reconfigurable cache itself.

5.2 A Hardware-based Detection Mechanism

The starting point for the detection mechanism was the hardware counters implemented within the cache. The details of the counters were presented in chapter 3. Apart from the counters, an additional table-like structure was implemented to track suspicious behaviors which could indicate a side-channel attack.

5.2.1 Eviction Table

One simple indicator to detect side channel attacks which are not based on the cache flush instructions is the unusually high eviction count. In the prime step of a ‘prime+probe’ attack, the attacker uses eviction sets, which are sets of addresses which map to the same cache set, to evict all the addresses currently occupying the cache set. This should result in up to ‘W’ evictions in a W-way set associative cache. The number of evictions will always be close to ‘W’, unless the cache set is mostly empty, which would indicate that the cache is designed with larger than required associativity for the workloads running on it. This fact can be used to detect side-channel attack-like behaviors.

A table-like structure is implemented to track the last ‘n’ cache sets which had a line evicted from it. Value of ‘n’ is determined at design time. The table stores the set index and the number of evictions. It also tracks the set which encountered an eviction least recently among the sets recorded in the table. If a line was evicted from a set which is already in the table, the counter corresponding to that set is incremented and the set is brought to the top of the stack. When a line is evicted from a set which is not already in the table, it is added to the table and the set which encountered an eviction least recently is removed from the table. If the counter of a table entry reaches a predetermined value ‘w’, it indicates that the cache set encountered a large number of evictions. This is an abnormal behavior and can be considered as an indicator of a potential side-channel attack. To increase the confidence in detection, a large number of evictions to a single set alone is not considered as an indicator of an attack, but identical behavior for a predefined ‘k’ number of sets is. The ‘w’ and ‘k’ values should be fine-tuned using profiled memory traces of side-channel attacks.

The upper bound on the size of the table is based on the latency between encountering a miss and cache starting normal operations after fetching the requested line and inserting it to the cache. This is because when a line is evicted from set ‘x’, table entries should be accessed one by one until an entry for ‘x’ is found or the last entry of the table is reached. In the worst case scenario where the cache set in question has not previously encountered an eviction in the period tracked by the table, the whole table has to be accessed. These accesses should fit in the duration for fetching and inserting the missing cache line. While it is possible to implement the whole table with LUTs (without using block RAMs), and allowing all table entries to be read in parallel, the resource requirements of such an implementation will not allow us to have a big table. This in return eliminates the need to make the table accesses parallel. Therefore, the structure is implemented using BRAM and the size of the table is bounded by the latency for fetching a missing cache line.

5.2.2 Impact of Detection Accuracy on Performance and Correct Program Execution

When a side-channel attack is detected, the operating system can terminate the process associated with it. However, this requires good detection accuracy since false positives would mean that benign programs are flagged as malicious and terminated. A distinct feature of the defense proposed by this work is that it does not involve termination of processes. In fact, it has no notion of processes. The defense is entirely contained within the cache subsystem and the detection is based on how memory accesses are observed by the cache subsystem. Once suspicious memory access patterns are detected, the cache will change its internal organization. If there is an attack taking place, the cache reconfiguration will make some of the assumptions made by the attacker invalid and hence the attack ineffective. If the detection is a false positive, the proposed defense will not prevent the incorrectly flagged application from completing execution. Instead there will be a performance degradation due to; (i) the time taken to reconfigure the cache, (ii) the cache being empty when the execution starts after a cache reconfiguration, and (iii) potentially selecting a non-optimal cache configuration. Even if the detection accuracy was low and a certain benign program is flagged as malicious, there is a high probability that the cache still requires a reconfiguration to match the access pattern of that particular program. A high number of evictions for a few sets does not indicate the cache resources being utilized optimally. Therefore, such a program can still benefit from a cache reconfiguration. The fact that program completion is guaranteed reduces the pressure on the detection mechanism to have high accuracy in terms of low false positives.

Although this is described as a hardware based solution, it doesn't have to be. This method can be complemented by software based defenses. Particularly, the hardware based detection mechanism can be used to alleviate the performance overhead of continuous monitoring in software. If the hardware based monitoring detects any suspicious memory access patterns, it can trigger the software based monitoring via an interrupt. Then the software based method can be used to verify with high confidence whether a malicious program is being executed. On the other hand, the cache reconfiguration can complement software based detection schemes. When the detection confidence is not high, the software based detection scheme may choose to reconfigure the cache instead of terminating the suspicious program. Although this will incur a performance penalty, it will not impact the correct execution of an innocent program.

Run-time cache reconfigurations can be used to supplement some of the real-time detection methods proposed in previous works. Terminating the processes with suspicious memory access patterns is the action proposed by most work. However, reconfigurable caches provide another viable response to a potential side-channel attack. If the detection mechanism detects potential attacks with varying degrees of confidence, it is important to have a defense which does not include potentially terminating a benign application. In a system with run-time reconfigurable caches, a low confidence detection of a potential attack could trigger a cache reconfiguration. High confidence detections can still terminate the suspicious process.

Chapter 6

Conclusions

6.1 Summary of the Thesis

In this thesis we explored the viability of adaptive caches as a defense against cache side channel attacks. Out exploration began with developing a run-time reconfigurable cache architecture. Chapter 2 of this thesis provided a detailed account of the reconfigurable cache architecture and the design methodology. We were able to design a novel cache architecture which provides very high flexibility in terms of different configurations. A key feature of the cache architecture proposed in this work is the ability to configure the level of run-time reconfigurability at design time. The same architecture can be implemented with a different number of distinct configurations. However, there is a tradeoff between the number of configurations and the amount of resources used. We also presented case studies which show the potential performance improvements provided by a run-time reconfigurable cache.

Chapter 3 of this thesis focused on adaptive caches. We implemented several hardware counters to track different cache events and performance metrics. Two table-like structures were also implemented to monitor the cache resource utilization. Then we analyzed the relative importance of different cache parameters in optimizing a certain metric. Using cache hit rate as the target metric, different methods of establishing the priority for cache parameters were described. Using one of the methods, we established that the number of ways in a cache has the highest impact on the cache hit rate. Cache line width and the number of cache sets have decreasing impact on hit rate, in that order. Next, the performance impact of the proposed cache tuning logic was tested using several benchmark programs.

In Chapter 4, Our exploration moved on to the core research topic of this thesis, using adaptive caches as a defense mechanism against cache side-channel attacks. We first described the intuition behind using adaptive caches as a defense. We then analyzed the pros and cons of doing so. Finally, the impact of run-time reconfiguration on the ability of an attacker to successfully complete a cache side-channel attack was analyzed experimentally. The experimental results proved that a run-time change of even a single cache parameter is able to drastically reduce the accuracy of data collected by an attacker. We established cache associativity as the parameter with highest impact on the attack success. Both increasing and decreasing associativity were shown to be viable options to thwart a side-channel attack. Changes in line width and set count also had varying levels of impact on the success of an attack. Another fact established through the experimental results was, the importance of attack detection speed. Against a 'prime+probe' attack, cache reconfigurations taking place after 75%, 50%, and 25% of the probe phase resulted in decreasing accuracy in the data collected by the attacker. Reconfigurations during the prime phase resulted in even lower success for the attacker.

The final chapter of this thesis was focused on run-time detection of potential cache side-channel attacks. A simple hardware based method of detecting potential side-channel attacks was proposed. The proposed solution showed good detection capability against simple attack programs used in experiments. However, it was shown that a targeted implementation of a side-channel attack can easily circumvent the proposed solution. An important feature of the proposed defense mechanism is its ability to tolerate low detection accuracy. Since the proposed defense does not involve terminating suspicious processes, a false positive in detection results in a performance degradation rather than terminating a benign process.

6.2 Key Takeaways

The major takeaways from this research are as follows;

- Reconfigurable caches are an effective defense mechanism against cache side-channel attacks.
- Reconfigurable caches provide strong defense with negligible performance overhead compared to previously proposed methods.
- To provide strong defense, reconfigurable caches should be accompanied by an attack detection mechanism.
- The speed of detecting a potential attack is paramount to the success of the defense.
- Using run-time cache reconfigurations to thwart cache side-channel attacks reduces the pressure on attack detection mechanism because the defense does not involve terminating suspicious processes.
- Changing even a single cache parameter at run-time reduces the accuracy of a cache side-channel attack significantly.
- However, high flexibility provided by the cache in terms of the number of configurations is still important to limit the ability of the attacker to perform a preprocessing step after a cache reconfiguration and identify the new cache configuration.
- Building the cache structure with small independent memory blocks and changing the logical organization of the blocks can provide very high flexibility in terms of distinct configurations.

- An architecture with above features will allow multiple implementations with different tradeoffs between the level of reconfigurability and resource usage.

6.3 Future Work

There are several future research directions originating from this work. This section presents the research areas under two categories.

6.3.1 Reconfigurable Cache Architecture

The first class of future research based on the reconfigurable cache is improving the decision making logic. Currently the reconfiguration decision is made based on a handful of hardware counters, cache set and way utilization, and a predetermined priority of cache parameters derived from linear regression. There are two obvious improvements. First is developing a mechanism to track how many words/bytes in a cache line were actually referenced before the line is eventually evicted from the cache. Second improvement is to derive the priority of the cache parameters from a more accurate model of a cache which takes in to account the memory access patters, replacement policy, secondary caches, etc. While we have briefly explored developing a statistical model of a cache in section 3.4.2, it was meant as a demonstration of how to incorporate different cache parameters to a model of some cache performance metric, rather than an exercise to develop an accurate model. Another future research area will be exploring machine learning based methods implemented in hardware, as cache tuning logic. A potential interesting design choice would be implementing the machine learning algorithm directly on hardware versus implementing a small specialized processor inside the cache to perform the learning tasks.

The second class of future research is exploring alternate uses of the proposed cache architecture. One such alternate use is developing a caching structure which allows cache resource sharing among processor cores. Typically, all the cores on a multi/many core processor is allocated the same resources. However, there can be situations where one core among several on a processor runs a resource hungry application which fully utilizes the private caches on that particular core, while the other cores run applications which puts less stress on their local caches. If the processor has a unified caching structure which allows transferring cache resources among the cores, the unused cache memory form the cores executing less intensive programs can be transferred to the core which runs the resource hungry application.

Another area the proposed run-time reconfigurable cache architecture can be utilized is thread migration among processor cores. When a thread migration takes place, the data of the migrated thread is not available in the caches of the new core. The process associated with the thread will experience a performance degradation due to the cache misses. A reconfigurable cache structure based on the architecture proposed in this work will be able to re-allocate the memory blocks used by different cores in a way that the memory blocks are also migrated among cores parallel to the thread migration.

6.3.2 Cache Side-Channel Attack Defense

Future research directions under cache side-channel attack defense revolves around integrating the reconfigurable caches with other defense mechanisms. The main focus is on integrating the reconfiguration with methods of run-time detection of cache side-channel attacks. If a detection method has different levels of confidence for its detections, cache reconfiguration can be used as the defense when a potential attack is detected with low confidence. This will ensure the program completion in case of a false positive. In case of a high confidence detection, the suspicious process can be terminated. Terminating processes is the defense mechanism used in some of the prior works. Those could be improved by incorporating cache reconfiguration and expanding the number of actions available when a potential attack is detected.

References

Agarwal, A., Hennessy, J., and Horowitz, M. (1989). An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215.

Albonesi, D. H. (1999). Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 248–259, Washington, DC, USA. IEEE Computer Society.

Anderson, G. and Baer, J.-L. (1994). *Design and evaluation of a subblock cache coherence protocol for bus-based multiprocessors*. Citeseer.

Bandara, S., Ehret, A., Kava, D., and Kinsy, M. (2019). Brisc-v: An open-source architecture design space exploration toolbox. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, pages 306–306, New York, NY, USA. ACM.

Bazm, M.-M., Sautereau, T., Lacoste, M., Sudholt, M., and Menaud, J.-M. (2018). Cache-based side-channel attacks detection through intel cache monitoring technology and hardware performance counters. In *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 7–12. IEEE.

Bernstein, D. J. (2005). *Cache-timing attacks on AES*. Available at `http://cr.yp.to/papers.html#cachetiming`.

Chen, L., Zou, X., Lei, J., and Liu, Z. (2007). Dynamically reconfigurable cache for low-power embedded system. In *Third International Conference on Natural Computation (ICNC 2007)*, volume 5, pages 180–184.

Chiappetta, M., Savas, E., and Yilmaz, C. (2016). Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174.

Collins, J. D. and Tullsen, D. M. (1999). Hardware identification of cache conflict misses. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 126–135, Washington, DC, USA. IEEE Computer Society.

Daemen, J. and Rijmen, V. (1999). Aes proposal: Rijndael. Available at `https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf`.

Dai, C. and Adegbija, T. (2017). Exploiting configurability as a defense against cache side channel attacks. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 495–500. IEEE.

Dhodapkar, A. S. and Smith, J. E. (2002). Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 233–244.

Domnitser, L., Jaleel, A., Loew, J., Abu-Ghazaleh, N., and Ponomarev, D. (2012). Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):35.

Dubnicki, C. and LeBlanc, T. J. (1992). Adjustable block size coherent caches. In *ACM SIGARCH Computer Architecture News*, volume 20, pages 170–180. ACM.

Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. (2016). Jump over aslr: Attacking branch predictors to bypass aslr. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 40. IEEE Press.

Gordon-Ross, A. and Vahid, F. (2007). A self-tuning configurable cache. In *Proceedings of the 44th annual Design Automation Conference*, pages 234–237. ACM.

Gordon-Ross, A., Vahid, F., and Dutt, N. D. (2009). Fast configurable-cache tuning with a unified second-level cache. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(1):80–91.

Gordon-Ross, A., Viana, P., Vahid, F., Najjar, W., and Barros, E. (2007). A one-shot configurable-cache tuner for improved energy and performance. In *2007 Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6. IEEE.

Harper, J. S., Kerbyson, D. J., and Nudd, G. R. (1999). Analytical modeling of set-associative cache behavior. *IEEE Transactions on Computers*, 48(10):1009–1024.

Hashemi, M., Swersky, K., Smith, J. A., Ayers, G., Litz, H., Chang, J., Kozyrakis, C., and Ranganathan, P. (2018). Learning memory access patterns. *arXiv preprint arXiv:1803.02329*.

Hund, R., Willems, C., and Holz, T. (2013). Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE.

Kadayif, I., Kandemir, M., Vijaykrishnan, N., Irwin, M. J., and Ramanujam, J. (2001). Morphable cache architectures: potential benefits. In *ACM SIGPLAN Notices*, volume 36, pages 128–137. ACM.

Kadiyala, M. and Bhuyan, L. N. (1995). A dynamic cache sub-block design to reduce false sharing. In *Proceedings of ICCD'95 International Conference on Computer Design. VLSI in Computers and Processors*, pages 313–318. IEEE.

Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2018). Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*.

Kumar, S. and Singh, P. (2016). An overview of modern cache memory and performance analysis of replacement policies. In *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, pages 210–214. IEEE.

Kumar, S., Zhao, H., Shriraman, A., Matthews, E., Dwarkadas, S., and Shannon, L. (2012). Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 376–388, Washington, DC, USA. IEEE Computer Society.

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown. *arXiv preprint arXiv:1801.01207*.

Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. (2015). Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622.

Malik, A., Moyer, B., and Cermak, D. (2000). A low power unified cache architecture providing power and performance flexibility (poster session). In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, ISLPED '00, pages 241–243, New York, NY, USA. ACM.

Mcilroy, R., Sevcik, J., Tebbi, T., Titzer, B. L., and Verwaest, T. (2019). Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178*.

Montanaro, J., Witek, R. T., Anne, K., Black, A. J., Cooper, E. M., Dobberpuhl, D. W., Donahue, P. M., Eno, J., Hoeppner, G. W., Kruckemyer, D., et al. (1997). A 160-mhz, 32-b, 0.5-w cmos risc microprocessor. *Digital Technical Journal*, 9:49–60.

Oren, Y., Kemerlis, V. P., Sethumadhavan, S., and Keromytis, A. D. (2015). The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1406–1418. ACM.

Osvik, D. A., Shamir, A., and Tromer, E. (2006). Cache attacks and countermeasures: the case of aes. In *Cryptographers track at the RSA conference*, pages 1–20. Springer.

Peir, J.-K., Lee, Y., and Hsu, W. W. (1998). Capturing dynamic memory reference behavior with adaptive cache topology. In *ACM SIGPLAN Notices*, volume 33, pages 240–250. ACM.

Peng, M., Sun, J., and Wang, Y. (2007). A phase-based self-tuning algorithm for reconfigurable cache. In *First International Conference on the Digital Society (ICDS'07)*, pages 27–27. Ieee.

Qureshi, M. K., Thompson, D., and Patt, Y. N. (2005). The v-way cache: demand-based associativity via global replacement. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 544–555. IEEE.

Ranganathan, P., Adve, S., and Jouppi, N. P. (2000). Reconfigurable caches and their application to media processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 214–224, New York, NY, USA. ACM.

Reddi, V. J., Settle, A., Connors, D. A., and Cohn, R. S. (2004). Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, page 22. ACM.

Sakr, M., Levitan, S. P., Chiarulli, D. M., Horne, B. G., and Giles, C. L. (1997). Predicting multiprocessor memory access patterns with learning models. In *Proceedings of the 14th International Conference on Machine Learning (ICML 1997)*, pages 305–312.

Segars, S. (2001). Low power design techniques for microprocessors. In *IEEE International Solid-State Circuits Conference (Tutorial), Feb. 2001.*

Suh, G. E., Devadas, S., and Rudolph, L. (2014). Analytical cache models with applications to cache partitioning. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 323–334. ACM.

Tao, J., Kunze, M., Nowak, F., Buchty, R., and Karl, W. (2008). Performance advantage of reconfigurable cache design on multicore processor systems. *International Journal of Parallel Programming*, 36(3):347–360.

Tradowsky, C., Cordero, E., Orsinger, C., Vesper, M., and Becker, J. (2016). Adaptive cache structures. In *International Conference on Architecture of Computing Systems*, pages 87–99. Springer.

Vijaykrishnan, N., Kandemir, M., Irwin, M. J., Kim, H. S., and Ye, W. (2000). Energy-driven integrated hardware-software optimizations using simplepower. *ACM SIGARCH Computer Architecture News*, 28(2):95–106.

Vila, P., Köpf, B., and Morales, J. F. (2018). Theory and practice of finding eviction sets. *arXiv preprint arXiv:1810.01497*.

Wang, Z. and Lee, R. B. (2007). New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 494–505. ACM.

Wang, Z. and Lee, R. B. (2008). A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 83–93. IEEE Computer Society.

Yarom, Y. and Falkner, K. (2014). Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, volume 1, pages 22–25.

Yu, S., Gui, X., and Lin, J. (2013). An approach with two-stage mode to detect cache-based side channel attacks. In *The International Conference on Information Networking 2013 (ICOIN)*, pages 186–191.

Zhang, C., Vahid, F., and Lysecky, R. (2004). A self-tuning cache architecture for embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(2):407–425.

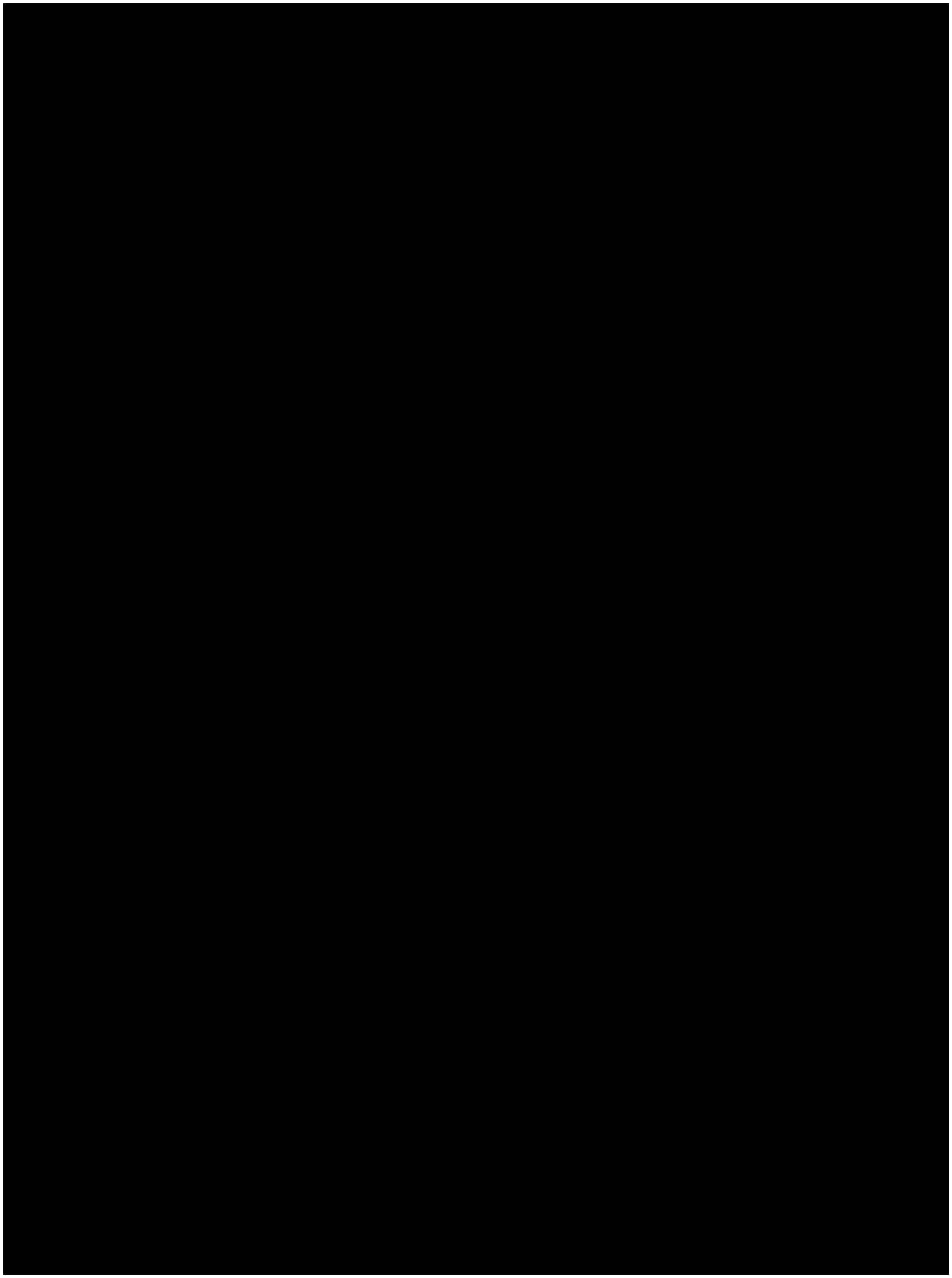
Zhang, C., Vahid, F., and Najjar, W. (2003). A highly configurable cache architecture for embedded systems. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 136–146. IEEE.

Zhang, T., Zhang, Y., and Lee, R. B. (2016). Cloudradar: A real-time side-channel attack detection system in clouds. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 118–140. Springer.

Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. (2012). Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316. ACM.

CURRICULUM VITAE

[redacted]



[redacted]