A Review of the impact of Cache Configuration and Context Switching on Cache Performance

Randika Viraj
Department of Computer Engineering,
University of Peradeniya,
Peradeniya, Sri Lanka
e16332@eng.pdn.ac.lk

Wishva Madushanka
Department of Computer Engineering,
University of Peradeniya,
Peradeniya, Sri Lanka
e16222@eng.pdn.ac.lk

Isuru Nawinne
Department of Computer Engineering,
University of Peradeniya,
Peradeniya, Sri Lanka
isurunawinne@eng.pdn.ac.lk

Roshan Ragel
Department of Computer Engineering,
University of Peradeniya,
Peradeniya, Sri Lanka
roshanr@eng.pdn.ac.lk

Mahanama Wickramasinghe
Department of Computer Engineering,
University of Peradeniya,
Peradeniya, Sri Lanka
mahanamaw@eng.pdn.ac.lk

Haris Javaid
AMD
Singapore
haris.javaid@amd.com

***Abstract*—Caches are used to optimize memory access time and energy consumption. Due to different memory access patterns, Cache configuration for a particular process has an impact on cache performance. Although When we do the context switch, the data stored in the cache of the current thread should have to flush and load new data from the new thread. This will increase memory access time and energy consumption. This overhead has a significant impact on cache performance. There are many techniques and methods presented to increase the cache performance in the context of cache configuration and the effect of context switches for the currently running process. In this review paper, we discuss those methods and techniques.**

***Index Terms*—Computer Architecture, Operating Systems, Systems-on-chip Design**

I. Introduction

The speed of the CPU has increased more quickly than the speed of DRAM memory over the past few decades. Each is getting better exponentially, but their differences are getting bigger exponentially as well. The time it takes to access DRAM doubles every 6.2 years in terms of instructions [14] due to DRAM memory latency improvements of only 7% per year over the past few decades compared to CPU speed improvements of 50 to 100% per year [13]. As a result, hierarchical memory systems made up of a variety of memory devices with varying performance and capacity have been developed. A hierarchical memory's design objective is to make it appear as though the entire system is made up of the fastest devices while in reality its cost is dominated by slower, less expensive, higher capacity devices. As one moves down the hierarchy, the frequency of access to each memory device decreases, which is essential for the hierarchy to function. The locality of reference principle [15] serves as the foundation for this. The cache is the part of the memory hierarchy that matters the most in terms of complexity and performance. A cache is a compact, quick buffer where the processor can store pieces of larger, slower memory that are likely to be needed soon. By offering an average memory access time, a cache serves the purpose of masking the latency of the slower memory.

The time between when the CPU requests a word and when the desired word arrives is referred to as cache or memory access time. A cache is a section of main memory that is organized into blocks called lines. Each block is the smallest unit of storage in a cache that is transferred between the cache and memory during a cache miss and contains several words. A tag, index, and offset are used to uniquely address each block. The offset field selects the desired data from the block, the index field selects the set, and the tag field compares whether a hit occurs.

Without cache memories, modern high-performance computer architectures would not be possible. However, the disparity in system performance between the processor and the cache memory has harmed overall system performance ever since the first cache memory implementations [11]. The performance difference between processor and cache systems was identified as early as the 1970s [12].

When it comes to today's processor world, the multiprocessor paradigm has a great impact on performance. Researchers are focusing on the impact of multiprocessor systems on the domain of caches. Two studies of the effects of caches on multi-processor systems performance are indicated below,

- Effect of context switching on cache performance
- Relation between cache configuration and the cache performance

To address the above-mentioned areas, researchers have heavily researched, and continue to research in the areas of Computer Architecture and Operating systems.

II. OBJECTIVES AND CONTRIBUTIONS

The objective of this paper is to taken the reader to the forefront of the battle to improve the performance of multiprocessor systems in the context of cache performance. We are

concentrated our efforts on a few key areas, which are further discussed below.

1) In 3," EFFECT OF CACHE CONFIGURATION ON SYSTEM PERFORMANCE". Paper is pointed to how the cache configuration affects a process. Paper is included the review of novel advancements in Switchable Caches and Dynamic Cache configurations.
2) In 4, "THE EFFECT OF CONTEXT SWITCHING ON CACHE PERFORMANCE", Paper is discussed how context switching impacts cache performance, and is discussed advances in reducing the effect on cache performance on the techniques of dynamic cache partitioning, dynamic cache switching.
3) In 5, "CONCLUSION", Paper is summarized the findings and provide a case study of the effect of cache configuration and context switching on the domain cache performance in embedded systems.

III. EFFECT OF CACHE CONFIGURATION ON SYSTEM PERFORMANCE

A. Background

Cache memories are a very important part of almost every hardware system. It improves performance between processor and memory as well as reduces energy consumption in memory access [16], [27]. Caches can be represented by C,k, and L where C is capacity, k is associativity and L is the line size. By choosing suitable C,k, and L values, different cache configurations can be expected. Direct cache map, set associativity and full associativity are the three types of caches.

During the program execution, the memory address is mapped to the cache line depending on the cache configuration. When accessing the cache, the memory address(physical address) is divided in the following way.

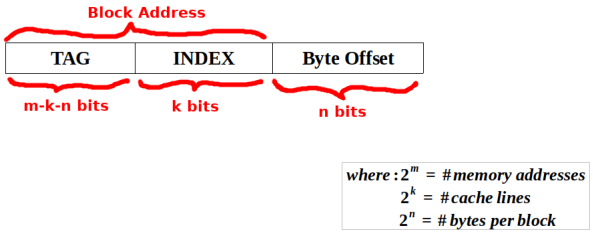

Fig. 1. Cache memory addressing format.

In a direct map cache line is always mapped to a fixed position in a cache. In set associativity cache, set of lines mapped to x position in cache (x-way set associativity cache). A one-way set associativity cache is the same as a direct map cache. In full associativity cache lines can be mapped to any position in the cache. A full associativity cache can be called N-way set associativity as well where N is the number of blocks in the cache.

When cache access happens, if the line is found in the cache, it's called a cache hit while if the line is not found in the cache it's called a cache miss. Mainly there are three types of cache misses. They are Cold miss, conflict miss and capacity miss. When the program starts all the cache memory data is invalid. Those miss called cold misses. Cold misses cannot be reduced. Conflict miss happens when the cache memory is already filled for a given address in a direct map or cache set already filled in the set associativity cache. Conflict miss can be reduced by increasing ways in set associativity cache. The capacity misses are the difference between the misses in a given cache and the miss in a fully associative cache of the same size. Even though caches increase the performance of the systems, According to [20], [33] having a large cache does not provide fast memory access or less energy consumption. Cache performance is dependent on temporal and spatial localities. The data or instruction that the CPU needs to read from the present main memory address is also saved in the cache memory because it may be needed again soon. This is known as temporal locality. Data or instructions close to the memory location being fetched may be required soon. Because of that, when the memory access happens, instruction or data near to current memory access are fetched at the same time. This is known as spatial locality. Spatial locality determines how data is loaded into the cache while temporal locality decides how data is removed from the cache.

B. Importance Of Cache Configuration

In embedded systems, one program or a couple of programs execute always. Most of the embedded system's power source is a battery. Reducing power consumption systems like that is very important. The power used by a microprocessor's memory hierarchy can account for up to 50% of the power used by the entire microprocessor system. Cache optimization is an important fact in the embedded system context.

Program to program memory access pattern is not uniform. Because of that, the best cache configuration differs from program to program. Even though caches improve performance in memory access and reduce power consumption, According to [20], [33] using the best cache for a particular program improves performance and reduces power consumption even more in the context of embedded systems.

C. Related works

1) Reconfigure one cache parameter:

A.Patel et al.(2006) [21] propose an improved indexing schema for direct-mapped caches. This method decreases the number of conflict misses significantly by using application-specific data. Simply choose a subset of the address bit which can reduce the number of conflict misses. The solution's two key advantages are First, it gives a symbolic algorithm to compute the best solution based on an analytical model for the conflict-miss conditions of a given trace. Second, it enables the best cache indexing to meet a specific application according to a reconfigurable bit selector that can be configured at runtime. Using standard benchmarks, they have shown with their method, that conflict misses reduce by 24%.

Albonsei et al.(1999) [22] propose a method called selective cache ways with a minor hardware modification that provides

the ability to enable a subset of cache ways when the demand increases. When great performance is necessary, all cache ways are enabled, however, when cache demands are lower, only a portion of the ways are enabled. They have shown that 4-way set associative caches can provide a 40% reduction in overall cache energy dissipation with less than a 2% overall performance deterioration.

In set associativity cache, memory data is not utilized across the sets. Because of that cache performance decreases due to underutilization of the cache. Rolan et al.(2009) [23] propose a method called "Set balancing" for balancing the set's pressure. Set Balancing Cache or SBC moves lines from sets with high local miss rates to sets with underutilized lines, where they can subsequently be found. While an increase in associativity is equivalent to merging sets randomly, this method only uses the resources of several sets in concert when it appears to be advantageous. Increases in associativity are unable to decide which sets to merge, whereas the SBC may be implemented using either a static policy, which also determines which sets can be associated or a dynamic one, which allows any set to be associated with any other set. In comparison to equal associative growth without its drawbacks, the SBC performs better. Authors show that with this method The miss rate was reduced by an average of 9.2% and 12.8% for the static and dynamic SBC, respectively, or 14% and 19% computed as the geometric mean, in experiments utilizing 10 representative benchmarks from the SPEC CPU2006 suite.

Zhang et al.(2003) [29] propose new cache architecture that can be configured by software. The cache can be set up in software to be directly mapped or set associative while still using the entire cache's capacity by setting a few bits in a configuration register. They called this technique a way concatenation and they extend this to support way shutdown.

2) Reconfigure all parameters:

Gordon-Ross et al.(2007) [17] introduce non-intrusive on-chip cache-tuning hardware modules that can predict the accurate best configuration for an executing program. With this method, addresses generated by a microprocessor are non-intrusively collected. Then it analyzes those data to predict the best cache configuration. Then instantly updates the cache to the new best configuration, never looking at subpar configurations. As a result, there is less energy consumption and performance overhead, allowing for more frequent cache tuning. Through experiments, authors have shown that the one-shot cache tuner can reduce memory-access related energy for instructions by 35% and comes within 4% of a previous intrusive approach. It also results in 4.6 times less energy overhead and 7.7 times faster tuning time compared to a previous intrusive approach, but at the main cost of 12% larger size. After practically implementing this one-shot tuner they suggest In scenarios with very high persistence, it is suggested that the invasive heuristic is still the best choice. However, where applications or phases frequently vary, the non-invasive one-shot tuner is much superior. As a future work authors propose to extend the one-shot tuner to multiple levels of cache.

Zhang et al.(2004) [18] created configurable cache architecture. Four parameters are included in a configurable cache design.

1) cache size, which can be configured as 8 Kbytes, 4 Kbytes, or 2 Kbytes. Through way shutdown, they allow cache size tune. With less than a 1% area increase, the critical path is increased by about 5%.
2) associativity, which can be 4-way, 2-way, or 1-way (direct mapped). : Technique authors called way concatenation is used to implement the configuration of associativity. Four banks that can be used in four different ways make up the base cache. The ways can be effectively "concatenated" by configuring a tiny register, producing either a two-way or direct-mapped 8 Kbyte cache. They replace the inverter (word line driver, output inverter of the comparator) with NAND gates.
3) line size, which can be 16 bytes, 32 bytes, or 64 bytes.
4) way prediction, which can be turned off or on (but is always off if the cache is configured as direct-mapped).

With this implementation, authors showed that on average over 40% memory access energy savings for Powerstone and MediaBench benchmarks, and up to 70% on certain benchmarks and the Typical performance overhead of this customizable cache is less than 2%. Zhang et al.(2004) [18] present a self-tuning strategy as well. It's done by small additional hardware that can be enabled and disabled by software. By analyzing each parameter, they develop heuristics that minimize the number of configurations examined during tuning and minimize the need for cache flushing.

Gordon-Ross et al.(2007) [19] present a self-tuning cache design that dynamically invokes a self-tuning mode. Self-tuning mode is active at a specified tuning interval and then explores the configuration space to predict the best cache configuration for the currently running program. Gordon-Ross et al.(2007) [19] present a method for dynamically adjusting the tuning interval. For that, they design a feedback-controlled self-tuner that can examine execution and determine tuning intervals. This method reduces average energy by 29% over a base cache. The best non-Oracle tuner that tunes at half the phase interval saves 11% of the energy while 13% saves the oracle. As a Future work authors suggest improving the self-tuner to make it more durable in a wide range of conditions.

Janapsatyat et al.(2006) [20] provide a method for quickly and correctly examining the cache design space by simultaneously predicting the cache miss rates for numerous distinct cache configurations and examining the impact of various cache configurations on a system's energy use and performance. By precisely and effectively simulating the number of cache misses that would occur for a given set of cache configurations, they do design space exploration on the cache parameters. By simulating numerous cache configurations simultaneously and consolidating multiple readings of extensive program traces into a single reading, they reduce the runtime of their cache simulation. To determine the ideal cache setup for the specified embedded application, they created an energy model and an execution time model. Using Mediabench

benchmarks, they discover that their approach explores the design space on average 45 times faster than Dinero IV while maintaining 100% correctness.

Dynamic cache reconfiguration reduces energy consumption as well as improves overall system performance. To reduce energy consumption while maintaining the same service level, Li et al.(2017) [24] provide a novel method for implementing cache reconfiguration in soft real-time systems. For both statically and dynamically scheduled real-time systems, This methodology offers an effective scheduling-aware cache optimization strategy based on static profiling. But here they have considered only the L1 cache. With little to no effect on timing constraints, this methodology uses the optimum blend of static analysis and dynamic cache parameter customization. As a result, The cache subsystem in soft real-time embedded systems uses 50% less energy overall as a result of the authors' strategy, according to experimental evidence presented in [24].

With the technology development, the number of transistors in the chip increases. Due to that power consumption in the chip increased and at once the whole chip cannot be powered on. According to [26], 93.75% of chip design is powered off to keep safe temperatures in the chip. Having the most suitable cache configuration for running programs increases the performance in memory access as well. As a solution to these problems Nawinne et al.(2016) [25] come up with a new concept called "switchable cache". By leveraging the abundant transistors available in chips due to the dark silicon phenomenon, the switchable cache consists of multiple caches that can configure each one. At runtime, only one cache configuration that is most suitable for running applications is active while another cache configuration is inactive. Authors introduce new design space exploration methods to find optimal cache configuration for the switchable cache. For a group of applications that are expected to run in the system, The space exploration algorithm is capable of finding an optimal cache configuration set within 2s by analyzing trillions of configurations. With the experiment result, the authors showed that the switchable cache improved memory access time by up to 26.2% compared to the fixed cache. In the future authors wish to expand this switchable concept into a multiprocessor system.

The method presented by the A.Patel et al.(2006) in [21] only considers direct map cache, but it reduces the conflict misses. The method presented by [22] enables and disables cache-way depending on the demand. Rather than using all caches-way even though it's not used, this method reduces power consumption. In set associativity cache, sometimes all caches-way are not fully utilized. Rolan et al.(2009) [23] present a method for this problem. With this method cache utilization increases, hence it leads to increased cache performance. The method presented by Zhang et al.(2003) [29] uses software to configure the cache. But in this method, it can only configure set-associativity.

The method in [17] is fast and it has a high-performance impact because it chooses the best cache configuration never going to a lower configuration. In this method cache configuration is chosen by past data addresses and never looking at future addresses. Programs like memory access pattern change continuously give a low performance in this method. Zhang et al.(2004) [18] present methods that have self-tuning hardware parts which can be enabled and disabled by software. Rather than continuously changing cache configuration, for some applications using a fixed best cache configuration reduces power consumption. With the flexibility to enable and disable this feature, this method increases the scope of the range of embedded systems that can use this method. But this method uses additional hardware parts that increase additional power consumption and area. Unlike other methods, methods presented by [19], tune the cache at specified intervals and they propose a method for adjusting the interval by using a feedback system as well. On some occasions rather than continuously changing the best cache configuration, it's good to change the configuration by interval because every time that performs cache configuration requires power as well as additional work that may lead to decreased performance. The method presented by Nawinne et al.(2016) [25] uses a completely different technique, they leverage the cache solving problem that happens because of the dark silicon phenomenon. Here the cache configuration is defined at design time considering programs that are supposed to run in the system. In this method the program does not use the best cache configuration, it reduces performance. But considering all programs that are running in the system, the method increases overall performance.

IV. THE EFFECT OF CONTEXT SWITCHING ON CACHE PERFORMANCE

Context switching is the process of switching the processor from the kernel to process A and vice versa [1]. When processes need to be run and completed quickly and effectively, context switching is crucial to the process lifecycle. It is the process in the operating system where the CPU is switched from one process to another. The process of storing a process's state in an embedded system so that it can be later restored or restarted from the same point will allow multiple processes to share a single CPU. It Saves the registers, stack pointer, program counter and process status of the old process in the process table entry for it. In this section we discuss,

- The effect of the context switching to the cache performance
- Effect of fast context switching on the cache performance
- Comparison of context switching and cache address schema
- Cache performance on cache partitioning
- Dynamic cache switching and context switching

A. The effect of the context switching to the cache performance

The performance of a processor depends on the cache performance. A well-performed program should execute nearly one instruction per cycle but because of the memory accessing takes more clock cycles. We can't achieve that rate. Cache

performance mainly depends on the locality. When we consider the multiprocessor system, the operating system switches the context to managing the multiple processes, but in this case, the main theory of locality is being violated because the instructions or data of the newly scheduled process may no longer be in the cache. So the cache flushing and reloading of context switching takes more cost on the performance of the CPU. Several researchers measure the cost of context switching. Mogul et al.(1991) [2] proposed an experiment to measure the cache performance cost while the context switching for a variety of programs executing through simulating several different cache organizations. It describes clearly the effect on cache while the context switching.In the proposed method [2] run their experiment on a UNIX system by concurrently running a group of programs. And by feeding the instructions and data address traces which are generated by executions, into a cache simulator. The short interval of time after the context switch is the time which has more impact on the cache performance, and [2] have limited it to that interval. Their study shows that cache misses and context switching make the processor performance worse because the memory speed is much slower than the processor speed. Because of the burst of cache misses which happen on context switching has a great impact on performance loss. Also manuscript of [3]. research the effect of operating systems and multiprogramming on cache performance. To obtain accurate cache performance statistics quickly, they combined trace sampling with a novel steady-state analysis method, called ATUM, to capture distortion-free system traces of large multitasking workloads. They compared the uni process effect on the cache and the multi-process effect on the cache. Their results indicate that multi-processor systems decrease the cache performance but large caches decrease this impact when purging the content after cache switching. But the small caches do not indicate that much performance according to Agarwal et al.(1988) [3]. Also [6]–[8] did their research on cache-related preemptive context switching delay. This research has focused on the cost of flushing a cache when memory blocks that were previously overwritten by another task need to be re-fetched.To make scheduling algorithms more effective, the emphasis is on calculating these costs.

B. Effect of fast context switching on the cache performance

Snyder et al.(1995) [4] have proposed a method for reducing the overhead due to context switching. They have proposed this method for real-time systems which can require very frequent preemptive context switching. This method tries to reduce the time takes for context switching. In this method, when tracing the execution of a program, the state of each register is examined. If the register is in use, called live, if not called to be dead. When context switching happens, if there are no live registers, then the cost of context switching can be reduced by saving and restoring the Program counter and stack pointer. Using this concept, a group of registers which are dead can be identified by a data flow analysis of a function in a program. They also proposed an investigated optimization called register remapping to provide a greater number of instructions with zero live registers. This technique reduces the total number of live registers for each instruction in a group of instructions that can be potentially executed contiguously with the same live register. This method does not require any addition of extra instructions or memory references. This leads them to investigate the effect of cache performance at the changing point when the context switch occurs. And the results of the research show that this approach leads to fast context switching. They have done their analysis on the context switch interval. The length of time intervals between desired context switches is called the context switch intervals. Both the number of memory references avoided and the cache performance are affected by the context switch interval. They have found that the effect of context switch interval on the effect of the percentage of fast context switches is less. But at the same time, their overall percentage of fast context switching increased by reducing the context switch interval. Their analysis of cache performance on whether the fast context switching has better performance over the slow context switching is overly negligible because there was no clear distinction on the clue. But the instruction cache which had fast context switching, had a better cache performance. Other hand the test program which used a slower context switch had better data cache performance.

C. Comparison of context switching & cache address schema

Hyok-Sung et al.(2005) [5] have performed a comparison on context switching and IPC performance between uClinux and Linux on the ARM8-based Processor uClinux is a derivation of Linux kernel which provides a single shared address space for all processes, intended for MMU fewer processors. But the Linux kernel provides separate virtual address space for each process. According to the manuscript, The architecture's cache algorithms affect how quickly an operating system switches between contexts when it supports virtual address space and the addressing scheme can divide the cache architecture into virtual cache and physical cache. The physical cache stores with a physical address, while the virtual cache stores the location of a specific context with a virtual address. According to the results of the manuscript, UClinux has shown much-improved performance while context switching. Linux has shown much delay and IPC, this results from the Linux kernel's virtual address usage for cache architecture and support for virtual address spaces, which require invalidating all caches and cause a constant amount of cache-miss load whenever processes are switched between contexts. In Linux after context switching, every memory access results in a cache miss on the first attempt, creating a significant load. The manuscript proved that a type of application that requires rapid context switching, as do time-critical embedded systems, will greatly benefit from the use of uClinux.

D. Cache performance on cache partitioning

In this approach, the cache is partitioned among the threads or processes. There are several ways this can be archived, a

partition per process, a special partition for a high-priority process and a set of the pool for the other process. Cache partitioning can be implemented using either software or hardware. These approaches can be further classified based on the structure of a set-associative cache into index-based cache partitioning, in which partitions are formed as an aggregation of associative sets in the cache, and way-based cache partitioning, in which each available cache way is exclusively assigned to a single partition. This classification is out of our scope of the survey.

1) Hardware Partitioning:

In the hardware solutions need special hardware units to control the cache partitioning. The first hardware-based implementation of a cache mechanism designed to provide predictability for uniprocessor real-time systems was proposed by the [37]. In this approach, the cache is divided into a fixed number of segments and these segments are mapped into the processes. Shared data structures which are accessed by two or many threads(processes) are stored in a specific cache partition titled shared partition. In this approach, the shared partition can consist of a set of segments. Although to get more performance some processes can have private caches. The cache id is used to identify the segments assigned to the task, segment count and hardware flags which indicate whether accessing a private or shared segment has to be loaded in every context switch. So this leads to an extra overhead also but gives performance increases rather than using a single cache in a multiprocessor system. The method proposed in [36], proposed an algorithm called CQoS. This algorithm helps to identify the applications which have higher priority memory accessing. Then it allocates set partitions depending on the priority. In the proposed method of [34], the shared L2 cache selects the cache set according to the CPU ID. In this method, a cache controller which is configurable by the OS is used to map the CPU Id to associate its set. When memory access happens first look for a dedicated set and also can look into the other sets before going into the main memory. But in this method, there is no mechanism to resolve the cache misses which are affecting the inter-core. Also proposed method in [35] follows the same strategy but it has an advanced technique to resolve inter-core interference, in this method they use a dedicated private core-owned cache for each core. This core is used to store the memory blocks that are affected by inter-core cache misses. In the method proposed in [38], the cache partition mechanism can perform three partition mechanisms. There are no partitions that any process can occupy the cache, process base partitioning where each process map to the portion of the cache, and core-based partition where each core is a map to the portion of the cache. Cache partitioning is then combined with either a static or dynamic cache-locking mechanism. The experiment was carried out on a dual-core processor with 2-way set-associative L1 and L2 caches. Caches led to the development of a set of guiding design principles for real-time systems, they are core-based partitioning is the best cache-partitioning method to use with static cache locking, core-based partitioning is the best option regardless of locking strategy and dynamic cache locking is superior to static cache locking only when tasks have a large number of hot regions and a small shared cache.

E.Suh et al.(2001) [10] proposed a dynamic cache partitioning method for multithreading systems. In this method, they are presenting an algorithm that uses dynamically partitions the cache. The miss-rate characteristics of each thread are estimated at runtime by this online cache partitioning algorithm, which dynamically distributes the cache among the running threads. To determine which partition minimizes the overall number of misses, the algorithm estimates marginal gains as a function of cache size. In their novel cache partitioning plan, a cache miss only gives a thread a new cache block if its current allocation is below the limit. They use counters in the cache that provide real-time estimates of individual thread miss rates in order to implement this scheme. Based on these counters, they improve LRU replacement or use column caching, which enables threads to be assigned to overlapping partitions, to better distribute cache resources to threads. For set-associative caches with multiple threads and a coarse-grained partition, their suggested partition is effective. A trace-driven cache simulator has been used to implement the partitioning algorithm. According to their simulation results, partitioning can significantly outperform the traditional LRU replacement policy in terms of cache performance for a range of cache sizes for specific threads. The outcomes of the simulations have demonstrated that, for a variety of cache sizes, the partitioning algorithm can address the issue of thread interference in caches. However, they noted that if caches are too small for the workloads, partitioning by itself cannot improve performance. Therefore, it is important to choose concurrently running threads carefully, taking into account their memory reference behavior.

2) Software Partitioning:

When it comes to software based solutions, proposed methods are using os or compiler support for partitioning the cache. Page coloring is the most popular software-based cache-partitioning method. When caches are physically indexed, page colouring examines the virtual to physical page address translations that are available in virtual memory systems at the OS level. The mapping of page addresses to pre-defined cache regions prevents cache space overlap. There are many shared cache partition techniques recently proposed on the page colouring .In the proposed method [39] introduced a page-colouring mechanism in the Linux kernel for x86 processors. In this method, custom memory management used the Linux kernel to support multiple lists. These lists consist of pages that have the same colour. The round robin algorithm is used to search the list when a process gets a page fault. Static and dynamic cache-partitioning policies were both supported. Each process's assigned memory colours at the start of execution are specified by the static policy. Every time a process requests more cache resources, the dynamic policy enforces a page

recolouring. By rebuilding the virtual to physical memory mapping, page recolouring is accomplished. Data is copied from an old page to a new page with a new colour when a page is recoloured. Also, there is more recent research on cache partition based on the compiler techniques. [40] is the penior of introducing compiler support for the cache partition. The cache size and task partition size are additional inputs given to the compiler. The linker combines the final object files into an executable after dividing them into code and data partitions for each task. The compiler inserts an unconditional jump to the following code partition at the end of each code partition to handle code partitions that are larger than the partition size. As the authors indicate, by adjusting the stack pointer as needed, local data on the stack is divided into partitions. As long as memory requests don't go beyond the cache partition size, dynamic allocation on the heap is supported. Libraries and operating systems are also handled as separate partitions. The method targets real-time uniprocessor systems and uses a page colouring-based cache partitioning mechanism. In this method, each partition is kept in its object file, which may be extended to the exact size specified by the cache partition size by padding the end with no-ops. The size of the data cache partition is used to divide global data into memory chunks.No data structure that spans multiple partitions is allowed, thanks to the compiler. The compiler must transform the access to the data structure if the size of the data structure is larger than the size of the cache partition.

The main advantage of cache partitioning is that it offers protection against the eviction of one core or partition from another. There are two main methods to accomplish this. In hardware methods, the processor comes with built-in support for partitioning the cache, giving each system core access to a separate area. But in software methods, it is ensured that no two cores or partitions will end up using the same section of the cache by placing the software in specific memory blocks and using techniques like cache colouring. Cache partitioning overly helps to get a performance improvement in the context switching, because timing verification is minimized and inter-task cache eviction is avoided when a process is given its cache partition. But cache partitioning has the drawback that, depending on the characteristics of the hosted application, it may significantly affect performance in both the average and worst-case scenarios. Because each core or partition now has a smaller portion of the cache to work with, if it cannot fit, it will experience an increased cache miss rate, which directly affects execution time. Although, a shared cache is susceptible to timing channel attacks, which leak confidential data from one process to another, according to earlier research [41]–[43]. So static cache partitioning is suggested in [44] as a defense against cache timing channel attacks. In this approach, the shared cache is statically divided into numerous partitions when using static cache partitioning. Because each application is limited to using its cache partition, cache interference is effectively eliminated. But in this case, there is a huge performance overhead because cache partition size and configuration are independent of the cache behavior of each application. Although dynamic cache partitioning is still a security threat because the partitioning policy relies on the run-time behavior of each application.

E. Dynamic cache switching and context switching

The possibility of dynamic cache switching to enhance performance arises as reconfigurable softcore systems gain in popularity and run multiple operating systems and applications. The new research idea of dynamic cache switching in a preemptive softcore system was proposed by [9]. They try to optimize the cache performance on context switching. Dynamic cache switching involves altering cache behaviour while it is running to conform to the active application. It makes use of a general-purpose cache controller as well as several specialized cache controllers designed for particular applications. According to their proposed method, applications that are not profiled or that do not heavily utilize the cache use the generic cache controller. In response to profiled applications connected to the specialized caches, the pre-built specialized cache controllers are switched on and off. In this method, they try to archive the Optimisation of a subset of the applications. They have proposed this method for embedded because the majority of the time, a known set of primary applications run on embedded systems with softcore processors. Since these applications are typically known at the time of synthesis, specialized caches can be created to support them. In their approach they have described the main things that are needed to archive dynamic cache switching, they are a cache management system, cache configurations to switch and analysis of which cache to switch between. This paper demonstrates a creative and useful technique for software developers to implement cache switching. Also, they investigate the idea of cache mismatch, a more accurate metric for assessing performance enhancements in dynamic cache switching. In this idea, during a context switch, the performance loss of switching to another cache which is optimized for the new thread and the performance loss of using the cache configured for the previous thread is calculated. This optimization mismatch informs the switching algorithm when switching to a different cache configuration is advantageous. It is not worthwhile to perform a cache switch if the optimization mismatch is greater than the overheads of cache switching. So that in this method there is a chance to decide on the previous data cache for the new thread, then cache flushing overhead would happen.

V. CONCLUTION

As we discussed so far, we can say that context switching and cache configuration impact the performance of the cache. According to these methods rather than using a fixed cache for all programs, it's proven that all techniques that configure caches according to the memory access pattern, improve cache performances. The statical cache configuration method looks at the overall group of programs and configures caches according to that to improve overall throughput. The dynamic cache configuration method changes caches at runtime according to

the current running program. All those cache configuration methods add extra complexity and additional power consumption to the exciting system, there is a trade-off because of that. Therefore there is no best or superb method, depending on the program that runs on the embedded system we need to choose the most suitable cache configuration method. When it comes to context switching, researchers have proposed many techniques to reduce the impact on cache performance. Cache partitioning is the most frequently used technique. Cache partitioning can be implemented using either software or hardware. In the hardware-based approach, a special hardware unit is required. In software-based solutions, proposed methods are using os or compiler support for partitioning the cache, and page colouring is the most popular software-based cache-partitioning method. But cache partitioning has the drawback that, depending on the characteristics of the hosted application, because each core or partition now has a smaller portion of the cache to work with, if it cannot fit, it will experience an increased cache miss rate. Also, a popular novel idea is dynamic cache switching. This method involves altering cache behaviour while it is running by switching between cache controllers. With all the references we discussed, we can see that, when it comes to application-specific processors, the cache configuration and switchable caches are a major concern to improve the performance of memory access. There is a clear research gap between the combination of these two areas. Also, there is huge research space on how to map multiprocessor systems on a set of switchable caches.

REFERENCES

[1] Finkel, R. A. (1988). An Operating Systems Vade Mecum.

[2] Mogul, J. C., Borg, A. (1991). The Effect of Context Switches on Cache Performance. ACM SIGPLAN Notices, 26(4), 75–84.

[3] Agarwal, A., Hennessy, J., Horowitz, M. (1988). Cache Performance of Operating System and Multiprogramming Workloads. ACM Transactions on Computer Systems (TOCS), 6(4), 393–431.

[4] Snyder, J. S., Whalley, D. B., Baker, T. P. (1995). Fast context switches: compiler and architectural support for preemptive scheduling. Microprocessors and Microsystems, 19(1), 35–42.

[5] Hyok-Sung, C., Hee-Chui, Y. (2005). Context Switching and IPC Performance Comparison between uClinux and Linux on the ARM9 based Processor. In Samsung Technology Conference.

[6] Stärner, J. Asplund, L. (2004). Measuring the cache interference cost in preemptive real-time systems. Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), 146–154.

[7] Lee Joosun Hahn Yang-Min Seo Sang Lyul Min Rhan Ha Seongsoo Hong Chang Yun Park Minsuk Lee Chong Sang Kim, C.-G. (1997). Bounding Cache-related Preemption Delay for Real-time Systems. In ieeexplore.ieee.org.

[8] Staschulat, J., Schliecker, S., Ernst, R. (2005). Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. Proceedings - Euromicro Conference on Real-Time Systems, 2005, 41–48.

[9] Shield, J., Sutton, P. Machanick, P. (n.d.). DYNAMIC CACHE SWITCHING IN RECONFIGURABLE EMBEDDED SYSTEMS. In ieeexplore.ieee.org. Retrieved August 21, 2022.

[10] E. Suh, L. Rudolph, and S. Devadas.(2001). Dynamic Cache Partioning for Simultaneous Multithreading Systems Computation Structures Group Memo 438. In Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on, pages 111–116. IEEE, 2007.

[11] IBM Products Division, "System/370 model 168 theory of operation/diagrams manual (volume 1)," Poughkeepsie, NY, 1976.

[12] Goodman, J. R. (1983). Using cache memory to reduce processor-memory traffic. 124–131.

[13] J Hennessy and D Patterson. Computer Architecture: A Quantitative Approach (2nd edition), Morgan Kavfmann, San Mateo, CA, 1996.

[14] Boland, K., Dollas, A. (1994). Predicting and Precluding Problems with Memory Latency. In IEEE Micro (Vol. 14, Issue 4). https://doi.org/10.1109/40.296166.

[15] Denning, P. J. (1968). The working set model for program behavior. Communications of the ACM, 11(5), 323–333.

[16] Basu, A., Hill, M. D. Swift, M. M. (2012). Reducing memory reference energy with opportunistic virtual caching. ACM SIGARCH Computer Architecture News, 40(3), 297–308.

[17] Gordon-Ross, A., Viana, P., Vahid, F., Najjar, W., Barros, E. (2007). A one-shot configurable-cache tuner for improved energy and performance. Proceedings -Design, Automation and Test in Europe, DATE, 755–760. https://doi.org/10.1109/DATE.2007.364686

[18] Zhang, C., Vahid, F. Lysecky, R. (2004). A Self-Tuning Cache Architecture for Embedded Systems. ACM Transactions on Embedded Computing Systems, 3(2), 407–425.

[19] Gordon-Ross, A., Vahid, F. (2007). A self-tuning configurable cache. Proceedings - Design Automation Conference, 234–237. https://doi.org/10.1109/DAC.2007.375159

[20] Janapsatyat, A., Ignjatović, A., Parameswaran, S. (2006). Finding optimal LI cache configuration for embedded systems. Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC, 2006, 796–801.

[21] A., Patel, K., Benini, L., Member, S., Member, S., Macii, E., Poncino, M. (2006). Reducing conflict misses by application-specific reconfigurable indexing. Ieeexplore.Ieee.Org, 25(12).

[22] Albonesi, D. H. (1999). Selective cache ways: On-demand cache resource allocation. In Proceedings of the Annual International Symposium on Microarchitecture (Vol. 2). https://doi.org/10.1109/micro.1999.809463

[23] Rolán, D., Fraguela, B. B., Doallo, R. (2009). Adaptive line placement with the set balancing cache. Proceedings of the Annual International Symposium on Microarchitecture, MICRO, 529–540.

[24] Li, J., Wang, Z. L., Zhao, H., Gravina, R., Fortino, G., Jiang, Y., Tang, K. (2017). Networked human motion capture system based on quaternion navigation. BodyNets International Conference on Body Area Networks, 0(1).

[25] Nawinne, I., Javaid, H., Ragel, R., Parameswaran, S. (2016). Switchable cache: Utilising dark silicon for application specific cache optimisations. IET Computers and Digital Techniques, 10(4), 157–164.

[26] Taylor, M. B. (2012). Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. Proceedings - Design Automation Conference, 1131–1136. https://doi.org/10.1145/2228360.2228567

[27] Gordon-Ross, A., Zhang, C., Vahid, F., Dutt, N. (2005). Tuning Caches to Applications for Low-Energy Embedded Systems. Ultra Low-Power Electronics and Design, 103–122.

[28] Basu, A., Hill, M. D., Swift, M. M. (2012). Reducing memory reference energy with opportunistic virtual caching. ACM SIGARCH Computer Architecture News, 40(3), 297–308.

[29] Zhang, C., Vahid, F., Najjar, W. (2003). A highly configurable cache architecture for embedded systems. Association for Computing Machinery,136.

[30] Khatwal, R., Jain, M. K. (2014). Application Specific Cache Simulation Analysis for Application Specific Instruction set Processor. International Journal of Computer Applications, 90(13), 975–8887.

[31] Liang, Y., Mitra, T. (2008). Static analysis for Fast and accurate design space exploration of caches. Embedded Systems Week 2008 - Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2008, 103–108.

[32] Schneider, J., Peddersen, J., Parameswaran, S. (2014). MASH-fifo: A hardware-based multiple cache simulator for rapid FIFO cache analysis. Proceedings - Design Automation Conference. https://doi.org/10.1145/2593069.2593159

[33] Shwe, S. M. M., Javaid, H., Parameswaran, S. (2013). RExCache: Rapid exploration of unified last-level cache. Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC, 582–587. https://doi.org/10.1109/ASPDAC.2013.6509661

[34] Liu C., Sivasubramaniam A., Kandemir M. (2004). Organizing the last line of defense before hitting the memory wall for CMPs. In IEEE High-

Performance Computer Architecture Symposium Proceedings (Vol. 10). https://doi.org/10.1002/0471648272.ch11
[35] S. Srikantaiah, M. Kandemir, and M. J. Irwin. (2008). Adaptive set pinning: Managing shared caches in chip multiprocessors. In Proc. of the 13th ASPLOS. ACM, 135–144.
[36] Ravi Iyer. (2004). CQoS: A framework for enabling QoS in shared caches of CMP platforms. In Proc. of the18th ICS. ACM, 257–266.
[37] D. B. Kirk and J. K. Strosnider. (1990). SMART (strategic memory allocation for real-time) cache design using the MIPS R3000. In Proc. of the 11th RTSS. 322–330.
[38] Vivy Suhendra and Tulika Mitra. (2008). Exploring locking & partitioning for predictable shared caches on multi-cores. In Proc. of the 45th DAC. ACM, 300–303.
[39] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. (2008). Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In Proc. of the HPCA. IEEE, 367–378.
[40] Frank Mueller. (1995). Compiler support for software-based cache partitioning. In Proc. of the ACM SIGPLAN LCTES. ACM, 125–133.
[41] Bernstein, D. J. (2005). Cache-timing attacks on AES.Association for Computing Machinery.
[42] Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R. B. (2015). Last-level cache side-channel attacks are practical. Proceedings - IEEE Symposium on Security and Privacy, 2015-July, 605–622. https://doi.org/10.1109/SP.2015.43
[43] Percival, C. (2005). Cache missing for fun and profit. In BSDCan 2005. http://css.csail.mit.edu/6.858/2014/readings/ht-cache.pdf
[44] Page, D. (2005). Partitioned Cache Architecture as a Side-Channel Defence Mechanism. In IACR Cryptology ePrint Archive, Report 2005/280. https://eprint.iacr.org/2005/280.pdf