Automatic Characterization and Mitigation of Shared-resource Contention in Multi-core Systems

Jakob Danielsson

2021

Till min familj

Abstract

Multi-core computers are infamous for being hard to use in time-critical systems due to execution-time variations as an effect of shared resource contention. In this thesis, we study the problem of shared resource contention, which occurs when multiple applications executing on different cores do not have exclusive access to a shared hardware resource. We investigate performance variations of parallel tasks in multi-core systems and present a method to pinpoint the source of the contention using hardware performance counters. We investigate mitigation methods for performance variations due to resource contention, including the Jailhouse hypervisor and the cache-partitioning tool PALLOC. We propose a benchmark strategy that quantifies the isolation gained from a specific technique and exemplify this strategy using the Jailhouse hypervisor. We furthermore present and implement solutions for cache-partition allocation during application runtime. Our implementation aims to avoid over-provisioning of cache through pre-runtime estimations of an application's dependency towards the cache and continuous re-partitioning of the cache memory during application runtime.

The primary goal of this thesis is to contribute to a process that automates some of the tedious manual testing needed to detect resource contention bottlenecks. The methods we present in this provide a holistic solution for automatically mitigating resource-contention in a multi-core system. First, we evaluate the risk for shared resource contention when several applications execute simultaneously. We then allocate partitions to mitigate resource contention for applications that risk severe performance degradations. We finally present methods that dynamically re-allocate partition space to meet the performance requirements of the running applications.

Sammanfattning

Flerkärniga datorer är ökända för att vara svåra att använda i tidskritiska system på grund av prestandavariationer som sker på grund av samtidigt delande av hårdvaruresurser. I denna avhandling studerar vi problemet med delade resurser som uppstår när flera applikationer som körs på olika kärnor inte har exklusivt ägande av en delad resurs. Vi undersöker prestationsvariationer för parallella uppgifter i flerkärniga system och presenterar en metod för att identifiera källan till resurskonflikten med hjälp av befintliga hårdvaruprestationsräknare. Vi undersöker begränsningsmetoder för prestationsvariationer på grund av resurstvister, inklusive Jailhouse-hypervisor och cachepartitionsverktyget PALLOC. Vi föreslår en riktmärkesstrategi som kvantifierar isoleringen från en specifik isoleringsteknik och exemplifierar denna strategi med hjälp av Jailhouse -hypervisor. Vi presenterar och implementerar dessutom lösningar för tilldelningskontroll för cachepartitioner under applikationstiden. Vår implementering syftar till att undvika onödiga cacheallokeringar genom att uppskattninga programmets beroende av cacheminnet och kontinuerlig omallokering av cacheminnet medans applikationen kör.

Huvudmålet med denna avhandling är att underlätta den manuella testningen av resurskonflits-flaskhalsar och istället föreslå en automatiska metoder. De metoder vi presenterar ger en helhetslösning för automatisk lindring av resurskonflikter i ett flerkärnigt system. Först utvärderar vi risken för negativ påverkan genom delade resurser när flera applikationer körs samtidigt. Vi tilldelar sedan partitioner för att mildra resurskonflikter för applikationer som riskerar allvarliga prestandaförsämringar. Vi presenterar slutligen metoder som dynamiskt omallokerar cacheminne för att uppfylla prestandakraven för de applikationer som körs.

Acknowledgments

I would like to express my sincere gratitude towards my supervisors for their patience, help and valuable discussions throughtout this thesis. I would like to acknowledge each supervisor's special contribution to my thesis; Mikael Sjödin, for his deep knowledge and ability to guide me towards a feasible path when i've gotten stuck. Moris Behnam, for challenging my critical thinking and encouraging new ideas. Tiberiu Seceleanu for spending countless of hours of invaluable technical discussions and encouragement to pursue new ideas. Marcus Jägemar for teaching me, providing hands-on technical feedback and help and for being my source of inspiration and positive thinking.

The work presented in this thesis has been funded by Mälardalens Högskola and KKS throughout the DPAC project.

I would also like to thank my mother Annika Danielsson and father Christer Danielsson for supporting me. I want to acknowledge the value of the technical discussions that I have had with my father and for taking his time to read my thesis. I also thank my grandfather Bo Danielsson for helping me with photoshop.

I want to thank Nandinbaatar Tsog, my room-mate, my closest colleague, my "brother-in-arms". I am very grateful that I got the opportunity to work beside you for these years and for of the technical discussions that we have had.

My final expression of gratitude goes to Ida Carlén, my girlfriend, who has stood by my side during my master years and PhD student years.

Jakob Danielsson
Västerås, 2021

List of Publications

Papers included in thesis

Paper A : Jakob Danielsson, Marcus Jägemar, Tiberiu Seceleanu, Moris Behnam and Mikael Sjödin. Measurement-based evaluation of data-parallelism for OpenCV feature-detection algorithms. In *42^{nd} Computer Society Signature Conference on Computers, Software and Applications (COMPSAC)*. IEEE, 2018.

Paper B : Jakob Danielsson, Tiberiu Seceleanu, Marcus Jägemar, Moris Behnam and Mikael Sjödin. Resource Dependency Analysis in Multi-core systems. In *44^{th} Computer Society Signature Conference on Computers, Software and Applications (COMPSAC)*. IEEE, 2020.

Paper C : Jakob Danielsson, Tiberiu Seceleanu, Marcus Jägemar, Moris Behnam and Mikael Sjödin. LLM-shark – A Tool for Automatic Resource-boundness Analysis and Cache Partitioning Setup. In *45^{th} Computer Society Signature Conference on Computers, Software and Applications (COMPSAC)*. IEEE, 2021.

Paper D : Jakob Danielsson, Tiberiu Seceleanu, Marcus Jägemar, Moris Behnam and Mikael Sjödin. Testing Performance-Isolation in Multi-Core Systems. In *43^{rd} Computer Society Signature Conference on Computers, Software and Applications (COMPSAC)*. IEEE, 2019.

Paper E : Jakob Danielsson, Tiberiu Seceleanu, Marcus Jägemar, Moris Behnam and Mikael Sjödin. Automatic Quality of Service Control in Multi-core Systems using Cache Partitioning In *26^{th} In proceedings of the Emerging Technologies and Factory Automation (ETFA)*, IEEE, 2021.

Paper F : Jakob Danielsson, Marcus Jägemar, Tiberiu Seceleanu, Moris Behnam and Mikael Sjödin. Run-time Cache-Partition Controller for Multi-core Systems. In *45^{th} Annual Conference of the IEEE Industrial Electronics Society (IECON)* IEEE, 2019.

Paper G : Jakob Danielsson, Janne Suuronen, Tiberiu Seceleanu, Marcus Jägemar, Moris Behnam and Mikael Sjödin. Modelling Application Cache Behavior using Regression Models. In *IEESD: The 11^{th} International Workshop on Industrial Experience in Embedded Systems Design (IEESD)* IEEE, 2021.

Papers not included in thesis

Paper G : J. Danielsson, M. Ashjaei, M. Behnam, T. Sörensen, M. Sjödin, T. Nolte Performance Evaluation of Network Convergence Time Measurement Techniques. In *22nd Emerging Technologies and Factory Automation (ETFA)*, IEEE, 2017.

Paper H : J. Danielsson, N. Tsog and A. Kunnappilly A Systematic Mapping Study on Real-time Cloud Services In *1st workshop Quality Assurance in the Context of Cloud Computing (QA3C)*, IEEE, 2018.

Paper I : J. Danielsson, M. Jägemar, M. Behnam and M. Sjödin. Investigating Execution-Characteristics of Feature-Detection Algorithms. *Work in progress paper Published in proceedings of the 22nd Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2017.

Paper J : S. Imtiaz, J. Danielsson, M. Behnam, G. Capannini, J. Carlson and M. Jägemar Towards Automatic Application Fingerprinting Using Performance Monitoring Counters. In *7th Published in proceedings of Engineering of Computer Based Systems (ECBS)*, ACM, 2021.

Paper K : S. Imtiaz, J. Danielsson, M. Behnam, G. Capannini, J. Carlson and M. Jägemar Automatic Platform-Independent Monitoring and Ranking of Hardware Resource Utilization. In *26th Published in proceedings of the Emerging Technologies and Factory Automation (ETFA)*, IEEE, 2021.

Contents

Part I

Thesis

Chapter 1

Introduction

Multi-core systems are becoming the de-facto standard in both the commercial off-the-shelf (CoTS) and embedded computation domains. Multi-core processors present greater computational capacity while offering a decrease in size and weight than their single-core predecessors. Multi-core processors enable parallelization at both application-level and system-level and simultaneously execute different applications on different cores. Multi-core systems often implement an internal shared-resource structure to increase inter-core communication speeds. Examples of shared resources include the Translation lookaside Buffers (TLB), The Last Level Cache (LLC), the memory bus, and the general-purpose I/O peripheral. We exemplify a typical shared resource-structure in Figure 1.1.

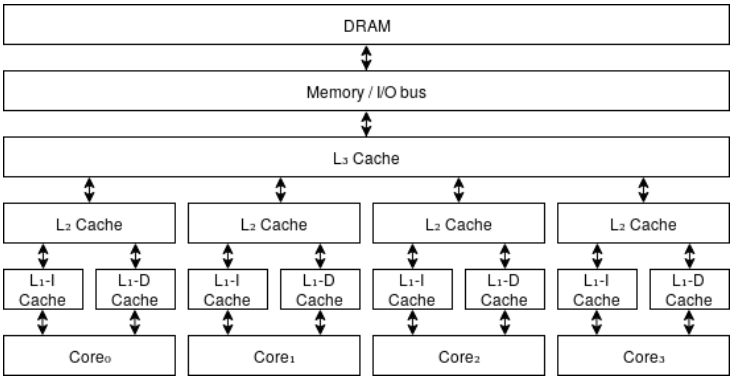

Figure 1.1: Example of a typical common 4-core system

Figure 1.1 exemplifies a modern computer's memory subsystem that contains multiple shared resources across different cores, such as the cache memories and the memory bus. Each core also shares resources logically between threads, such as the branch predictor unit (BPU) and floating-point unit (FPU). Processor manufacturers have since the multi-core paradigm's release followed the trend to add additional cores to the same chip. Intel®, for example, implements a 10-core architecture in their latest high-performance versions of the Comet-Lake processor. At the same time, Xilinx offers up to 4-core's in many of their chip solutions that uses an ARM architecture.

An increase in computational cores may seem tempting at first glance since the overall computational capacity increases. However, increasing the core count of a processor also increases the capacity requirements on the shared computing resources such as the LLC. Multi-core architectures allow applications executing on different cores to utilize the LLC simultaneously, wherefore a race-condition appears where multiple applications on different cores requests to use the LLC. This occurrence is called cache contention and causes severe execution-time fluctuations for applications. LLC contention is one of the major bottlenecks trying to bring multi-core processors into the time-critical domain [8].

This dissertation aims to improve the knowledge on resource contention in multi-core systems, caused by simultaneous resource usage from executing applications. We propose ad-hoc methods for measuring an application's resource usage and techniques to automatically determine the dependency between an application's performance and a specific hardware resource, which we denote as resource-boundness. We use resource-boundness calculations for determining partitioning methods that reduce resource contention in multi-core systems. Our final contribution includes run-time reconfiguration of a partitioned system to avoid wasting hardware resource space.

1.1 Scope of the thesis

This thesis focuses on homogeneous multi-core systems that utilize two or more cores to execute applications. We target the Last-level cache (LLC), an encompassing term for all caches located last in the memory hierarchy. The LLC is most commonly physically shared across a processor's different cores and links the connection between a processor's local cache and the memory bus and is especially prone to shared-resource contention since it is shared across different cores. We target two different architectures; ARM-cortex A53 [1] where the LLC is implemented as an L_2-cache and therefore stands as the second step in the internal memory hierarchy and Intel® Core™-i5 3570 [12]; where the LLC is implemented as an L_3-cache and therefore the stands as the third step in the memory hierarchy.

Our research mainly targets shared-resource contention, which happens due to the simultaneous use of multiple cores. We focus mainly on resource contention in the LLC, exemplified by an L_3-cache in Figure 1.1, but we also touch the topic of CPU contention.

1.2 Thesis outline

This thesis is composed of two parts. Part I, Chapters 1–6, describes the shared-resource contention problems and our research results. Part II contains the included papers, from Chapters 7–13.

Part I is organized as follows: Chapter 2 gives background information on multi-core computing and explains the origins of shared-resource contention. We also explain the memory subsystem of a computer and exemplify how to avoid shared-resource contention by isolating shared resources. Chapter 3 lists the research challenges, formalizes the challenges into research questions, and describes the methodology we have used to solve these challenges. Chapter 4 provides relevant related work, and Chapter 5 discusses our thesis contributions in detail with respect to the research challenges. Chapter 6 finalizes part I part of the thesis with discussions, conclusions, and directions for future work. Part II consists of the seven papers included in the thesis.

Chapter 2

Background

2.1 Internal memory subsystem of a computer

It is essential to understand how data requests travel through the memory hierarchy to grasp the resource contention problem. A data request starts with a load instruction in the CPU and must (if “unlucky") search through the internal memory hierarchy, the main memory and finally the hard drive to locate the requested data. The internal memory subsystem, in turn, is very complex and usually consists of several layers of caches and translation lookaside buffers (TLB’s). The sizes of the different memory units forms a pyramid. The least spacious memory (the register memory) locates at the top of the pyramid, while the most spacious memory (the hard drive) locates at the bottom.

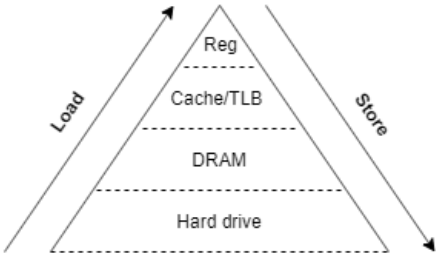

Figure 2.1: Model on the memory hierarchy

The chain of a computation always starts in the processor, where an operation uses one or more registers. The CPU holds a small set of general-purpose registers - modern 64-bit Intel® processors, for example, host a total of 16 different

general-purpose registers named R1-R16. The processor uses these registers to perform various operations such as load, store, addition, subtraction, jump, and comparison. Figure 2.1 illustrates the memory hierarchy starting with the registers on top and the hard drive at the bottom. The hard drive is most spacious, followed by the DRAM, caches are next, and the register memory is the least spacious.

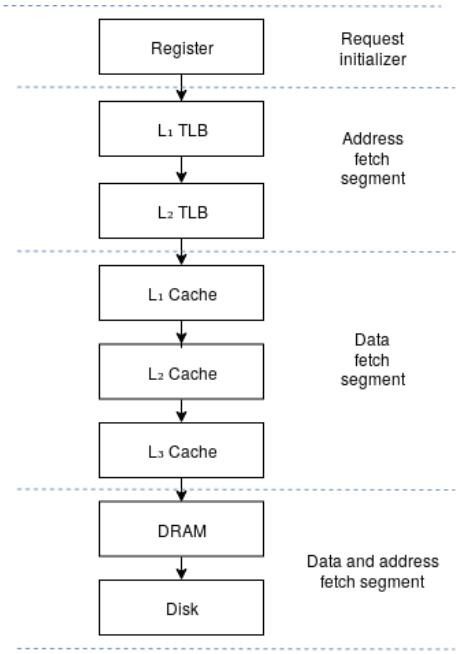

Figure 2.2: The entire memory call chain

Each memory operation contains a lookup procedure which checks if the requested data segment is available in the respective memory - see Figure 2.2. These memory lookups start in the caches/TLBs where caches store data and TLBs store address translation data. Data lookup failures in caches are called *cache misses* and require re-writes into the cache. Address translation lookup failures in TLBs are called *TLB misses* and require the requested address translation to be read from the page table in the physical memory. Lookups failed from the page table in the physical memory are called *page faults* and require the requested address to be read from disk. The following sections provide background information on address management and data management.

2.1.1 Address management

The Dynamic Random Access Memory (DRAM) divides addressable data into into *words*. The DRAM use a 32-bit or 64-bit wide word depending on the computer architecture. The program address space of modern computers is often significantly more spacious than the actual DRAM address space. Modern computers implement *virtual memory* which is a technique that uses the hard disk as a secondary memory address storing space. Using virtual memory reduces the possibility of system crashes due to insufficient physical memory. The virtual address space is significantly more spacious than the physical address space - a 64-bit system hosts a total of 2^{64} available entries in the virtual memory, while most physical memories in regular computers only host 2^{34} to 2^{35} entries in the physical memory. Since the virtual address space is significantly larger than the physical memory, there is no possibility to fit all virtual addresses in the physical address space. The virtual address of a word is therefore translated into a physical address. The translation information for a word is called a page and is stored in a list called the page table, see Figure 2.3. Entries in the page table are called page table entries.

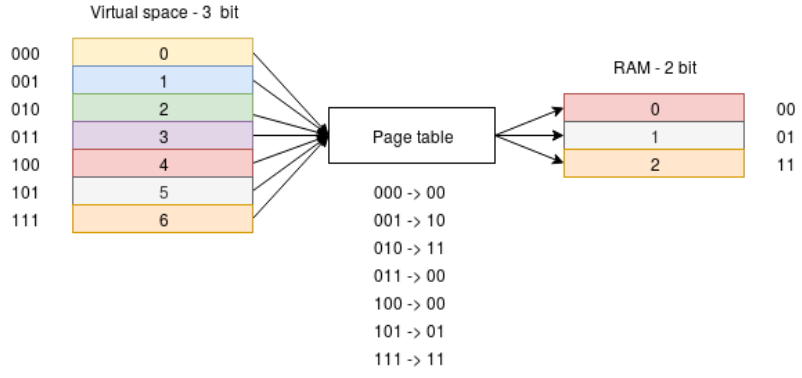

Figure 2.3: Illustration on page table mapping

The memory divides into pages. Each page has a size of 4 KB on most architectures, but the size may be configurable depending on system requirement. It is possible to use huge pages, which typically reside in the megabyte or gigabyte range to extend the map for very large memory regions. Virtual address requests trigger a lookup in the page table. If the requested virtual address is present in the physical memory, a *page hit* has occurred. If the requested virtual address is not present within the physical memory but instead is located on the disk, a *page miss* has occurred, the memory content is brought from the

disk. Page misses require new page table entries to be made for the requested address and cause substantial latency.

2.1.2 Translation lookaside buffer

The page table is relatively spacious, but also slow. The TLB is a small hardware-implemented buffer for storing page table entries and increases address translation latencies. The TLB often contains a small number of entries, such as 32 or 64 entries. The TLB is thus significantly less spacious than the page table, but also significantly faster – the TLB is essentially another cache used for address translations instead of data. TLB is often arranged into different types of TLB's for instruction and data - ITLB and DTLB which determines the page address space for instructions and data. Figure 2.4 depicts the lookup procedure and presents the respective penalties for each stage.

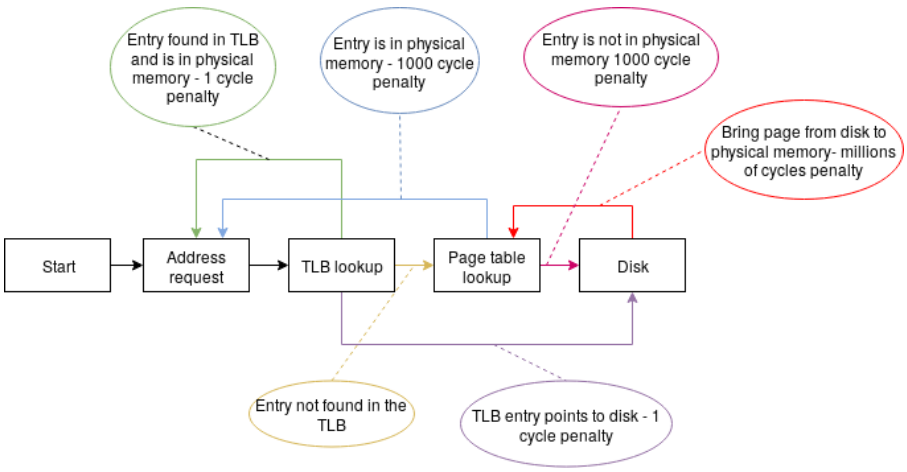

Figure 2.4: Virtual memory lookup procedure

The figure exemplifies an address request call-chain. The CPU instantiates an address lookup process for data in the first step. The address is returned directly if the requested address is present within the TLB (as whole, or as pointer to disk). The Page table is instead searched if the TLB's does not contain any information on the requested address. The disk is searched if the page table does not contain the requested address. Each step difference cycle penalties since it different amount of time to fetch addresses from the units. TLB's are fastest (1 cycle) and the disk is slowest (millions of cycles).

2.1.3 Registers

The register memory is the closest memory to the processor, but it is small: 32 or 64 bits per register depending on architecture. The registers are split into different subsets such as general-purpose registers, branch history registers, special-purpose registers, and clock registers. The special-purpose registers are pre-defined and are mainly used to interact with the hardware of the chip. The general-purpose registers are used to perform operations, also called instructions. Instructions include arithmetic-, logical-, comparison-, and memory operations. When executed, an instruction splits into smaller parts and is sent to the instruction pipeline. The pipeline executes these partial instructions in *stages*. We exemplify a classic RISC pipeline that uses 5-pipeline stages in figure 2.5, for demonstrative purposes.

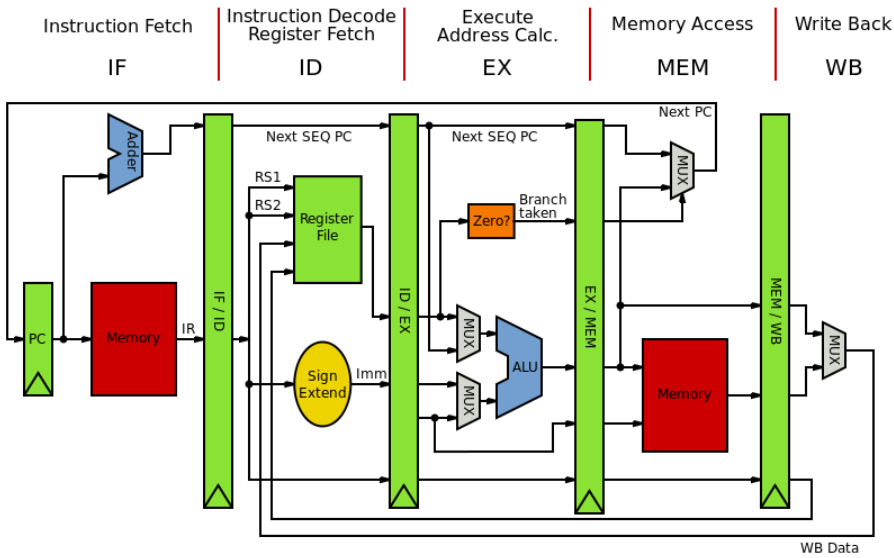

Figure 2.5: Classic 5-stage RISC pipeline [4]

A processor that utilizes a classic RISC-pipline executes an instruction in five different stages (IF, ID, EX, MEM, WB). An instruction that goes through the final stage in the processor pipeline is completed and marked as *retired*. The number of pipeline stages and the functionality of each pipeline stage varies greatly between different processors. Modern pipelines typically split an instruction into even more stages, including, e.g., *predecode* and instruction *queues*. For example, ARM Cortex-A53 uses a dual-issue, 8-stage pipeline [1] that and Intel® Core™ Ivy-Bridge uses a pipeline length of 14-18 stages [13].

2.1.4 Cache memories

In this thesis, we focus on the management of data used by processes executing in parallel. Therefore, we only discuss data caches and not instruction caches. In this section, we discuss the cache hierarchy from a three-layer cache perspective. This type of hierarchy is the most commonly occurring structure in Intel® desktop computers. Figure 2.6 shows the cache hierarchy and cache size of an Intel® Core™ i5-3570k processor.

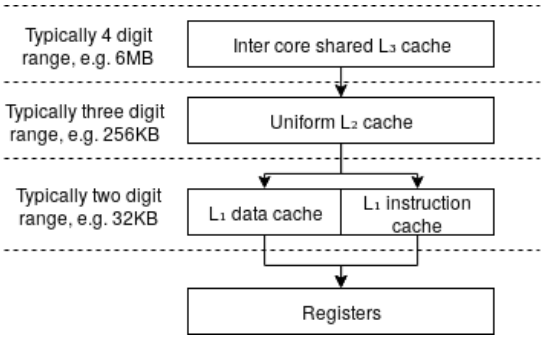

Figure 2.6: Intel® Core™ i5-3570k cache hierarchy

The cache is the second closest memory to the processor and is used to improve memory latencies of applications [7]. Modern Intel® chips often use several cache layers of different sizes. One of the most common designs includes two L_1-cache's (one L_1I-cache for instructions, and one L_1D-cache for data) and one unified (instruction + data cache) L_2-cache per core. The L_2-cache connects to one system-wide L_3-cache, that is shared by all cores in the system. The L_1-cache is closest to the processor and therefore fastest, but also the smallest. The L_2-cache is located further away from the processor and slower than the L_1-cache, but more spacious. The L_3-cache is furthest away from the processor and, therefore, slowest, but is also most spacious. Data requests from the processor start a lookup procedure in the cache, exemplified in a three-level cache system by Figure 2.7.

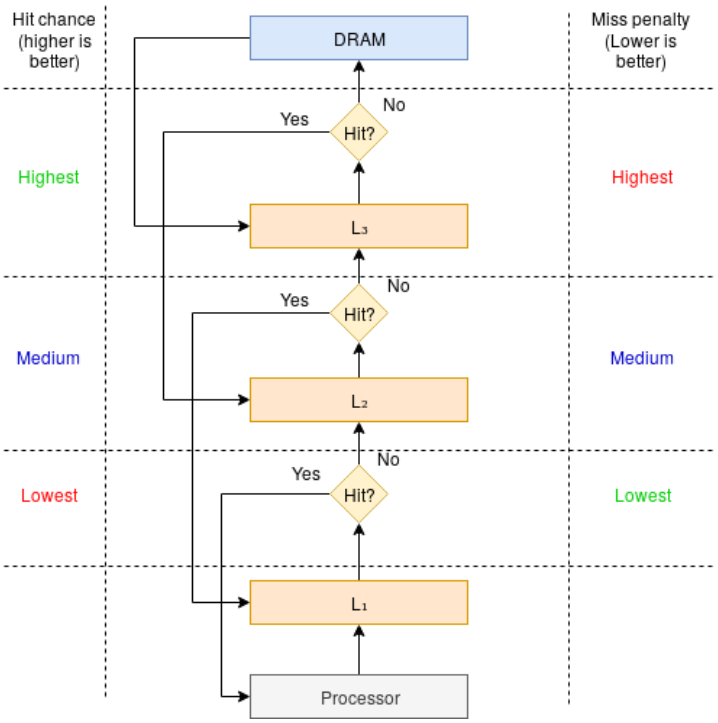

Figure 2.7: Three-level cache lookup procedure

The cache lookup procedure searches for the requested data within the sections of the cache memory, known as cache lines or cache blocks. Data found in the cache lines of the L_1-cache causes an L_1-cache hit and returns the data immediately to the processor for use. Data not found in the cache lines of the L_1-cache causes an L_1-cache miss and triggers a second lookup in the upper levels of the cache hierarchy to continue searching for the data. An L_1-cache hit causes no performance penalty and is, therefore, preferable to an L_1-cache miss, which causes extra latency, called *cache miss penalty*. The cache lookup procedure is common for all the cache levels. However, the latency caused by a cache miss depends on which cache level caused the miss. L_3-cache misses require data from the DRAM and therefore causes the highest latency. L_3-cache misses also require insertion of the DRAM data into the lower level L_2-cache and L_1D-cache. Misses in the L_1D-cache cause the lowest latency while the miss penalty of the L_2-cache falls in between. The L_3-cache has the highest hit chance since it is the most spacious cache, while the L_1D-cache has the lowest hit chance. The last level cache is also often shared between multiple cores, which presents interesting problems, described in section 2.4.1.

Cache memories are relatively small (usually ranging in the KB to single-digit MB sizes) compared to the DRAM. The small cache memory space means that the likelihood of experiencing only cache hits during application execution is small. The memory footprint of most applications is usually greater than the LLC. Since caches have only a limited memory space, the cache will inevitably, at some point, become full during application execution. Cache's implements memory replacement strategies such as least recently used (LRU) and first in first out (FIFO) to determine what data to replace when the cache is full. These strategies are called cache eviction policies. Eviction policies are one of the prime reasons for cache contention – a hazardous situation where different applications continuously steal cache memory from each other. Cache contention is further described in detail in section 2.4.1 while cache mechanisms for dealing with incoming data are described in section 2.1.4.1.

2.1.4.1 Cache data mechanisms

Cache lines are inserted into rows of the cache matrix, also called *sets*. The memory location placement for insertion of new cache lines into the cache depends on the *cache placement policy*. Caches are divided into three categories; direct-mapped, fully associative, and set associative. Cache's are structured into $n * m$ matrices where n represents the columns (cache ways) of the cache and m represents the cache rows (cache lines).

- Direct-mapped caches - the cache is organized into multiple sets, one set for each individual cache line, and can be represented as an $n * 1$ matrix.
- Fully associative caches - the cache is organized as one set, which contains all different cache lines, and can be represented by an $1 * m$ matrix.
- Set associative - a tradeoff between fully associative and direct mapped. The set-associative cache contains multiple sets, each set contains multiple cache lines and can be represented as an $n * m$ matrix.

These three policies have their respective advantages and disadvantages, and choosing one cache placement policy needs to be carefully thought out before committing to one policy. This section will only discuss set-associative caches since they are the most relevant to this thesis and also most commonly used in modern computers. The structure of a set-associative cache can be represented as an $n * m$ matrix, which means the entire set of cache lines is split between

different partitions of the cache. The smaller partitions are called *ways*. Figure 2.8 exemplifies the lookup procedure using a 2-way set-associative cache.

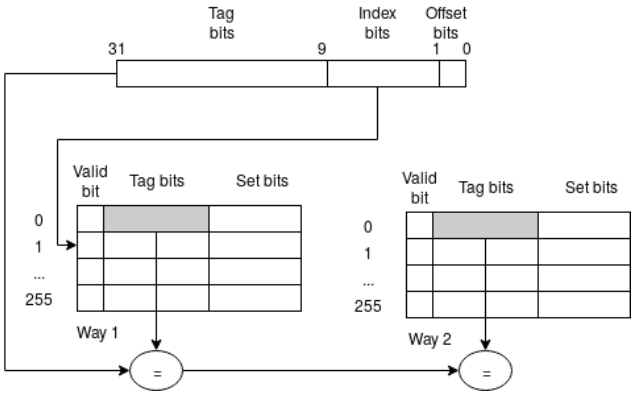

Figure 2.8: 2-way set-associative cache lookup example

The set-associativity of a cache creates a border between two ways. Incoming data will end up in either the first or second way of the cache. The set-associative bits determine the cache-way placement of data. If the set-associative bit is 0, data is placed in the first cache way; if the set-associative bit is 1, the data is placed in the second cache way. The index bits determine the row that stores the data. The final tag bits are used during cache lookups to match existing cache lines with new incoming data. If the tag bit of a new incoming data is equal to the tag bit of a cache line currently in the cache, a cache hit has occurred. The last valid bit is a final check to see if the cache line has been loaded with valid data yet. Set-associativity makes it possible to choose which cache line gets evicted in case the cache gets full. It furthermore enables isolation of tasks, further discussed in section 2.11, which can be accomplished by techniques such as page coloring.

2.2 Performance monitoring unit

It is possible to monitor system behavior using special-purpose registers. The performance monitoring unit (PMU) is responsible for sampling the hardware performance counters, a set of special-purpose registers built into processors. The performance counters are used for monitoring certain hardware events within the processor and do not cause any extra overhead when used. Modern PMUs support a vast set of events for different purposes, such as profiling for

system optimization. We exemplify some PMU events in Table 2.1 to give a brief idea of what can be measured using the PMU.

Table 2.1: Example of PMU events provided in the ARM cortex A-53 architecture

PMU event	Description
$L1D\_CACHE\_REFILL$	Counts L_1D-cache line replacements
$L2D\_CACHE\_REFILL$	Counts L_2-cache line replacements
BUS_ACCESS	Counts memory bus accesses
$L1D\_TLB\_REFILL$	Counts TLB replacements

Performance counters are originally per-core bound, meaning each performance counter only measures the events of its own designated core. It is, however, possible to insert trace functionalities to the PMU, which enables the measurement of process ID (PID) specific events rather than core-specific events. Tools such as perf [29] create PMU mappings for the Linux operating system and provide a more accessible interface for the usage of performance counters - compared to using assembly instructions to set up the PMU events. Other tools such as the Performance API (PAPI) [21] re-use the PMU mappings created by perf and enable an in-code usable API to trace performance counter events. Performance counters run ad-hoc of an application and increment for each occurred event until the application has finished executing or until the programmer sends a stop command to the register.

2.3 Application performance

It is possible to measure an application's performance in several ways, including total execution time, data processed per time interval (such as packets per second and frames per second), memory operations per time interval, etc. In this thesis, we view performance as the raw throughput of instructions that passed the final stage in the processor pipeline. Successfully executed instructions have passed all pipeline stages and are marked as retired once passing the final step. Therefore, an application should display a higher count of instructions retired in a time interval than a low count since more executed instructions per time interval typically mean the application finishes faster and, therefore, displays a better execution time. There are, however, exceptions to this rule since a high number of instructions does not necessarily mean that the application is executing useful work. For example, a network application that utilizes a busy-wait loop while waiting for packages to arrive will display

a high count of instructions retired while doing no valuable work. For such applications, it is not feasible to use instructions retired as a performance metric, but instead, other metrics such as packets per second should be used.

It is preferable for an application to exhibit a high count of instructions retired since it means the application will execute more instructions and complete its execution faster than if run with a low count of instructions retired in a time interval. Simple architectures that operate on one edge of a squared-wave clock can execute one instruction per cycle, assuming zero delays in the pipeline. Thus, in absolute ideal settings, an application will execute one instruction every cycle. Many things, such as memory faults and branch prediction faults can happen in modern computers that hinder the pipeline from executing one of the stages; such faults cause pipeline stalls and temporarily suspends the pipeline from executing.

Pipeline stalls occur when instructions or data are not yet available and can result from data dependencies, branch mispredictions, and memory not being available. The processor cannot execute instructions during pipeline stalls and thus has to wait until the stall has been resolved to execute instructions again. Pipeline stalls decrease the number of instructions possible to execute in a time interval. An application's pipeline stalls vary depending on the application functionality; sort algorithms typically contain many conditional statements that make them prone to branch mispredictions. Image processing algorithms contain lots of data transfers, making them prone to memory stalls. Applications with poorly structured code can suffer from structural data hazards. An application's performance thus builds a dependency towards such pipeline stalls, high count of stalls means less instructions are executed compared to the ideal case without pipeline stalls.

2.3.1 Resource-boundness

Several resources can cause pipeline stalls, such as the TLB's, the BPU, and caches, affecting an application's performance. An application's performance which decreases when the pipeline stalls increases, forms a dependency towards the resource that causes the pipeline stalls. The PMU offers capabilities to monitor most events that cause pipeline stalls and include events such as L_1-cache- and L_2-cache-misses, branch mispredictions, and TLB-misses.

We call the dependency between an application's performance and a specific hardware resource *resource-boundness* and illustrate an example of resource-boundness in Figure 2.9.

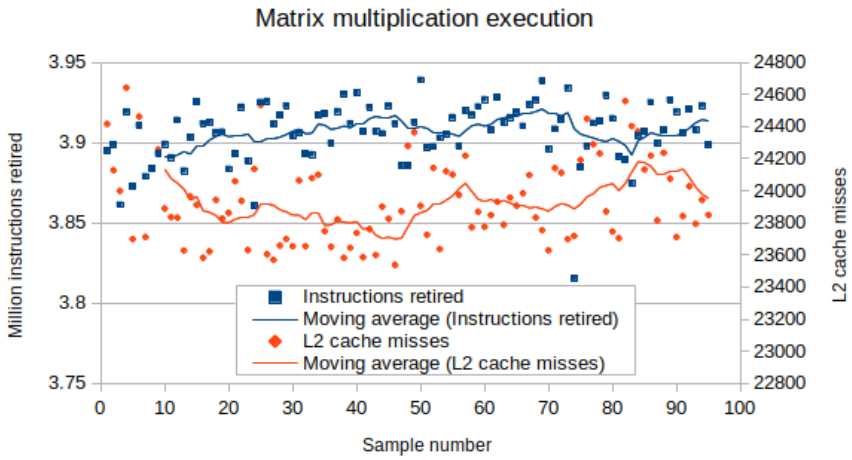

Figure 2.9: L_2-cache-bound matrix multiplication

Figure 2.9 plots the millions of instructions retired on the left-hand side y-axis (blue squares) and the number of L_2-cache-misses on the right-hand side y-axis (red rhombus) of a 200x200 matrix multiplication. We sample the performance counters instructions retired and L_2-cache misses every ten milliseconds until the matrix multiplication finishes its execution. The x-axis marks each sample number. We also plot the moving averages with a window size of 10 to emphasize the of the resource-boundness.

The performance of the matrix multiplication shows a dependency to the L_2-cache-misses where an increase in L_2-cache-misses leads to a notable decrease in instructions retired, and vice-versa, which in turn means the matrix multiplication is L_2-cache-bound. Similar dependencies can be investigated for all shared resources which have PMU-counters available to monitor. Applications that display a high resource-boundness towards one resource will perform significantly worse if the capacity/size of that hardware resource is reduced, or if that resource is simultaneously used by another core.

2.4 Resource sharing

Hardware resource sharing splits into three categories: CPU-, memory-, and I/O-sharing [31], all of which are subject to resource contention in multi-core systems. CPU-sharing is handled by scheduling techniques and/or virtualiza-

tion containers that control CPU utilization for specific tasks. I/O-sharing is also handled by virtualization techniques that create virtual I/O drivers which schedule I/O requests into software buffers to avoid contention problems that can occur. Memory sharing is the most complex mechanism in multi-core system since it spans over multiple units such as DRAM, TLB's and caches. In this thesis, we focus mainly on the consequences of cache sharing in multi-core systems. The following subsection explains cache sharing and its negative performance implications.

2.4.1 Memory sharing

Memory sharing is present in all modern multi-core systems due to the limited memory space of the internal memory units such as caches and TLB's. The shared L_3-cache is an excellent example of unpredictable performance in multi-core systems due to the cache replacement mechanisms. Figure 2.10 exemplifies a system suffering from cache contention due to the usage of multiple cores.

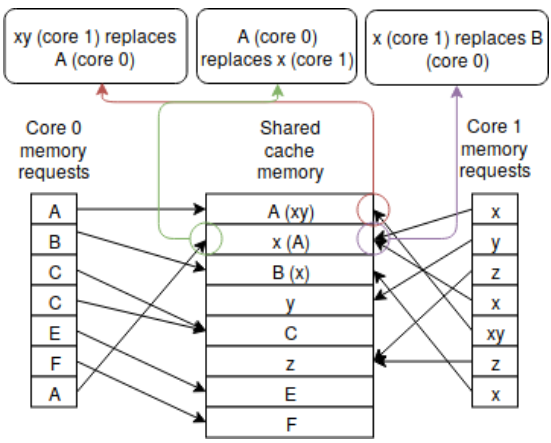

Figure 2.10: Example of LLC contention

The figure shows the memory requests of two applications app_0 and app_1, executing on core 0 and core 1. The applications are synchronized, which means app_0 is running just about before app_1. When the applications are synchronized in this manner, the shared cache memory starts storing the memory requests from each application in a sequential way. The cache memory requests start with request A from core 0, stored in the first cache line, continues with request x from Core 1, etc. Sharing the cache starts to become a problem once

the maximum cache capacity is reached. The maximum cache capacity in the example is eight cache lines, and the maximum capacity is reached once app_0 writes F to the cache. The next memory request, xy, from app_1 now will evict one of the existing cache lines to make room for the new xy request.

In the example, we have used LRU as eviction policy, which means A will be the evicted cache line. The next memory request from app_0 is A, which was recently replaced by the previous xy memory request from app_1. The A memory request will thus result in a cache miss and suffer from a cache miss penalty. This occurrence would not have happened if the xy memory request from app_1 did not evict that cache line - it would instead have resulted in a cache hit. The chain of replacements continues where app_0 and app_1 continuously replace the cache lines of each other, resulting in a behavior known as *thrashing*. Thrashing can be very hard to predict since it often occurs due to two workloads executing independently. Thrashing is not limited to only the cache but can also occur in the TLB or the page table.

2.5 Resource isolation

Hardware isolation using software techniques is a concept based on removing the resource sharing aspects of a multi-core system without altering the underlying hardware architecture, thus creating an isolated environment. Therefore, executing applications within an isolated domain should not affect applications in another isolated domain.

Complete isolation of entire systems can, however, become immensely complex due to a large number of hardware units within a computer, see Figure 2.11 (a system with two partitions). The figure shows a complex environment with many shared hardware units such as the DRAM, the caches, memory bus, I/O and TLB's. These units need to be put in different isolated domains in order to provide full isolation including different techniques such as TLB coloring [23], DRAM bank partitioning [37], memory bus bandwidth scheduling [38], I/O virtualization [27].

Section 2.1.1, 2.1.3 and 2.1.4 explain that the memory hierarchy call chain has different dependencies. The cache memory can only be utilized at maximum efficiency if there are minimal misses in the TLB because all data needs an address. The TLB can only be utilized at maximum efficiency if there are minimal misses in the Page Table. The resource contention in one certain hardware unit can happen due to resource contention in one of the higher memory hier-

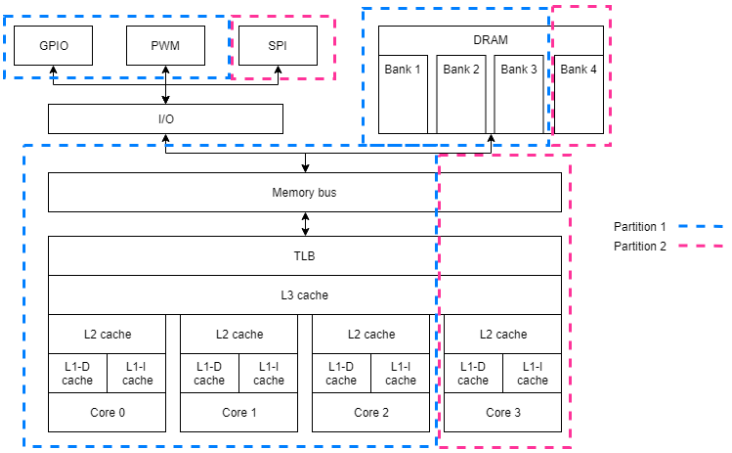

Figure 2.11: Example of a completely partitioned system

archies. Therefore, there is a risk that indications of resource contention in a certain hardware unit can be falsely reported.

2.5.1 Cache coloring – an example of an isolation technique

We exemplify cache isolation in Figure 2.12, through cache coloring.

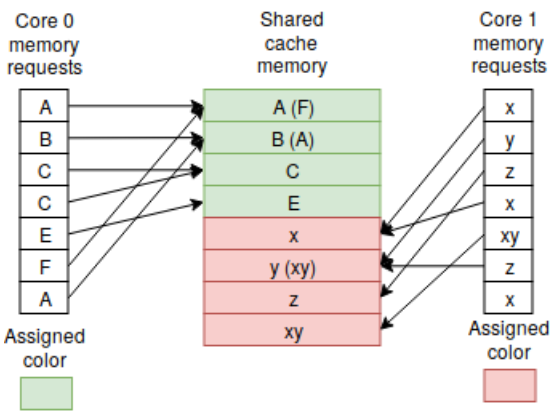

Figure 2.12: Thrashing avoided by cache coloring

The idea behind cache coloring is to remove the "shared" aspects of the cache through software techniques that alter how memory is mapped from the DRAM to the cache. Memory requests from different applications are assigned specific colors corresponding to specific memory regions within the

cache. Enforcing the cache coloring methodology disables applications on different cores from using each other's cache lines. The only cache evictions which occur now will happen due the application that "owns" the colour, as can be seen on core 1, where xy replaces the y cache line. Since the cache coloring methodology disables applications from using each other's memory, the cache is now *isolated*.

We compare the effects of cache partitioning to a non-partitioned system in Figure. 2.13. The figure shows 150 executions of a 200x200 matrix multiplication that runs on core 0. The y-axis plots the execution times, and the x-axis marks one matrix multiplication execution. We start another matrix multiplication on core 1 at execution 75.

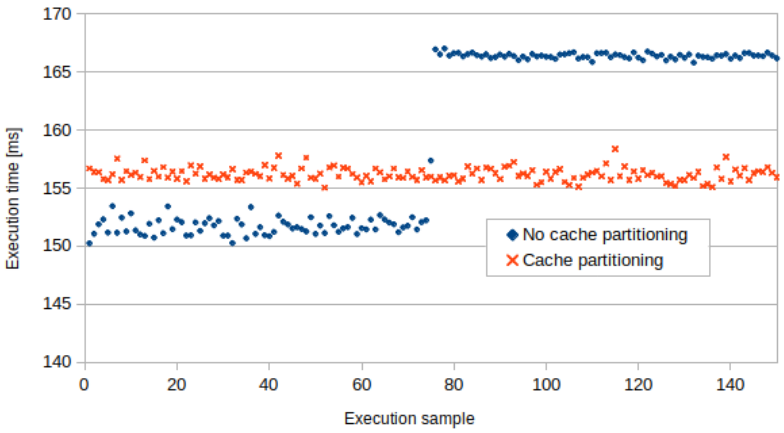

Figure 2.13: Cache coloring

The figure further shows a matrix multiplication running under two system setups; unpartitioned Linux (blue) and cache partitioned Linux (orange). We call the matrix multiplication that runs in a non-partitioned system $matmult_1$ and the matrix multiplication that runs within a partitioned system $matmult_2$. $Matmult_1$ is significantly affected when the second matrix multiplication is started on another core. $Matmult_2$ presents a slightly decreased performance compared to $matmult_1$ for the first 75 iterations, which is an effect of the cache partitioning overhead. The main benefit of partitioning is shown on iteration 75 when starting another matrix multiplication on another core, where $matmult_2$ displays no performance degradations.

Chapter 3

Research Overview

3.1 Problem formulation

Avoiding shared-resource contention is important for time-critical systems, since it negatively impacts an application's performance and predictability. Unfortunately, shared-resource contention is also difficult to detect – system engineers have to create stand-alone testing suites that overload a shared resource's capacity to gain knowledge on which applications may experience performance degradation due to resource contention. This process cannot be done during system run-time and adds additional overhead to system testing. Finding resource-contention and managing system partitions in a multi-core system is thus a system engineer's task. This thesis aims towards migrating this task to an automated process managed by an operating system. We formalize this thesis overall goal as follows.

Thesis goal: An automated resource-contention mitigation process.

Two significant challenges that arise from our overall goal are; i) identification of shared-resource contention and; ii) appropiate allocation of resources using isolation techniques. We discuss the two challenges in the following subsections.

3.1.1 Identification of resource contention

There are techniques such as cache partitioning that solve resource contention in the cache memory but they often come at a cost of execution-time overhead [5] [6] [37]. It is therefore important to discuss when and how to use such isolation techniques. In this thesis we employ a cache partitioning method called page coloring which is suitable for removing resource contention, but can on the other hand dramatically reduce an application's performance if assigned inappropriately. Therefore, it is essential to assign cache partitions only to applications that display a boundness towards the LLC, in order to avoid waste of resource space. Hence, the first research challenge that we face is the *identification* of resource-boundness. Identification of resource-boundness is critical to multi-core computations since it quantifies the risk of an application suffering from shared-resource contention. An exact measurement of an application's resource-usage is, however, time-consuming to provide, due to code and hardware complexity. System engineers need to carefully analyze an application's hardware interactions within the code to create a resource-boundness model, which can become a time-consuming process. We formalize research challenge 1 as follows:

Research challenge 1 (RC1): Automatic identification of shared-resource contention and resource-boundness.

3.1.2 Resource management

There are finite hardware resource capacities in a computer. Therefore it becomes an interesting challenge to determine how to allocate partition space for applications running in a system. Allocating too small amount of resources may lead to unacceptable application performance degradation. In contrast, allocating too much resources for an application may lead to depletion of resources and degraded performance for other applications (without any significant gain for the application that have been overprovisioned with resources). We must consider each application's resource-boundness to allocate the correct amount of resources to an application and furthermore gain an understanding on how a change in resource allocation will affect the application's performance. We formalize research challenge 2 as follows:

Research challenge 2 (RC2): Appropiate allocation of resources using isolation techniques.

3.2 Research methodology

This thesis addresses the research challenges using empirical studies. The empirical studies are concentrated on executing industrial workloads on a traditional operating system for embedded applications. We have used a set of feature detection algorithms which are image processing algorithms, common in obstacle localization and avoidance domains such as avionics and autonomous vehicles. We also use synthetic workloads, that occupy a particular resource. We categorize our studies into two types, exploratory and implementation. We list our papers and the corresponding type for each paper as follows:

Publication	**RC**	**Type**
Paper A	1	Exploratory
Paper B	1	Exploratory
Paper C	1 & 2	Implementation
Paper D	2	Exploratory
Paper E	2	Implementation
Paper F	2	Implementation
Paper G	1	Exploratory

Table 3.1: The contribution of the individual papers to the research sub-goals

In our exploratory studies, the main research focus is investigating the effects of resource-contention in multi-core systems. We first execute application tests on one core, without any deliberately disturbing loads on other cores. The single-core execution of an application is our reference point for comparison and we call this *baseline* execution. Once we have established a baseline execution of an application, we execute the application in a multi-core environment. the multi-core environment also runs other applications on different cores. The outcome from our exploratory studies are conclusions on why applications behave in a certain way in a multi-core setting.

Our implementation studies proposes methods to counter resource-contention behavior detected in our exploratory studies. We compare our solutions to existing techniques and discuss positives and drawbacks. The outcome from our implementation studies are application binaries written in C/C++ for Linux.

3.3 Research approach

We use a methodology based on empirical case studies on memory-intensive applications combined with theoretical reasoning. Figure 3.1 provides an overview of our research process. The research steps are listed as follows:

1. Identification of the research problem and establishing an overall research goal of the thesis. This step also includes state of the art research.
2. Dividing the research problem into smaller and more manageable research challenges.
3. Categorize the research challenges into thesis contributions which more clearly defines what problem is solved.
4. Perform case study on application run-time behavior.
5. Perform solution suggestions.
6. Evaluate solution suggestions.

Figure 3.1: Research methodology

Step 1 provides an overall goal for the thesis, while step 2 splits the overall goal into research challenges. Our iterative investigation process starts from

step 3, where we target one research challenge and work towards solving that research challenge. In step 4, we conduct our exploratory research in forms of case studies. The case studies produces a deeper understanding on application behavior in mutli-core systems. In step 5 we propose implementation solutions to manage resource isolation techniques. In step 6, we evaluate the usefulness of our proposed solutions.

3.4 Delimitations

This thesis work was conducted closely with representatives from several companies that provided input to our case studies. The industrial input included use-cases that we evaluate to ensure that they have an industrial relevance. Our tests execute in a Linux environment on CoTS hardware which means we do not simulate an environment for our tests. The main strength of this approach is that the tests for a specific platform become realistic since we operate in an environment that contains real hardware and all software events that are tied to this hardware. The main weakness of this approach is that the test results are specialized to that particular platform. We design our control and analysis methods to be generalizable for different platforms, but hardware with different cache properties will display a difference in cache utilization. E.g., an application that displays a strong cache boundness on one platform could potentially display a weaker boundness on another platform with a more spacious LLC.

We use the performance metric instruction retired, which means we must limit the scope of our test applications to only include applications that does not utilize busy-wait loops. We chose this approach since it enables us to automatically sample an application's performance without having to modify the code. This approach, however, means we must limit the scope of the applications that benefits from a high number of instructions retired. I/O applications are often stuck in tight busy-wait loops while waiting for the I/O resource to become ready and therefore display a display a high number of instructions retired while doing little useful work.

Our final limitation of this thesis is the usage of isolation methods. We investigate two isolation methods, Jailhouse hypervisor (For isolation of CPU and local resources) and Palloc (For LLC partitioning). Other isolation options such as TLB coloring, DRAM-bank partitioning, memory bus partitioning exists, but we wanted to gain a more profound knowledge in a specific contention area (the LLC) and control resource allocation in that area.

Chapter 4

Related work

In our approach, we divide a multi-core system into three critical parts which are at risk due to resource contention: the CPU, the internal memory components, and the I/O units. In this thesis we placed our focus on investigating internal memory.

While there exist a rich body of research looking into the isolation of shared hardware resources that affect the execution time predictability of applications, the following subsections discuss related work aspects that are in line with our research goals and chosen approach.

4.1 Resource-boundness

Resource dependency analysis splits into two subcategories; tools and methodologies. Tools are fully-fledged automatic processes that take an application as input, utilizes the performance counters to investigate the application's runtime characteristics, and then provide an output in terms of application behavior. In this subsection, we discuss fully-fledged tools that determine an application's resource-boundness using performance counters.

Scarphase [10] is a PMU-based tool that is closely related to our research on the boundness topic and presents methods for investigating performance counters in detail. Sembrandt et al. [25] [26] discuss how to split applications into phases according to their resource usage. The works present how to distinguish resource dominant parts of an application such as cache-heavy parts and branch-prediction unit (BPU) heavy parts. The research related to the

Scarphase tool shows how an application utilizes computer resources differently during run-time and can therefore be used to investigate contention scenarios. Scarphase only investigates the resource usage and does not account for how an application's performance relates to that specific resource usage.

Charmon [14] is another PMU-based tool that investigates how resource usage relates to application performance. Charmon correlates a performance metric (such as packets per second) with a performance counter. The higher correlation between the performance metric and the performance counter event means a higher resource-boundness since the performance depends on that particular hardware resource. The difference between Charmon and our work is the definition of performance, where Charmon views performance as something which needs to be user-specified depending on application functionality. We use a generic metric rather than an application specific which allow our method to to analyze legacy software binaries without any annotations of the source code. Our approach, however, limits scope of applications to non-I/O-bound applications.

Hua et al.[11] propose a scheme utilizing curve fitting to build a model on an application's resource-dependency. The paper uses instructions per cycle as performance metric and segments thread execution and into smaller segment (similar to [25]) slices. The paper creates models on how the performance varies with the size of a determined resource (in the paper, the variable resource is described as the register rename file). The main difference between our approach in our tool and the previously presented tools is the plug-and-play nature of our tool. Our tool can monitor and determine the resource-boundness without modifying an application's source code. Our approach runs cross-platform directly on hardware and does not require simulated resources to estimate an application's resource-boundness.

4.1.1 Understanding cache contention

Our work mainly targets cache contention as the source for performance degradation. Related methodologies to display and verify cache contention are relevant to us as they provide methodologies that enforce cache contention and thus degrade applications' performance. Cache pirating [8] and Bandwidth bandit [9] present methods to pollute the cache and the memory bus with an excessive amount of data and, thus, force an application into a resource contention state. Cache pirating and Bandwidth Bandit run one application on one core while running another "resource hungry" load on another core, creating a disadvantageous execution environment for the first application. The two

methodologies are essentially benchmarks on how poorly an application will execute in the absolute worst-case conditions.

Sandberg et al. [24] utilizes cache pirating and proposes a model utilizing the phase concept for determining when multiple co-executing applications risk cache contention. The models rely on run-time data such as cache usage from applications that execute. The authors then create models that estimate the performance degradation of an application when running it simultaneously with another application. The model provides an accurate estimate of an application's performance degradation due to contention but performs these calculations offline due to their reliance on run-time application data. Our approach does not provide a metric for an application's performance degradation but instead serves as an indicator for cache contentious scenarios. Our approach using a performance counter can, however, be used online and does not require pre-sampling.

Chandra et al. [2] present three different models, including frequency of access, stack distance, and inductive probability for predicting the severity of cache contention. Subramanian et al. [28] proposes the application slowdown model (ASM), which utilizes the cache access ratios of an application's phases. It demonstrates the model's ability to predict application slowdown due to both bandwidth and cache partitioning. Xu et al.[32] propose a shared cache performance prediction model using histograms of an application's cache re-usage distance, citing a lack of performance (in terms of overhead) and throughput from previous solutions. Zhao et al. [39] presents an approach to detect cache contention by examining the shadow memory. The paper combines the trace from both cache misses and cache invalidates to determine if contention exists. The solution is based on running threads and does determine the number of cache contentions between threads.

The main difference between our work and above-mentioned studies is the utilization of resource-boundness as an indicator for inbound cache contention scenarios. The approach is less accurate compared to prediction models [2, 24, 28, 32] since we do not consider aspects such as re-use distance or cache invalidates. Resource-boundness is determined online during the execution of an application, can therefore be applied directly towards running systems without pre-processing steps. We argue that our approach is better to use in large-scale systems where the number of permutations of co-executing tasks makes offline measurements of the complete system too time-consuming. Our approach requires an application to run, and we can continuously build the application's resource-boundness profile during runtime. If the application stops executing, we store the application's resource-boundness value in a database

that is available for decision making (to partition or not to partition) the next time this application is scheduled for execution.

4.1.2 Utilization of isolation techniques

Mittal et al. [18] present an extensive survey on different cache partitioning methods for multi-core processors. The paper categorizes cache partitioning research papers based on the optimization goal of each paper, i.e., what are the algorithm proposals trying to accomplish. Mittal lists five different optimization domains, including system fairness, QoS or priority, static energy/dynamic energy, and power capping. QoS or priority is related to the work presented in this thesis.

Kasture et al. [15] present Ubik, an approach designed to guarantee tail latencies of latency-critical applications. The central concept of Ubik is exploiting the scheduling of tasks. Re-assessment of cache-memory size is done whenever latency-critical applications are scheduled to execute. The cache memory is redistributed to other applications once the latency-critical application goes back to an idle state.

Moreto et al. [20] propose flexDCP, a bridge for operating systems that tries to convert user-specified quality of service requirements to resource allocation using performance counters of currently running applications. The flexDCP methodology compares the current application performance versus the maximum achievable performance (i.e., when all cache partitions are assigned to the application). FlexDCP then builds a model that estimates QoS changes that may occur due to changing cache partition sizes. FlexDCP then allocates cache partitions according to the model with the user's QoS requirements in mind.

Mittal et al. [19] present the MANAGER framework which provides a QoS guarantee that an application running in the MANAGER will not operate under 22% of its maximum possible performance when using cache partitioning. MANAGER estimates an application's execution time loss due to cache misses during run-time and assumes that all memory cycle stalls vary linearly with the number of cache load misses. MANAGER provides a method for assessing how changing the cache partition size will affect the application performance since a decrease in cache partition size typically leads to more cache misses.

What is common to the quality of service papers listed above is the notion of QoS as user-defined and requires the user to know the specifications of what the application is supposed to produce, in order to provide a quantitative

measure on how well the application is performing.

In our work, we take a different approach and generalize the QoS of all non-I/O-bound applications to be the number of instructions retired produced in a specific time interval. We utilize performance counters and trace the number of instructions retired of individual process ID's which means we can make cache partition assignments without knowing what the application is supposed to execute.

Online partitioning provides solutions for cache partition allocations to an already running system. On the other hand, offline partitioning relies on knowledge of a system's executing tasks and can therefore perform exhaustive searches to find the optimal cache partition allocations based on all task permutations.

Our work utilizes the online, combined DRAM-bank and cache tool called PALLOC [37] for partitioning the cache memory. We aim to expand the usage of PALLOC by implementing control mechanisms for load-balancing applications using the cache memory, while PALLOC mainly focuses on the DRAM-bank perspective. Other related works include the Coloris cache partition scheme [36] which re-partitions cache memory for applications running on different cores by continuous monitoring the cache accesses performance counter. Pan and Pai [22] suggest another approach using cache-way partitioning using histograms of the cache re-use distance, similar to the approach proposed in [32]. However, the solution is limited to cache-way partitioning and relies on the hardware vendor to provide the tools for cache partitioning.

Other recent works by Xu et al. [34] embrace the new cache allocation technology (CAT) provided by Intel. CAT divides the cache into N cache partitions and assigns these to different cores using a bitmask. Xu presents the tool called vCAT, which enables cache partitioning through CAT to span over multiple cores in a virtualized environment to improve the WCET of tasks running in a virtualized environment. The main downside of the vCAT approach is the dependency on Intel® chips. vCAT can only work using Intel® chips, whereas ARM chips are often used in embedded systems.

Kloda et al. [16] implements the page-coloring algorithm for the Jailhouse hypervisor. Their page coloring algorithm allocates cache memory to improve isolation between Jailhouse guest operating systems (cells). The downside of Kloda's approach is the static nature of its implementation. Portions of the cache memory are assigned to the guest operating systems at hypervisor boot. Therefore, it is not possible to tune the cache partition sizes without a reboot of the hypervisor. In contrast to vCAT and Jailhouse page coloring, we uti-

lize PALLOC (page-coloring), which is cross-platform. PALLOC runs the Cgroup interface and is the best fit to run on lightweight hypervisors such as QEMU/Docker. In addition, our utilization of PALLOC enables us to automatically re-tune the cache memory for individual processes during runtime. Therefore, our approach has more functionality, than just assigning partitions according to a guest operating system. At the same time, it also operates cross-platform, due to the PALLOC interface.

Xu et al. [33] also present the work CaM, which is an extension of vCAT that also includes memory bus partitioning. CaM proposes an algorithm that contains multiple procedures that optimize task schedulability in a system. The main similarities between our work and CaM lie in resource allocation and load balancing. CaM takes the approach of allocating the minimum partition size to tasks executing on different cores and then re-allocates partitions until all tasks are schedulable. CaM also executes load balancing by migrating tasks from unschedulable cores to schedulable cores while providing WCET guarantees. Our work, instead, focuses on tweaking the performance of specific applications in an online fashion. Our controller does not evaluate all possible task permutations in a system, but focuses on tweaking the cache partition size of already running applications to satisfy performance needs.

4.2 Performance evaluation

Measuring performance degradation and performance isolation is important to our research. It enables us to quantify the amount of isolation given by an isolation technique and measure the effectiveness of the isolation technique. Perhaps the most common approach is measuring the slowdown ratio given between a baseline environment system and its extended counterpart [3, 17, 30, 35]. The baseline measurement value is often represented by the performance of an application running in a native operating system environment, which in most cases is Linux. The extended counterpart measurement value is often represented by the performance of an application running in a Linux system with isolation extensions, such as virtual machines or cache partitioning algorithms.

Our opinion is that the application performance metric is highly dependent on the purpose of the application. Therefore, measuring the performance of an application will differ depending on the application's purpose. For example, it is not sufficient to use execution time as a TCP/IP stack algorithm performance metric since the throughput will be partly dependent on waiting for data from

other units. For this reason, we argue that it is important to derive the root cause of performance degradation in applications rather than looking at the execution time as the prime measurement for performance. Countless studies evaluate how to increase performance through parallelisms, measuring system performance effects from a newly implemented algorithm, measuring performance degradation of virtual machines, and many more. In fact, the entire field of computer science is imbued with performance measurements; whenever a new application is implemented, the programmer will want to know the application's performance and how it fits the system specifics.

Our approach to measure an application's performance using instructions retired in a time-base is not new, but our with our methods simplifies the process of measuring application performance. We use a separate thread to continuously monitor the number of instructions retired of one application and therefore get quick feedback on the application's current performance. Our plug-and-play methodology for measuring an application's can be attached to an already running application, it does not require modification of the application's code. The performance measurements are also valuable when combined with measurements of other performance counters to estimate the application's resource-boundness.

Chapter 5

Thesis contributions

This thesis aims to move the responsibility of finding and mitigating shared resource contention from a system engineer to an automated process that is manageable by the system itself. Our primary resulting artefact is the LLM-shark tool, an application that monitors application performance ad-hoc, characterizes application resource-boundness, suggests sizes of cache partition containers, and, finally, re-partitions the cache to follow application requirements. We present a model of LLM-shark architecture in Figure 5.1.

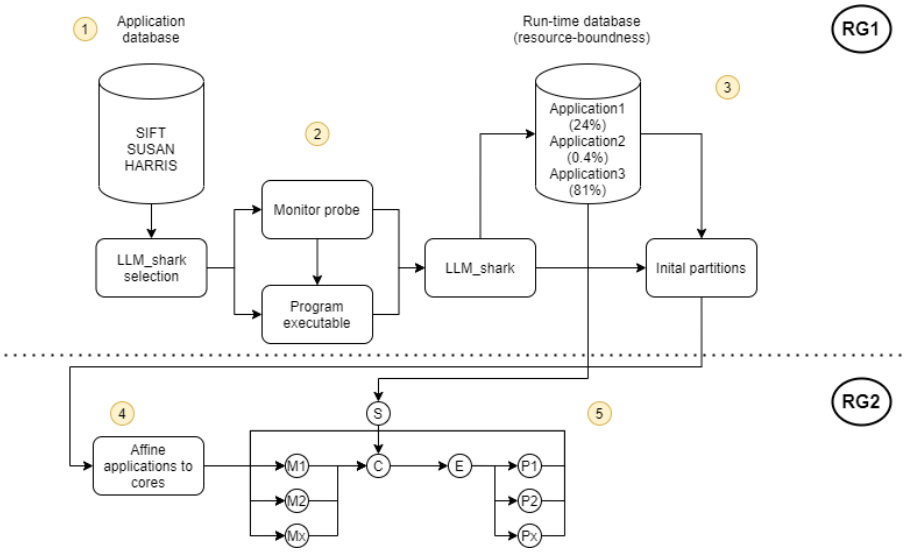

Figure 5.1: Holistic overview of LLM-shark

Figure 5.1 simplifies the LLM-shark architecture into 5 different parts, listed as follows:

1. **Selection step** – The database contains a list of applications that should run under the umbrella of LLM-shark. A system engineer must further specify the performance counters that are of interest in the LLM-shark selection step.

2. **Measurement step** – LLM-shark will run an application on one core and a performance-counter probe on another core that monitors the application's performance counter events. This procedure is done offline in our papers but can also be done online. However, online measurements can become less accurate as resource contention might exist in the system, and as such, we then measure the resource-boundness of an already resource contented load.

3. **Resource-boundness** – LLM-shark will automatically calculate the resource-boundness of all applications run in the previous monitoring step using correlation. LLM-shark stores resource-boundness values in a run-time database. We use the resource values as a basis for partitioning setup.

4. **Program start** – LLM-shark consults the run-time database to formulate initial program partitions, including each program's core affinity to avoid scheduling contention and cache partitioning to avoid cache contention. Then, LLM-shark starts the specified applications on different cores and actuates an initial cache partitioning setup.

5. **Dynamic adaptation** – LLM-shark runs each program in cache partitioned containers, in a controller-like fashion. An application has a setpoint performance (which is gained from the run-time database) and the tool tries to satisfy the application's performance setpoint using a priority-based allocation policy.

Based on the holistic view of LLM-shark, we present four thesis contributions that solve our research challenges, as follows.

1. Thesis contribution 1 **(TC1)** – Methodology for ad-hoc measurement of application's performance and run-time characteristics.

2. Thesis contribution 2 **(TC2)** – Automatic resource-boundness determination.

3. Thesis contribution 3 **(TC3)** – Methods for measuring the degree of resource-isolation in a system.

4. Thesis contribution 4 **(TC4)** – Dynamic allocation of cache memory.

Our overall research goal presents two research challenges, described in Chapter 3. We address the research challenges by thesis contributions, which, in turn, each partially contributes to solve the overall goal. Figure 5.2 depicts the connection between the overall goal, research challenges, and thesis contributions, and Figure 5.3 depicts the mapping between the included papers and our thesis contributions.

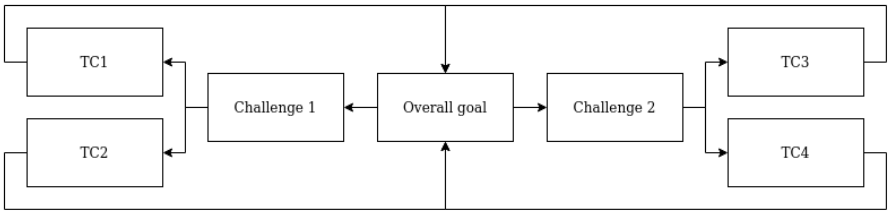

Figure 5.2: Mapping between thesis contributions and research goals

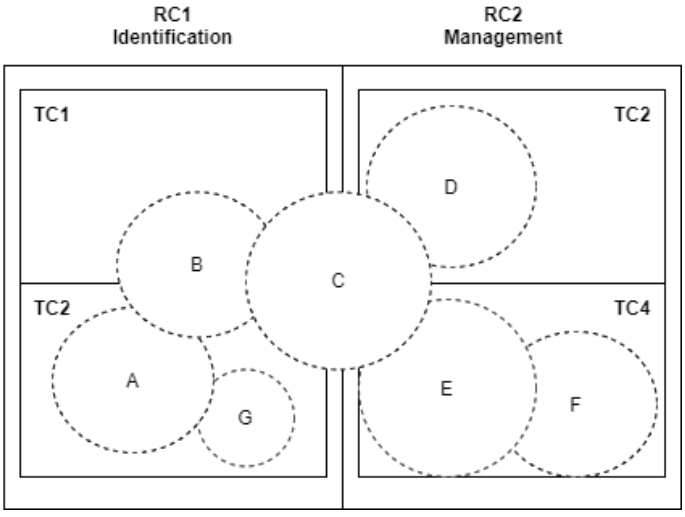

Figure 5.3: Mapping between thesis contributions and research goals via published articles

We elaborate more on the thesis contributions in the following subsections.

5.1 TC1 – Ad-hoc monitoring of performance

Continuous monitoring of application performance can become tedious and often requires modification of the source code. The programmer must 1) understand the application code, 2) insert time measurement probes around appropriate code parts and, 3) generate an understandable output. Steps 2) and 3) require modification of the source code. Step 3) is furthermore essential to structure according to specific output formatting to be understandable by computers.

This thesis contribution presents an ad-hoc way of monitoring an application's performance that does not require any of the modification of the source code. We utilize the performance counter event instructions retired as performance metric, instead of traditional execution time measurement.

We implement a performance-counter probe that samples the number of instructions retired of application's PID. Our probe utilizes the PAPI interface, can be attached to any user-space program and does not require code modification of the application. We utilize this approach for ad-hoc performance measurements in Paper B, C, and E. Paper B's primary goal is to investigate methods to determine an application's resource-boundness automatically. We use the instructions retired to automatically identify the application's different execution phases. We experiment with different methods using the application's instructions retired to determine when an application started executing a different phase. Figure 5.4 exemplifies the performance counter output of Harris' algorithm using our measurement approach. Purple marks the instructions retired on the left-hand side y-axis while green marks the L_2-cache misses on the right-hand side y-axis. We distinguish each phase with a vertical black line inside the graph.

The reason for detecting such phases is to automatically investigate the resource boundness of individual code segments rather than that of the entire execution. We discuss automatic resource-boundness detection in detail in the following subsection. Our performance measurement approach applies to applications that are not utilizing busy-wait loops. In paper E, we demonstrate the usability of this performance measurement approach using an online cache partitioning allocator that continuously monitors an application's performance and allocates cache partitions according to the performance.

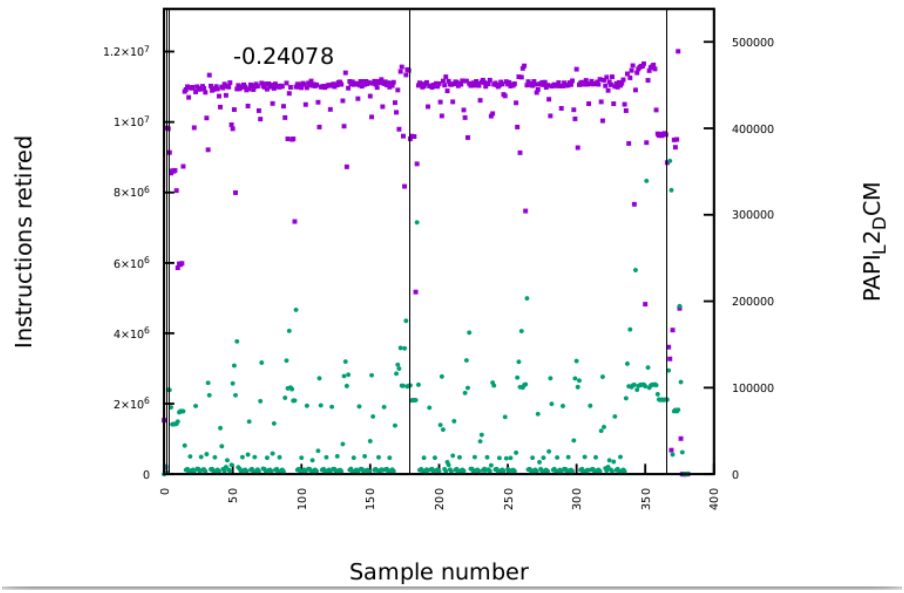

Figure 5.4: Harris feature detection

The main benefit of our approach is that application performance can be monitored ad-hoc and does not require any modifications to the code. The main drawback is the requirement that it puts on an application's functionality – the application must not use busy-wait loops. Paper B and C propose and use the measurement approach towards TC1, while Paper E utilizes the proposed performance concept in a running system.

5.2 TC2 – Automatic resource-boundness determination

In this contribution, we present our approach to determine an application's resource-boundness automatically. We utilize our sampling-based measurement approach from TC1 to monitor the performance and the usage of other resources in a processor. Our theory is that an application heavily dependent on L_2-cache for execution will display significant performance degradation when the cache is full and starts to evict cache lines, ultimately leading to cache misses for new incoming instructions. Thus, an application that binds to the L_2-cache will display a performance increase when the L_2-cache

misses decreases and a performance degradation when the L_2-cache misses increases. Using correlation between the two datasets (performance and a resource counter-event) we can quantify the magnitude of this relationship. There exists three types of relationships in a correlation; negative correlation, where the values of two datasets moves in opposite directions; positive correlation, where the values of two datasets moves in the same direction; and no correlation, where the values of the two datasets shows no trends on affecting each other. We exemplify the correlation types in Figure 5.5 (positive correlation), Figure 5.6 (negative correlation) and Figure 5.7 (no correlation).

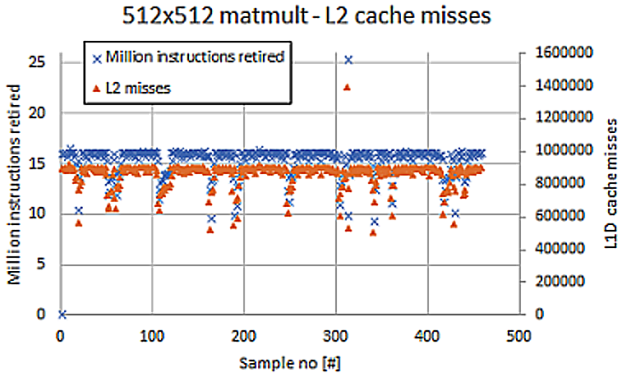

Figure 5.5: Positive correlation

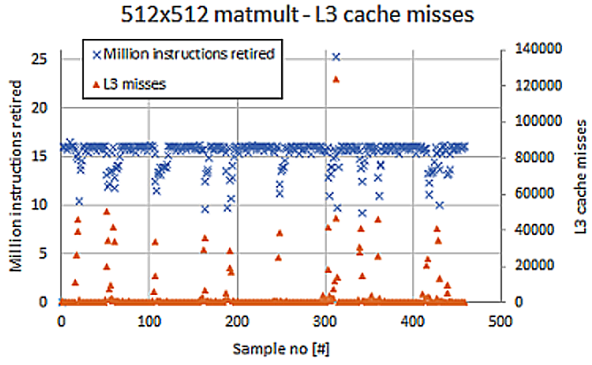

Figure 5.6: Negative correlation

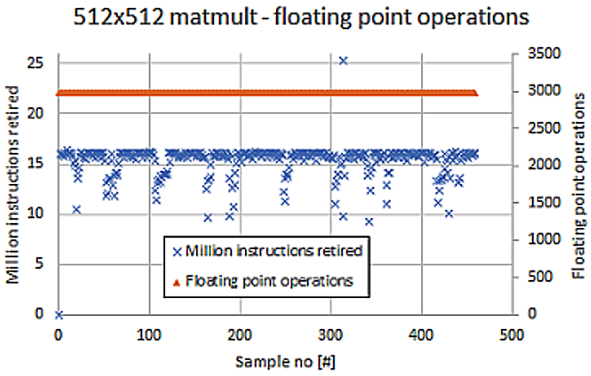

Figure 5.7: No correlation

The figures shows run-time measurements of one execution of a 512x512 matrix multiplication executing on a quad-core Intel system that implements a L_3-cache as the LLC. The blue crosses plot the number of instructions retired on the left-hand side y-axis. The orange triangle marks the performance counter-event on the right-hand side y-axis (L_2-cache-misses in Figure 5.5, L_3-cache-misses in Figure 5.6 and floating-point operations in Figure 5.7). The x-axis marks the sample number.

Figure 5.5 displays a positive correlation where the instructions retired decrease, while at the same time L_2-cache-misses decrease. The figure displays a positive correlation since the values of these two measurements decrease/increase in a similar fashion. In the background, we discuss the importance of pipeline stalls, and L_2-cache-misses is one event that causes stalls, but the graphs show a different result where a decrease in L_2-cache-misses also leads to performance degradation. This phenomenon occurs because the L_2-cache depends on data from the L_3-cache. If the L_3-cache suffers a lot of misses, it cannot provide enough data for the L_2-cache to operate at maximum capacity. Therefore, the number of L_2-cache accesses will decrease, which also results in a decrease in L_2-cache-misses. The phenomenon means that the matrix multiplication is not L_2-cache bound but instead is bound to another memory unit higher up in the memory hierarchy.

Figure 5.6 instead shows a negative correlation, where an increase in L_3-cache misses results in performance degradation. The matrix multiplication is thus L_3-cache-bound.

Figure 5.7 shows an insignificant correlation, where the performance varies while the performance counter event displays no changes over time. The ma-

trix multiplication is thus not bound to the Floating point operations.

Paper A presents an introduction to LLC contention and motivates our automatic resource-boundness detection approach, showing the complexity behind finding which applications are prone to resource contention. Paper B investigates correlation between a matrix multiplication's instructions retired and L_3-cache-misses and presents the foundation idea behind resource-boundness. Paper C introduces the LLM-shark tool and utilizes correlation to determine if an application should receive cache partition allocations in a cache partitioned system. The paper presents six applications, of which three (HARRIS, Matmult, SIFT) are bound to the LLC and three (SUSAN, SORT, FAST) are bound to other resources. We evaluate the correlation methodology using Leech tests, where we deliberately introduce an excessive amount of LLC misses using an application called Leech. Our tests reveal that highly LLC-bound applications display more significant performance degradations due to LLC contention than low correlated applications. Finally, paper G investigates different auto-regressive modeling approaches to foresee cache usage of different applications. Paper A, B and G runs on a live Intel® system while paper C runs on a live ARM system.

5.3 TC3 – Methods for measuring the degree of resource-isolation in a system

This contribution tests available isolation techniques and investigates the performance impact caused by a specific isolation technique. However, isolation techniques are often complex, since they have to override mechanisms of the operating system. Adding more isolation mechanisms to an operating system can thus decrease the overall performance of tasks. The application of isolation techniques, therefore, presents two questions. Firstly, does the proposed technique provide the promised isolation? Secondly, how much impact does the isolation technique have on applications' performance in the system?

To answer these questions, we create a methodology to verify the isolation gained, that is, increase I (Equation 5.1), employing the proposed methodology, using the setup depicted in Figure 5.8.

We divide our methodology into two steps:

1. Measure execution time of an application without any deliberately disturbing loads (baseline)

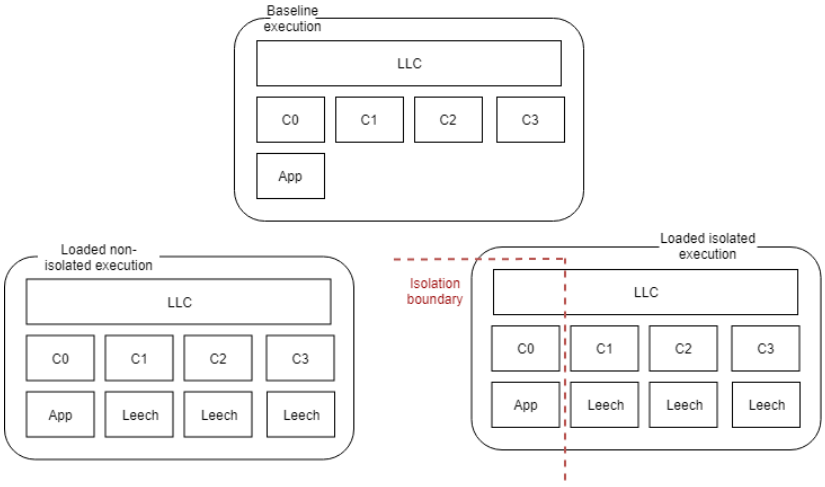

Figure 5.8: Verification of isolation

2. Measure execution time of an application while running deliberately resource heavy loads on other cores (loaded)

We execute these two steps using a non-isolated environment and an isolated environment. We then compare the executions in these environments and calculate the isolation coefficient - see Equation 5.1, where C_L is the execution time of an application C running simultaneously with other intentionally memory disturbing loads on other cores. O stands for induced overhead by system externals such as partitioning solutions. C_P stands for the baseline performance of application C, that is, the execution time of application C without any deliberately disturbing loads on other cores.

$$I = \frac{C_L - O}{C_P} \tag{5.1}$$

Non-isolated environments induce no additional overhead to the system and therefore O will be equal to 0 for a non-isolated system. We expect that a loaded non-isolated environment suffers significant performance degradation due to shared-resource contention and therefore, the C_L will be greater than C_P and I will be greater than 1. Furthermore, we expect an isolated environment to have an execution time close to the baseline execution time, with variations given by overhead (see Equation 5.1) of the isolation technique. A perfectly isolated system will, therefore, have an isolation coefficient I equal to 1.

We have tested two isolation techniques, the static Jailhouse partitioning hypervisor [27] and the combined LLC and DRAM partitioning kernel module PALLOC [37]. We measure the degree of isolation obtained using the Jailhouse hypervisor and performed similar experiments using the PALLOC kernel module. Both isolation techniques show isolation improvements in their respective domains, PALLOC as an LLC isolating tool and Jailhouse as a CPU/local cache isolation tool.

Paper D mainly addresses TC 3, illustrating the isolation gained from the Jailhouse hypervisor running on an ARM cortex A-53 chip with a petalinux 4.9 kernel.

Paper C and F also touch TC 3, where we use our benchmark to illustrate the isolation gained from cache partitioning through PALLOC on an ARM cortex A-53 processor (Paper C) and an Intel® Core™ i5-8850H processor (Paper F).

5.4 TC4 – Dynamic allocation of cache memory

In this contribution, we explore resource allocation management of the cache isolation technique called page coloring. We utilize the combined DRAM/cache partitioning tool PALLOC [37] as an actuator to partition the cache memory and thus, remove or diminish the cache contention from simultaneously executing applications. This contribution splits into three problems, listed as follows.

- Initial setup – Changes in cache partition size will dramatically affect a cache-bound application performance, but will not affect a non-cache-bound application to the same extent. It is critical to allocate cache partitions only to applications that are cache bound to avoid the waste of a precious resource. Applications with a higher cache-boundness factor should receive a more sizeable cache partition space than low cache-bound applications.

- Dynamic adjustment – There is a risk that the system applications do not execute at the desired QoS using the initially suggested cache partitions. Therefore, we need to dynamically adapt the cache partition size during the run-time of the system applications to answer to QoS needs.

- Stop trigger – There are scenarios where an increase in cache partition space does not benefit an application performance, since the assigned

cache partition space provides sufficient memory to cover the application's memory footprint. Therefore, assigning more cache partitions to this application will only result in a waste of cache space.

In paper C, we address the initial setup problem and automatically allocate more LLC partition space according to the application's resource boundness. We show that it prefferable is to allocate cache partitions based on their resource-boundness rather than allocating partitions based on the count ofLLC misses.

In paper E, we expand the LLM-shark tool to manage continuously running applications in a multi-core system. We implement an LLC partition control algorithm onto a system that contains three continuously running applications. Two are strongly cache-bound (matrix multiplications), and one application (SUSAN) does not show a strong LLC-boundness. We show in our experiments different scenarios on how our controller assigns LLC partitions to meet the setpoint performance of each applications.

There are circumstances where the number of cache partitions has reached a saturation point, i.e., no more performance will be gained from increasing the cache partition size for an application. Paper F proposes a cache-saturation detection mechanism using correlation, that measures when an application does not benefit from an increased LLC partition space.

5.5 Summary of papers

Table 5.1 summarizes how each paper covers each contribution and how each contribution links to the research challenges. We denote thesis contributions as **TC** and research challenges as **RC**.

Table 5.1: The contribution of the individual papers to the research sub-goals

Papers	TC1	TC2	TC3	TC4	RC1	RC2
Paper A		✓			✓	
Paper B	✓	✓			✓	
Paper C	✓	✓	✓	✓	✓	✓
Paper D			✓			✓
Paper E				✓		✓
Paper F				✓		✓
Paper G		✓			✓	

5.6 Overview of included papers

5.6.1 Paper A: Measurement-based evaluation of data-parallelism for OpenCV feature-detection algorithms

Jakob Danielsson, Marcus Jägemar, Tiberiu Seceleanu, Moris Behnam, Mikael Sjödin

Summary. We investigate the effects on the execution time, shared cache usage, and speed-up gains when using data-partitioned parallelism for the feature detection algorithms available in the OpenCV library. The purpose of this paper is to investigate how to cache contention affects the performance of parallelized workloads and also to give an insight into how performance counters can be used to localize cache contention. The measurements are used to conclude which algorithms are suitable for parallelization on hardware with shared resources.

Thesis contribution TC2

Research challenge RC1

Author's contribution I am the initiator, main driver and author to all parts in this paper. All other co-authors have contributed with valuable discussions and reviews.

Status Published in proceedings of 42^{nd} Computer Society Signature Conference on Computers, Software and Applications (COMPSAC), 2018, IEEE

5.6.2 Paper B: Resource Dependency Analysis in Multi-core systems

Jakob Danielsson, Marcus Jägemar, Tiberiu Seceleanu, Moris Behnam, Mikael Sjödin

Summary. In this paper, we evaluate different methods for statistical determination of application resource dependency in multi-core systems. We measure the performance counters of an application during run-time and create a system resource usage profile. We then use the profile to evaluate the application dependency on the specific resource. We discuss and evaluate two methods to process the data, including moving average filter and partitioning the data into smaller segments in order to interpret data for correlation calculations. Our aim with this study is to evaluate and create a generalizeable method for automatic determination of resource dependencies. The final outcome of the

methods used in this study is the answer to the question: "On which resources is this application dependent?". The recommendation of this tool will be used in conjunction with our last-level cache partitioning controller (LLC-PC), to make decision if an application should receive last-level cache partition slices.

Thesis contribution TC1 and TC2

Research goal RG1

Author's contribution I am the initiator, main driver and author to all parts in this paper. All other co-authors have contributed with valuable discussion and reviews.

Status Published in proceedings of 44^{th} Computer Society Signature Conference on Computers, Software and Applications (COMPSAC), 2020, IEEE

5.6.3 Paper C: LLM-shark – A Tool for Automatic Resource - boundness Analysis and Cache Partitioning Setup

Jakob Danielsson, Marcus Jägemar, Tiberiu Seceleanu, Moris Behnam, Mikael Sjödin

Summary. We present LLM-shark, a tool for automatic hardware resource-boundness detection and cache-partitioning. Our tool has three primary objectives: first, it determines the hardware resource-boundness of a given application; secondly, it estimates the initial cache partition size to ensure that the application performance is conserved and not affected by other processes competing for cache utilization; thirdly, it continuously monitors that the application performance is maintained over time and, if necessary, changes the cache partition size. We demonstrate LLM-shark's functionality through a series of tests using six different applications, including a set of feature detection algorithms and two synthetic applications. Our tests reveal that it is possible to determine an application's resource-boundness using a correlation scheme implemented in LLM-shark. We propose a scheme to size cache partitions based on the correlation coefficient applications depending on their resource boundness.

Thesis contribution TC1, TC2, TC3 and TC4

Research challenge RC1 and RC2

Author's contribution I am the initiator, main driver and author to all parts in this paper. All other co-authors have contributed with valuable discussion and reviews.

Status Published in proceedings of 45^{th} Computer Society Signature Conference on Computers, Software and Applications (COMPSAC), 2021, IEEE

5.6.4 Paper D: Run-Time Cache-Partition Controller for Multi-Core Systems

Jakob Danielsson, Marcus Jägemar, Tiberiu Seceleanu, Moris Behnam, Mikael Sjödin

Summary. In this paper, we investigate how to define performance isolation in multi-core systems. We investigate resource contention in the CPU, LLC, and also the memory bus. We define a test to determine the level of isolation gained by the isolation hypervisor called Jailhouse in comparison with a regular Linux system. Our paper concludes that the Jailhouse hypervisor does not produce any noticeable overhead when executing multiple shared-resource intensive tasks on multiple cores, which implies that running Jailhouse in a memory saturated system will not be harmful.

Thesis contribution TC3

Research challenge RC2

Author's contribution I am the initiator, main driver and author to all parts in this paper. All other co-authors have contributed with valuable discussion and reviews.

Status Published in proceedings of 43^{rd} Computer Society Signature Conference on Computers, Software and Applications (COMPSAC), 2019, IEEE

5.6.5 Paper E: Automatic Quality of Service Control in Multi-core Systems using Cache Partitioning

Jakob Danielsson, Marcus Jägemar, Tiberiu Seceleanu, Moris Behnam, Mikael Sjödin

Summary. We present a last-level cache partitioning controller for multi-core systems. Our objective is to control the Quality of Service (QoS) of applications in multi-core systems by monitoring run-time performance and continuously re-sizing cache partitions, according to the application needs. We discuss two different use-cases; one that promotes application fairness and another one that prioritizes applications according to the system engineers' desired execution behavior. We display the performance drawbacks of main-

taining a fair schedule for all system tasks and its performance implications for system applications. We, therefore, implement a second control algorithm that enforces cache partition assignments according to user-defined priorities rather than system fairness. Our experiments reveal that it is possible, with non-instrusive (0.3-0.7% CPU utilization) cache controlling measures, to increase performance according to setpoints and maintain the QoS for specific applications in an over-saturated system.

Thesis contribution TC4

Research challenge RC2

Author's contribution I am the initiator, main driver and author to all parts in this paper. All other co-authors have contributed with valuable discussion and reviews.

Status Accepted in proceedings of 26^{th} Emerging Technologies and Factory Automation (ETFA), 2021, IEEE

5.6.6 Paper F: Run-Time Cache-Partition Controller for Multi-Core Systems

Jakob Danielsson, Marcus Jägemar, Tiberiu Seceleanu, Moris Behnam, Mikael Sjödin

Summary. We propose a cache partition controller called LLC-PC that uses the PALLOC page coloring framework to decrease the cache partition sizes for applications during run-time. LLC-PC creates cache partitioning directives for the PALLOC tool by evaluating the performance gained from increasing the cache partition size. We have evaluated LLC-PC using three different applications, including the SIFT image processing algorithm, a matrix multiplication, and a random number generator. We show that LLC-PC can reduce the amount of cache size allocated to applications compared to intuitively chosen cache partitions while maintaining their performance. LLC-PC thus allows for more cache space to be allocated for other applications.

Thesis contribution TC4

Research goal RC2

Author's contribution I am the initiator, main driver and author to all parts in this paper. All other co-authors have contributed with valuable discussion and reviews.

Status Published in proceedings of 45^{th} Annual Conference of IEEE Industrial Electronics Society (IECON), 2019, IEEE

5.6.7 Paper G: Modelling Application Cache Behavior using Regression Models

Jakob Danielsson, Marcus Jägemar, Tiberiu Seceleanu, Moris Behnam, Mikael Sjödin

Summary In this paper, we describe the creation of resource usage forecasts for applications with unknown execution characteristics, by evaluating different regression processes, including autoregressive, multivariate adaptive regression splines, exponential smoothing, etc. We utilize Performance Monitor Units (PMU) and generate hardware resource usage models for the L_2-cache and the L_3-cache using nine different regression processes. The measurement strategy and regression process methodology are general and applicable to any given hardware resource when performance counters are available. We use three benchmark applications: the SIFT feature detection algorithm, a standard matrix multiplication, and a version of Bubblesort. Our evaluation shows that Multi Adaptive Regressive Spline (MARS) models generate the best resource usage forecasts among the considered models, followed by Single Exponential Splines (SES) and Triple Exponential Splines (TES).

Thesis contribution TC2

Research challenge RC2

Author's contribution I am the initiator, main driver and co-author to all parts in this paper. All other co-authors have contributed with valuable discussion and reviews.

Status Published in 11^{th} IEEE International Workshop on Industrial Experience in Embedded Systems Design (IEESD)

Chapter 6

Conclusions and Future Work

The main goal of this thesis is to understand the origins of shared-resource contention, investigate the reasons of it, propose means to solve the resource contention using isolation techniques, and finally propose automatic allocation strategies for adjusting application performance according to user-specified needs. Our investigation in paper A is designed to display the complexity of the resource contention problem – some applications suffer greatly from the cache contention while others do not.

We propose the resource-boundness concept for understanding the origins of resource contention. We measure an application's resource-boundness using the internal performance counters and specifically target the LLC as the source of contention. Our proposal is a correlation-based methodology between the number of instructions retired and the LLC misses. We propose that applications that show a negative correlation between the number of instructions retired and the number of LLC misses are bound to the LLC and thus risk suffering performance degradation due to LLC contention. Our methodology applies to all computer architectures that provide the performance counter utility. However, the resource-boundness results will vary, depending on the size of the hardware resources on different platforms.

We utilize resource-boundness to determine what kind of isolation techniques are suitable for a particular application. We investigate the usability of two isolation strategies: i) Jailhouse virtualization applies to applications that utilize I/O and core specific resources, such as the local caches and the floating-point unit; ii) Palloc page-coloring, which applies to applications that utilize the DRAM and LLC memory. We investigate in our papers how these strategies provide isolation and investigate the potential drawbacks in terms of overhead.

Jailhouse provides an isolation layer towards the global Linux scheduler and hides the partitions, making it impossible for the Linux scheduler to assign new tasks to the hidden partitions arbitrarily. Thus Jailhouse isolates core-specific resources such as the floating point unit, branch predictor unit, and local caches. Jailhouse shows a slight performance degradation of our test tasks, which happens due to an increase in LLC and memory bus usage due to the additional complexity in the Jailhouse MMU hierarchy. Palloc partitions the LLC and thus provides an isolation layer for applications that utilize the LLC. Palloc displays a notable performance degradation of our test tasks due to the complexity of the page coloring algorithm. However, the drawbacks of the performance may still outweigh the performance degradations caused by cache contention.

Our last contribution targets the adaptivity and load-balancing of tasks that run in a system with limited cache memory. We propose methods that:

1. Allocate cache memory at system start.
2. Re-allocate cache memory during system run-time.
3. Provide stop mechanisms for allocating cache memory, to determine when an increased cache partition space is no longer beneficial to an application.

Our methodology utilizes the resource-boundness concept to formulate initial cache partitions and effectively filters out applications that do not need cache partition space, as different to other methodologies that do not provide such a functionality. In addition, we implement two strategies (fairness and prioritization) to continuously load-balance the cache memory of applications that run simultaneously on different cores. Our final efforts investigate a stop mechanism that detects when an application does not benefit from additional cache partition space.

6.1 Future Work

In paper G, we compare different autoregressive models to model and forecast an application resource usage. Interesting future work is to schedule applications according to their resource usage and counter resource contention before it happens using resource-aware scheduling strategies. The benefit of using resource-aware scheduling is that we might be able to remove many pipeline stalls due to resource contention and thus allow the computer to do more useful work.

Current isolation methods are promising solutions to reduce unexpected performance degradations due to resource contention, but our work primarily focuses on optimizing the partitioning allocations to load-balance application performance. Isolation is primarily a method for real-time systems where deadline misses can become catastrophic; in this thesis, we do not focus on real-time verification of our holistic architecture. We, therefore, envision real-time verification as an interesting future work, where we execute time-critical loads inside our LLM-shark architecture to test its real-time applicability.

Other exciting works include incorporating more sophisticated algorithms into the cache-management part. For example, we currently implement a linear controlling algorithm due to the small amount of available cache memory in our ARM-chip. We believe that when the cache memory is more spacious, it will become more beneficial to incorporate a proportional element to the cache allocation controller, optimizing the time it takes to find the saturation point.

Bibliography

[1] ARM. Cortex-a53. https://developer.arm.com/ip-products/processors/cortex-a/cortex-a53. 2021-01-06.

[2] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *11th International Symposium on High-Performance Computer Architecture*, pages 340–351. IEEE, 2005.

[3] J. Che, Q. He, Q. Gao, and D. Huang. Performance measuring and comparing of virtual machine monitors. In *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, volume 2, pages 381–386. IEEE, 2008.

[4] W. commons. Risc architecture. accessed: 2019-11-04.

[5] J. Danielsson, M. Jägemar, M. Behnam, T. Seceleanu, and M. Sjödin. Run-time cache-partition controller for multi-core systems. In *IECON 2019-45th Annual Conference of the IEEE Industrial Electronics Society*, volume 1, pages 4509–4515. IEEE, 2019.

[6] J. Danielsson, M. Jägemar, M. Behnam, M. Sjödin, and T. Seceleanu. Measurement-based evaluation of data-parallelism for opencv feature-detection algorithms. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 701–710. IEEE, 2018.

[7] S. Di Carlo, G. Gambardella, M. Indaco, D. Rolfo, and P. Prinetto. Marciatesta: an automatic generator of test programs for microprocessors' data caches. In *2011 Asian Test Symposium*, pages 401–406. IEEE, 2011.

[8] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache pirating: Measuring the curse of the shared cache. In *2011 International Conference on Parallel Processing*, pages 165–175. IEEE, 2011.

[9] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Bandwidth bandit: Quantitative characterization of memory contention. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE, 2013.

[10] https://github.com/uart/scarphase. Scarphase tool.

[11] Y. Hua, Z. C. Ping, Z. Z. Hui, Z. Wei, and P. Z. Jin. Understanding performance-resource dependency by thread slicing and curve fitting. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 1, pages 17–22. IEEE, 2011.

[12] Intel. Intel product catalogue.

[13] Intel®. Intel® 64 and ia-32 architectures optimization reference manual. `https://software.intel.com/en-us/download/`. Accessed: 2019-11-04.

[14] M. Jägemar, A. Ermedahl, S. Eldh, and M. Behnam. A scheduling architecture for enforcing quality of service in multi-process systems. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2017.

[15] H. Kasture and D. Sanchez. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. *ACM SIGPLAN Notices*, 49(4):729–742, 2014.

[16] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14. IEEE, 2019.

[17] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 workshop on Experimental computer science*, page 6. ACM, 2007.

[18] S. Mittal. A survey of techniques for cache partitioning in multicore processors. *ACM Computing Surveys (CSUR)*, 50(2):1–39, 2017.

[19] S. Mittal and Z. Zhang. Manager: a multicore shared cache energy saving technique for qos systems. *Iowa State University, Tech. Rep*, 2013.

[20] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero. Flexdcp: a qos framework for cmp architectures. *ACM SIGOPS Operating Systems Review*, 43(2):86–96, 2009.

[21] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[22] A. Pan and V. S. Pai. Imbalanced cache partitioning for balanced data-parallel programs. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 297–309. IEEE, 2013.

[23] S. A. Panchamukhi and F. Mueller. Providing task isolation via tlb coloring. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 3–13. IEEE, 2015.

[24] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer. Modeling performance variation due to cache sharing. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE* 19^{th} *International Symposium on*, pages 155–166. IEEE, 2013.

[25] A. Sembrant, D. Black-Schaffer, and E. Hagersten. Phase behavior in serial and parallel applications. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 47–58. IEEE, 2012.

[26] A. Sembrant, D. Eklov, and E. Hagersten. Efficient software-based online phase classification. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 104–115. IEEE, 2011.

[27] A. Siemens. Jailhouse partitioning hypervisor. *Retrieved March*, 2016.

[28] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 62–75. IEEE, 2015.

[29] L. Torvalds. Perf tools. accessed: 2019-11-04.

[30] S. Toumassian, R. Werner, and A. Sikora. Performance measurements for hypervisors on embedded arm processors. In *Advances in Computing, Communications and Informatics (ICACCI), 2016 International Conference on*, pages 851–855. IEEE, 2016.

[31] S. H. VanderLeest. Arinc 653 hypervisor. In *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA* 29^{th}, pages 5–E. IEEE, 2010.

[32] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. Cache contention and application performance prediction for multi-core systems. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 76–86. IEEE, 2010.

[33] M. Xu, L. T. X. Phan, H.-Y. Choi, Y. Lin, H. Li, C. Lu, and I. Lee. Holistic resource allocation for multicore real-time systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 345–356. IEEE, 2019.

[34] M. Xu, L. Thi, X. Phan, H.-Y. Choi, and I. Lee. vcat: Dynamic cache management using cat virtualization. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 211–222. IEEE, 2017.

[35] X. Xu, F. Zhou, J. Wan, and Y. Jiang. Quantifying performance properties of virtual machine. In *2008 International Symposium on Information Science and Engineering*, volume 1, pages 24–28. IEEE, 2008.

[36] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: a dynamic cache partitioning system using page coloring. In *Parallel Architecture and Compilation Techniques (PACT), 2014* 23^{rd} *International Conference on*, pages 381–392. IEEE, 2014.

[37] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 155–166. IEEE, 2014.

[38] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE* 19^{th}, pages 55–64. IEEE, 2013.

[39] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 27–38, 2011.

Part II

Included Papers

Chapter 7

Paper A: Measurement-based evaluation of data-parallelism for OpenCV feature-detection

Jakob Danielsson, Marcus Jägemar, Tiberiu Seceleanu, Moris Behnam and Mikael Sjödin. Measurement-based evaluation of data-parallelism for OpenCV feature-detection algorithms. In *42nd Computer Society Signature Conference on Computers, Software and Applications (COMPSAC)*. IEEE, 2018.

Abstract

We investigate the effects on the execution time, shared cache usage and speed-up gains when using data-partitioned parallelism for the feature detection algorithms available in the OpenCV library. We use a data set of three different images which are scaled to six different sizes to exercise the different cache memories of our test architectures. Our measurements reveal that the algorithms using the default settings of OpenCV behave very differently when using data-partitioned parallelism. Our investigation shows that the executions of the algorithms SURF, Dense and MSER correlate to L3-cache usage and they are therefore not suitable for data-partitioned parallelism on multi-core CPUs. Other algorithms: BRISK, FAST, ORB, HARRIS, GFTT, SimpleBlob and SIFT, do not correlate to L3-cache in the same extent, and they are therefore more suitable for data-partitioned parallelism. Furthermore, the SIFT algorithm provides the most stable speed-up, resulting in an execution between 3 and 3.5 times faster than the original execution time for all image sizes. We also have evaluated the hardware resource usage by measuring the algorithm execution time simultaneously with the L3-cache usage. We have used our measurements to conclude which algorithms are suitable for parallelization on hardware with shared resources.

7.1 Introduction

Many industrial systems often use feature detection algorithms in various applications ranging from face recognition to autonomous vehicular systems. Detecting features in a frame is a time-consuming process [5] because of the high number of traversed pixels. The number of traversed pixels depends highly on the feature detection algorithm goal, e.g., detecting objects, corners, edges, blobs or key points. The number of traversed pixels affects the application execution time, which is often a limitation for time-sensitive real-time systems.

The process of feature detection stipulates that different calculation sequences search for specific conjunctions between pixels in a frame. The length of the feature detection sequence varies significantly among the used algorithm. The number of traversed pixels per frame grows if the feature detection sequence is long leading to a further increased execution time.

One way of decrease the execution time of these calculations is to parallelize the execution and use multiple CPU-cores at the same time. The computations for a frame are often suitable for execution on parallel architectures, where each CPU can operate on a sub-frame (i.e. a partition of the original frame). Luckily, almost all processors available today are, so called, multi-core processors which have at least 2 CPU cores.

However, in a multi-core architecture, the computing units compete for access to common hardware resources, such as caches, memory banks and memory buses. This competition lead to challenges in designing parallel software to avoid bottlenecks in the data-flow and to prevent computing units from interfering with each other. Examples of performance problems related to parallel execution include cache trashing (one core evicts data from the cache that is needed by another core), cache-line ping-pong (a false-sharing problem when cores that are seemingly unrelated manipulate data-elements that are allocated close in memory), and DRAM starvation (the DRAM controller may choose to serve only memory requests from one controller for a while, since that brings up the throughput of the memory system - at the expense of long delays for some cores).

The ideal execution environment for a feature detection algorithm running on a multi-core architecture is identified by several properties. Minimizing the shared-memory congestion side effects and interprocess synchronization time are the most important ones. One possible solution to reduce the harmful effects of shared resource congestion is to monitor and understand the algorithm

resource usage before-hand [15]. It is possible to obtain such knowledge by, for instance, using Performance Measurement Counters (PMC) [7].

The knowledge of how feature detection algorithms such as FAST, HARRIS or SURF affect the shared resources is an important part when incorporating them into a multi-core system, since it can give an indication on how well the algorithm scales with parallelism opportunities offered by multi-cores. Since the input data to such algorithms can be relatively large, there is a possibility that the algorithms may suffer from shared memory congestion and therefore obtain an insignificant speed-up when utilizing multiple cores. Therefore, it is possible that a feature detection algorithm has such characteristics that it is better suited for running on a single core, together with other general workloads instead of reserving the several computational units of the computer while achieving little execution time gains. However, the success of applying a parallel paradigm to a feature detection algorithm can however be an efficient tool to decrease the execution time of such heavy workloads.

In this paper we study how the feature-detection algorithms using the Open Computer Vision (OpenCV) library [4] behaves with respect to data-level parallelization in terms of L3 cache usage on multi-core processors. OpenCV is one of the most widespread libraries for image processing and hence these results should be valuable for a large community. The main contributions in this paper include:

- We have evaluated how the feature detection algorithms in the OpenCV features2d module [19] perform from data partitioned parallelism with respect to speed-up.
- We have measured the performance of the feature detection algorithms in the OpenCV features2d module together with each algorithm hardware resource usage. From these measurements, we deduced that the L_3-cache has the highest effect on the algorithm performance.

Outline: Section 7.2 give background information related to feature detection algorithms and their resource usage. Detailed information on our implementation is given in Section 7.3 and the experiments in Section 7.4. We conclude the paper by summarizing our conclusions in Section 7.5.

7.2 Background

It is possible to run image processing on multi-core systems with the purpose of decreasing the execution time by using coarse-grained data parallelized algorithms [27]. Relevant work include investigating how to parallelize feature detection algorithms such as SIFT [10], [29], SURF [28], and Harris [12] for performance increase. Applying these parallelization techniques however require an in-depth investigation of the algorithm functionality and also how to adapt the functionality parameters to the hardware in use. In this paper, we have instead executed a generalized coarse-grained parallelism model which can be applicable for speed-up gains without studying the workload in detail. Since our approach does not require in-depth knowledge of neither the hardware or the software, it is also easy to migrate between different hardware setups. In this paper, we have executed a generalized coarse-grained parallelism model which can be applicable even though the work-load is not studies in detail. To the best of our knowledge, our paper is the first that investigates the effects data-level parallelism has on the shared memory using OpenCV feature-detection algorithms. The algorithms investigated in this paper are well established feature detection algorithms, available in the free and non-free branches of *features2d* in the OpenCV library. We have used the default algorithm tuning values which come with the OpenCV library in order to have a reference for the comparison.

7.2.1 Feature detection

Feature detection is a way of distinguishing anomalies in an image. Feature detection can be divided into 4 sub-sets, edge detection, corner detection, object detection and blob detection. In this work, we have used the common interfaces class [19] of the OpenCV library which implements 11 different feature detection algorithms listed in Table 7.1.

A feature detection algorithm is typically built upon a set of mathematical rules which defines a corner. These mathematical rules control not only how a corner is defined, but also how the pixels in a frame are accessed. The main mechanism of every corner detection algorithm is to traverse each pixel within a frame. Detecting a corner in an image can become a costly process in terms of hardware resources since frames become larger as a consequence of higher resolution, which lead to an increased amount of pixels which have to be traversed. Larger frames can also potentially contain more corners, which furthermore increases the processing time of an image.

Table 7.1: Our investigated feature detection algorithms.

Algorithm	License	Description
Harris [11]	BSD	Corner detector
FAST [23]	BSD	Corner detector
SIFT [16]	Proprietary	Object detector
SURF [3]	Proprietary	Object detector
ORB [24]	BSD	Object detector
BRISK [14]	BSD	Corner detector
MSER [17]	BSD	Blob detector
GFTT [26]	BSD	Corner detector
STAR [1]	BSD	Corner detector
DENSE [4]	BSD	Feature extractor
Simple blob [4]	BSD	Blob detector

Feature detection algorithms use different mechanisms for detecting interest points in an image. There are although some common stages for all algorithms. The first step is always to read the input image file and translate it into a matrix filled with RGB (Red, Green, Blue) data points, where each data point represents a pixel. The second common step is to convert the image in-to grayscale, which is translates the RGB values to a matrix of pixel intensities, which represent values of the brightness of the pixels. After this step, the algorithms begin to execute their respective interest point detection mechanism. The actual detection mechanisms differs a lot depending on the algorithm. To exemplify a diversity, we have depicted the mechanisms of two feature detection algorithms in Fig. 7.1. The figure illustrates a Sobel filter (marked 1 with purple boxes) which serves as one of the primary mechanisms for the Harris algorithm and a Bresenham Circle (marked 2 with blue boxes) which is the main mechanism of the FAST algorithm.

The second property all algorithms have in common is that the entire image matrix gets traversed at least once. Algorithms such as SURF and SIFT create new matrices that contain results from the initial image matrix. The algorithm repeatedly traverses the original image matrix until it has processed the complete image. There can also be co-dependence between the algorithms, meaning that one algorithm uses the results given by another algorithm. For example, ORB uses the result of Harris or FAST to detect objects. The last step of a feature detection algorithm is to return the pixels considered to be featured. OpenCV calls these features keypoints.

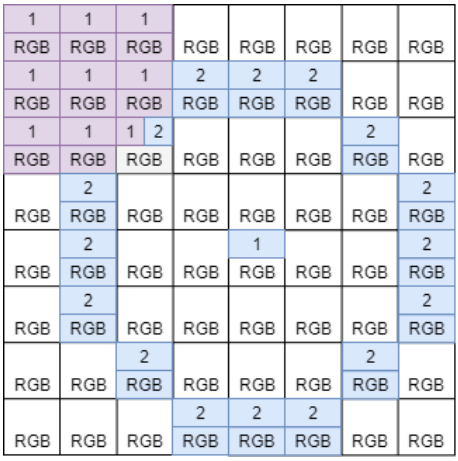

Figure 7.1: Example of FAST and Harris.

7.2.2 Parallel programming

There are various approaches reduce the execution time through parallelism [21]. Designing a feature detection program with a fork-join is one way of utilizing the core-level parallelism, which is efficient due to the mechanics of these algorithms. A fork-join model has two parts controlled by the main thread. First, the fork section where one or several tasks, feasible for parallelization, are allocated over the available CPU cores. The main thread resumes its execution when all spawned tasks have finished and entered the join section. Fig. 7.2 illustrates an example of the fork-join model utilizing 4 cores.

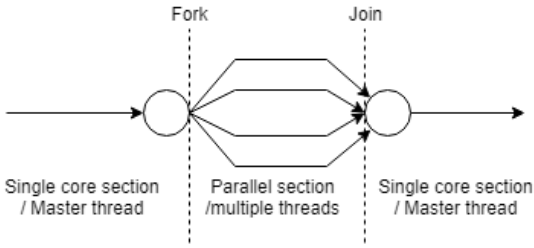

Figure 7.2: The fork-join model for parallelization of algorithms.

The fork-join model is a trivial way when trying to increase the performance of feature detection algorithms since there are no global variables shared. This means the algorithms can be split up to work on sub-parts of an image without interfering with another sub-part of the image.

7.2.3 Shared memory

Shared resource congestion is one of the major limiting performance factor when running applications, such that the application performance is correlated to the shared resource usage [13]. The resource usage of an application is usually measured by the Performance Monitoring Unit (PMU) [20], such as Intel [15], and deduce resource bottlenecks [7]. The application performance is typically [8, 9] measured in an application-specific metric [2]. In this paper we are mostly concerned with L_3-cache usage because it is the first system-wide shared resource, which makes it the first resource that is eligible to suffer from multi-core memory contention.

It is difficult to correlate the cache usage to execution time [6] when running applications on a HW with shared caches. Sanberg et al [25] focus on understanding and modeling the execution behavior caused by a congested shared cache. It is also possible to quantize how cache misses affect the system performance by profiling the resource usage of a system [22].

Most non-dedicated computer systems utilize caches to be able to access data quickly. However, the cache is often a costly part of a processor, which limits the amount of available to the CPU. The limited cache size force most CPU to implement cache eviction policies to remove less-used data from the cache and replace it with new data. One of the most commonly known algorithms for replacing data inside a cache is the Least Recently Used (LRU) policy. The LRU tracks data usage, and the least recently used data is removed from the cache and replaced with the new data when the cache is congested. Multi-core systems often make use of a shared cache when communicating between threads and processes. Shared caches of a multi-core processor can, however, lead to negative behavior when using policies such as the LRU policy. When multiple threads access the same memory, the risk is that one thread requests a block of data from the DRAM that replaces the data which was about to be read by another thread. Such congestion scenarios can lead to cache thrashing, where several threads continuously replace each other's data, which in turn can lead to a significant system performance decrease. Computers which execute corner detection algorithms and use a fork-join model will, at some point, have to use the shared resources, such as caches and memory. Shared caches may not be a problem if the image fits into the local cache. Such favorable scenario happens, for example, when the feature detection algorithm can process the whole image in a single iteration, i.e., before other processes replace the cache content. However, the processed memory depends highly on the used algorithm. We have focused to investigate the effects that shared cache congestion causes

on the speed-up gains when using the OpenCV feature detection algorithms utilizing a data-partitioned fork-join model.

7.3 Approach

Our study consists of two parts. The first part is a program that implements the OpenCV algorithms and samples the desired performance counters simultaneously as the test execution time. The second part analyzes the measurements.

7.3.1 OpenCV feature detection

OpenCV provides an overlying feature detection class that contains 11 different feature detection algorithms. We have used a data-partitioned fork-join model for evaluating the OpenCV library on multi-core systems. We depict the execution model in Fig. 7.3.

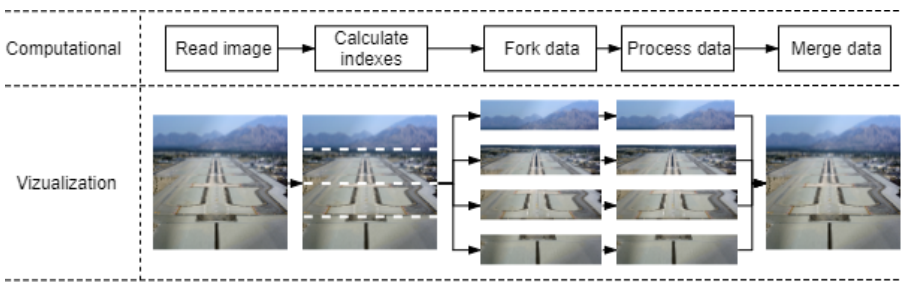

Figure 7.3: The image data is partitioned to support the fork-join model.

Fig. 7.3 shows how the workload is distributed to the different cores of our system. At the fork stage, each thread has its affinity set to a core which is not in use by the algorithm, which means thread 0 gets affinity 0 and therefore executes on core 0 and so on. The thread affinity is furthermore used for partitioning the Image. For partitioning the image, we have chosen to divide each image on a height basis. The threads work horizontally on the indexes calculated according to equation (7.1) where $Work_x$ is the work indexes and $ImageSize_x$ is the horizontal size of the image. The vertical workload is calculated according to equation 7.3 and 7.4, where UpperBound is upper vertical index bound, LowerBound is the lower vertical index bound, $ImageSize_y$ is the size of the entire image and *aff* is the core affinity of the current thread, which is indexed between 0 and n-1, where 0 is the first core and n-1 is the last core.

$$Work_x = ImageSize_x \tag{7.1}$$

$$if(aff) = 0, \quad UpperBound \quad = 0 \tag{7.2}$$

$$if(aff) > 0, \quad UpperBound \quad = \frac{ImageSize_y}{aff + 1} \tag{7.3}$$

$$LowerBound = \frac{ImageSize_y}{aff + 2} \tag{7.4}$$

7.3.2 Performance Monitoring

We have implemented a system function that simultaneously monitor the resource usage and performance of an application. The following subsections describe our test up for measuring application performance and application resource usage.

7.3.2.1 Application Performance

We measure the execution time of each algorithm using the high resolution clock chronox (c++11 library) for measuring the algorithm execution time. The placement of the timestamps are depicted in Fig. 7.4.

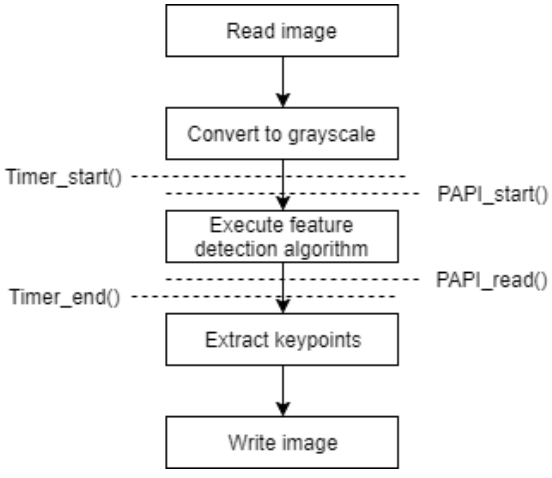

Figure 7.4: The algorithm performance measurement sequence.

7.3.2.2 Resource Usage

We monitor the number of shared cache misses by using the Performance API library (PAPI) [18] which provide an interface towards the PMU [20]. We insert PAPI start before the algorithm start and PAPI read when the algorithm is finished, as depicted in Fig. 7.4.

7.4 Experiment

We have run our experiments on a quad-core Intel® Core™ i5-3570 processor running at 3.40GHz using g++ version 5.4 with -pthread, -std=c++11 and -O3 as compiler arguments. The HW specifications are listed in Table 7.2. Streaming SIMD Extensions (SSE) instructions are enabled by OpenCV as default configuration.

Feature	Hardware Component
Core	4xIntel® Core™ i5-3570 CPU (Ivy Bridge) 3.4GHz
L_1-cache	32 KB 8-way set assoc. instruction caches/core + 32 KB 8-way set assoc. data cache/core
L_2-cache	256 KB 8-way set assoc. cache/core
LLC	6 MB 12-way set assoc. shared platform cache
MMU	64 Byte line size, 64 Byte Prefetching, DTLB: 32 entries 2 MB/4 MB 4-way set assoc. + 64 entries 4 KB 4-way set assoc., ITLB: 128 entries 4 KB 4-way set assoc., L_2Unified-TLB: 1 MB 4-way set assoc., L_2Unified-TLB: 512 entries 4 KB/2 MB 4-way assoc.

Table 7.2: Hardware specifications Intel® Core™ i5-3570.

We measure two different parameters in our test suite, utilizing different amount of cores. The first parameter is Application performance which measures the total execution time of the feature detection algorithms utilizing 1, 2, 3 and 4 cores. We then use the execution time to calculate the speed-up gained from using multiple cores compared to single core. The second parameter is the execution-time measured per image partition, which means we execute the same image partitions but on single core and compare them to

our per-core multi-core respective values. At the same time, we also measure the L_3-cache misses which describes the shared resource usage during the test execution. We have also measured the amount of keypoints detected using single-core on a full image, to be able to see what effects the amount of detected keypoints has on the speed-ups gained. In our tests we have used images designed to fit different parts of the Cache memory of our test system Intel® Core™ i5-3570k. The specification for our test images are listed in table. We present the test image size variations in Table 7.3.

Table 7.3: Image size variations and their cache boundess.

Figure nr.	Image Size	Mem. Req.	Cache boundness
1	103x103	32 KB	L_1-cache
2	209x209	131 KB	$4 \times L_1$-cache
3	295 x295	262 KB	L_2-cache
4	591x591	1 MB	$4 \times L_2$-cache
5	1431x1431	6.1 MB	L_3-cache
6	2862x2862	24.6 MB	$4 \times L_3$-cache

We have executed tests using different images, within a similar environment. The images are presented in Fig. 7.5 and follow the specifications presented listed in Table 7.3.

a) Test Image 1. b) Test Image 2. c) Test Image 3.

Figure 7.5: Test images.

The purpose of each test is to reveal the feasibility of our data-partitioned program model when using the standard OpenCV feature detection algorithms. The default parameters of the STAR algorithm in the OpenCV feature detection suite uses a specific set of image scales when executing the Laplacian operator. Some of these image scales are so large that they are not feasible for our smaller image variations, which results in a non-proportional speed-up when partitioning small images to even smaller image partitions. We have therefore exempted these inaccurate STAR detector results.

7.4.1 Data partitioned measurements

An important behavior to observe when using data partitioned parallelism is the speed-up given by executing the algorithm on multiple cores. This measurement gives us an absolute value on how well the algorithm responded to our proposed parallel data partitioned model. In this section, we present and discuss the speed-up gained by utilizing 2, 3 and 4 cores compared to 1 core. Each test on each core was repeated 500 times to provide a median of the execution times. The median execution time is then used to calculate the speed-up according to Equation (7.5), where *S* is the speed-up gained, t_0 is the single-core execution time, t_i is the execution time of core *i* and n is the number of cores used.

$$S = \frac{t_0}{\{max(t_i) : 0 \leq i < n\}} \tag{7.5}$$

Fig. 7.6 shows the speed-up of each feature detection algorithm. The y-axis denotes the gained speed-up, and the x-axis represents 3 test images, each one with 6 image size variations. The first cluster of 6 image sizes belongs to the image shown in Fig. 7.5 a, the second set to Fig. 7.5 b, and the third set to Fig. 7.5 c. We categorize speed-ups into three categories: The first is *linear speed-up*, where the resulting execution time is equal to the single core execution divided by the number of cores used. The second is *sub-optimal speed-up*, which provides a smaller speed-up than the linear one. The third and final is *super-linear speed-up* which provides a more significant speed-up than a linear one.

To increase readability, we will refer to specific test cases as $Img\_\#_{figure\_size}$ where # is the figure number.

The numbers for the BRISK detector show a sub-optimal speed-up using 4 cores. The achieved speed-up is small when using the smallest image but increases with the image size. However, the speed-up is at its peak at Img_1_{262KB}, Img_2_{1MB} and Img_3_{262KB}. When further increasing the image sizes, the speed-up decreases again. We call this behavior a pyramid-like behavior.

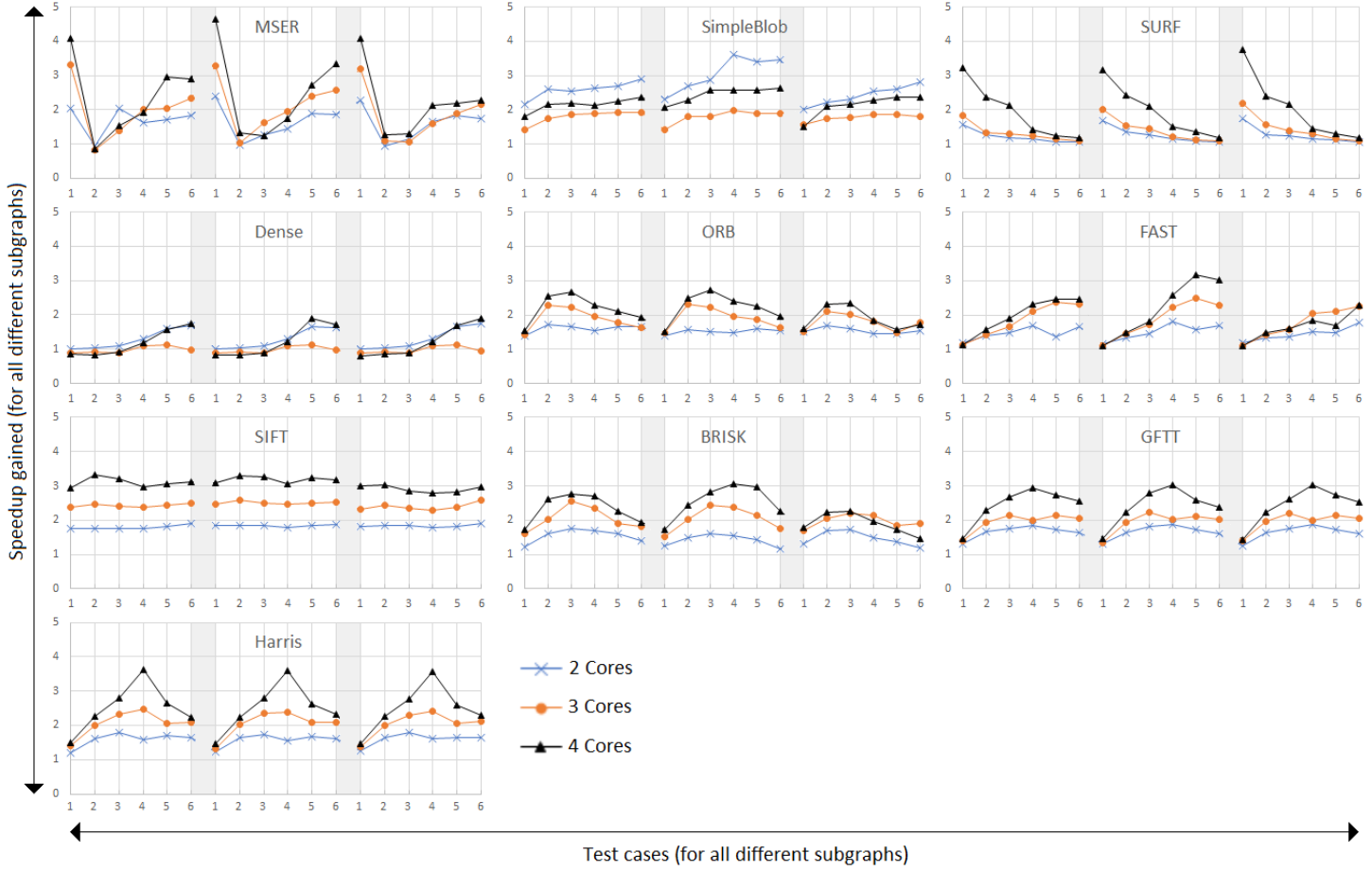

Figure 7.6: Feature detection algorithm speed-up factors for various test-cases when running a multi-core test system.

The Dense Feature detector shows a small speed-up using any of our multi-core tests, the peak speed-up is at roughly 70% faster than the original 1 core version. Furthermore, there is no gain at all from using multi-core until increasing the image size to 1 MB. The Smaller sizes of 32 KB, 128 KB, and 256 KB actually decrease the execution time compared to the single core version. The Dense detector also shows a pyramid-like behavior and has peak performance at $Img\_2_{6.1MB}$ and peak speed-up at $Img\_1_{24.6MB}$ and $Img\_3_{24.6MB}$, however, the differences between the speed-ups are roughly 15%, meaning it is small and could just be a coincidence.

The FAST feature detector has a low speed-up using the smaller image sizes and the speed-up increases with the image size. However, FAST reaches a sub-optimal performance at each speed-up peak which is between 2 to 3 times speed-up when using multi-core. The insignificant speed-up gained on the smaller images can be explained as an effect of the overhead gained by the data-partitioned parallelism. If the overhead of an algorithm is dominant, initializing the algorithm multiple times will make the algorithms parallelism less efficient, or even worse (as seen in the Dense algorithm) when using images so small that the work-load execution time does not match the overhead execution time.

The GFTT feature detector has a similar speed-up result for all three test suites. The smallest image has a speed-up of roughly 50%, which is similar to the speed-up of the largest image. Furthermore, the GFTT feature detector achieves a close to optimal speed-up using the 1 MB image. Due to the major speed-up differences, GFTT presents an even stronger pyramid behavior than the Dense and BRISK feature detector.

The speed-up obtained by using 2,3 and 4 cores on Harris are similar to the speed-ups of the GFTT feature detector which is reasonable since it is based upon the same fundamentals as GFTT. The 1 MB image provides the best speed-up, however, in the Harris case a speed-up of almost 4 instead of 3. Furthermore Test suite 1 and 2 of the Harris test are similar in the matter of speed-up behavior, but the 3rd test suite has a lesser peak speed-up at the 1 MB image.

The speed-up obtained utilizing four cores using the MSER feature detector show a different behavior from the other feature detectors. The speed-ups illustrate a reverse pyramid behavior, whereas the 32 KB image obtains a small super linear speed-up and the other images show a lesser speed-up. The trend is a speed-up to the 6 MB version of the images, and then a stall of the speed-up.

The speed-up of ORB illustrate a small pyramidic behavior with a peak at the

3rd size variation of each image. The speed-up the progressively decreases as the image size increases.

The Simpleblob speed-up illustrates a small speed-up as the image sizes increases. This is an on-going process as the speed-up is lowest at the smallest image variation and highest at the largest image variation. The exception is the test results from Img_2_{1MB}, which provides a slightly higher speed-up than the other 1 MB sizes.

The SIFT speed-up is the only algorithm which presents a close to consistent speed-up on all of the frames. Although the speed-up obtained from all frames is sub-optimal, the speed-up gained from SIFT is close to the same on the 32 KB version as the speed-up gained on the 24.6 MB version. This result suggest that SIFT is a scalable solution for every image size.

The SURF detector illustrates a behavior which originally expected for all algorithms, since the smaller images fit entirely in the L1 cache and potentially could be processed directly. SURF executes the 32 KB images at a super-linear which gradually decreases when the image size is increased.

7.4.2 Keypoints detected

OpenCV denotes features detected as keypoints. Due to the varying sizes of the images, there will be a variance in detected keypoints even though the algorithm in scale-invariant, simply because there are less pixels available. Table 7.4 presents the keypoints detected in each image variation for each algorithm. Since we are using the default settings of OpenCV, some algorithms use a threshold value of how many keypoints can be detected at max, this occurrence can be seen in the HARRIS, GFTT and ORB detectors.

As the number of detected keypoints increases with the image size, except for the algorithms which have a threshold value, we can conclude that the keypoint detection does not have a negative impact on the speed-up gained by an algorithm. This occurrence is especially clear in the FAST detector, which has a larger speed-up at the largest frame with 21253 (image 1), 71934 (image 2) and 142727 (image 3) keypoints detected than the smallest frame which only finds 330 (image 1), 280 (image 2) and 318 (image 3).

Image	Size	HARRIS	SimpleBlob	SIFT	SURF	ORB	MSER	GFTT	FAST	Dense	BRISK
1	32KB	90	0	55	61	50	24	276	330	324	13
1	128KB	110	0	281	285	358	29	737	1184	1225	72
1	256KB	217	0	450	644	453	59	1000	2211	2500	173
1	1MB	613	3	1502	2341	500	187	1000	7264	9801	565
1	6MB	1000	9	5632	10945	500	743	1000	23828	57121	2214
1	24MB	1000	38	16652	33346	500	1989	1000	51253	227529	5898
2	32KB	81	0	56	65	56	33	200	280	324	12
2	128KB	137	0	185	321	339	66	489	868	1225	51
2	256KB	203	0	433	545	428	97	720	1355	2500	108
2	1MB	593	7	1459	1769	500	198	1000	4443	9801	363
2	6MB	1000	9	3645	7856	500	614	1000	22833	57121	853
2	24MB	389	15	4824	28929	500	1021	1000	71934	227529	1154
3	32KB	100	0	80	93	59	35	199	318	324	18
3	128KB	347	0	245	385	370	63	763	1151	1225	86
3	256KB	497	0	455	710	461	97	1000	1824	2500	154
3	1MB	1000	12	1581	2399	500	276	1000	6212	9801	537
3	6MB	1000	70	6015	8288	500	1043	1000	10831	57121	1447
3	24MB	1000	133	27472	41037	500	3084	1000	142727	227529	6782

Table 7.4: Detected key points.

7.4.3 Execution time differences

We have measured the execution time of the program when it is run in parallel and compared it to a Sequential execution of the program to monitor any eventual losses in the execution time of the parallel program due to shared memory contention and overhead execution times. We executed this test using 4 different cores, introducing synchronization points between each core execution. The sequential version of our program is depicted in Fig. 7.7. Our sequential version of the program thus executes one image partition, running on one core before executing the next image partition on another core. The maximum execution time of the executing cores represent the execution time of the entire program, since a program is never faster than the slowest core. Each test was conducted 500 times to provide a median value.

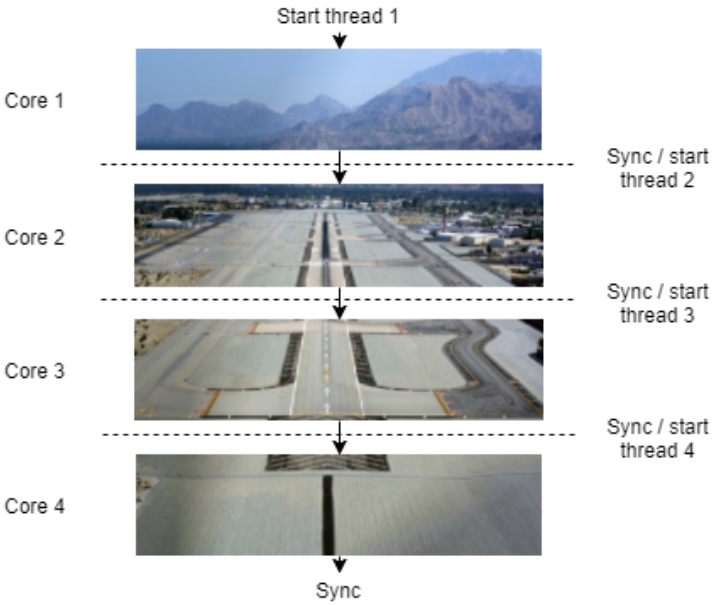

Figure 7.7: Sequential version of the test program.

We call the difference between our sequential execution and our parallel execution ΔT, which is calculated according to equation (7.6) where i is the core used, which are indexed starting from 0 and n is the number of cores used. tp is the median execution time using a parallel approach and ts is the median execution time using a sequential approach.

$$\Delta T = \{max(tp_i) : 0 \leq i < n\} - \{max(ts_i) : 0 \leq i < n\} \quad (7.6)$$

ΔT allows us to quantify how much of the program execution time is affected by utilizing a multi-core architecture. Fig. 7.8 illustrates the ΔT per core per image.

Fig. 7.8 depicts the ΔT on the y-axis using a logarithmic scale w the x-axis represents 3 test images, each one with 6 different image variations, separated with a gray field. The SURF algorithm performed worst in this test, with a ΔT of roughly 900000 microseconds compared to the sequential version using the largest image size.

FAST and Dense are the best overall algorithms according to the ΔT calculations, where the majority of the values are placed within the 80 microseconds range. There few outliers ranging 2300 microseconds using our largest image sizes which are small compared to the other algorithms.

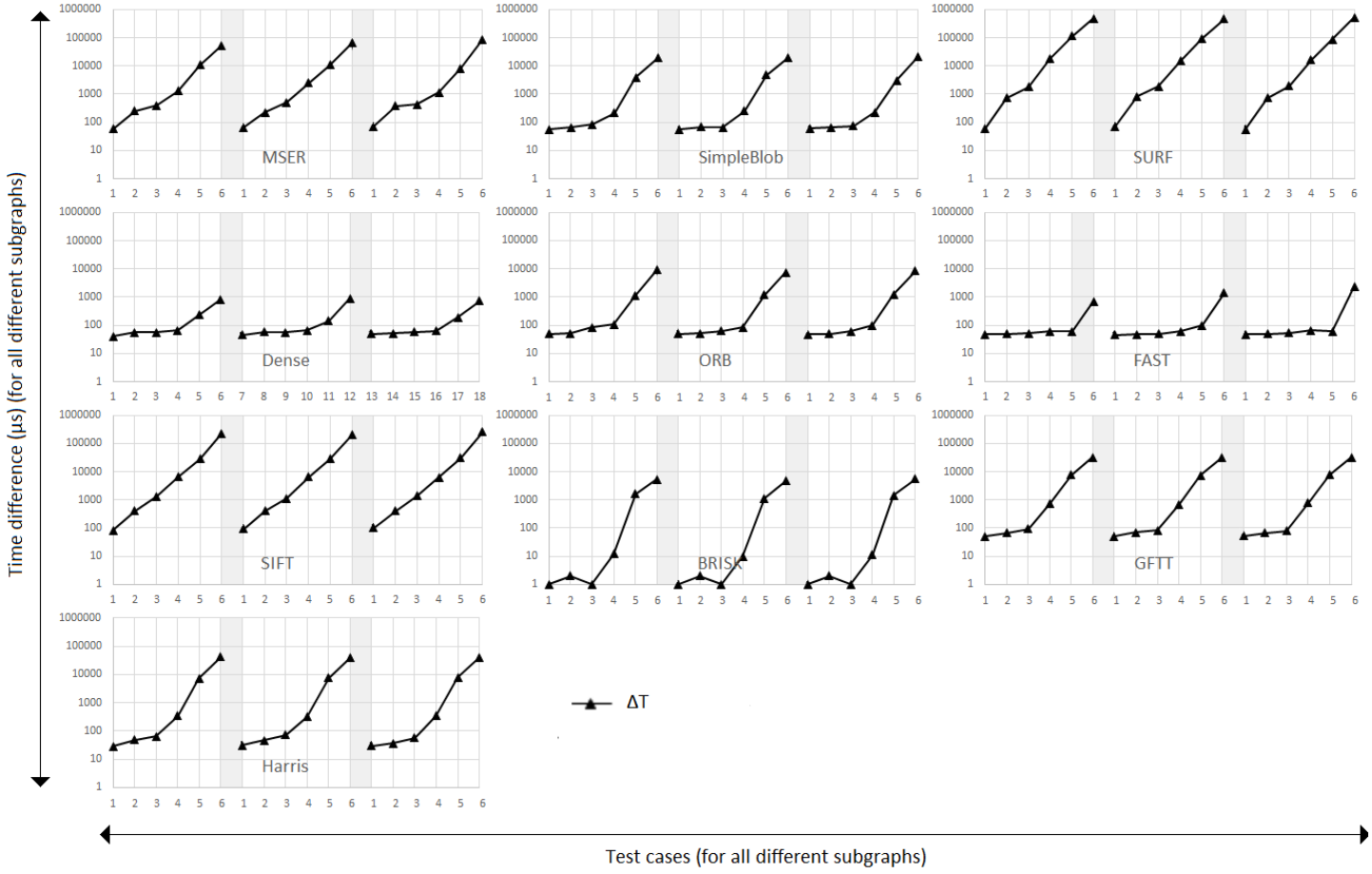

Figure 7.8: Differences in execution time using parallel and sequential approach.

7.4.4 Execution Characteristics

Given the different speed-up behaviors, there are certain events occurring within the hardware, which limits the size of the possible speed-up. We measured 16 different low-level metrics to investigate possible bottlenecks. However, the most important metric to measure is the first system-wide shared resource, which in this case is the L_3-cache, since it is the first shared resource with least amount of memory which makes it most likely to suffer from thrashing by other threads. We have chosen to visualize only the L_3-cache misses metric due to space limitations. Fig. 7.9 depicts the total amount of L_3-cache misses for both the sequential and parallel versions plotted on the left Y-axis, and the percentage deviation, denoted as ΔC plotted on the right Y-axis. The L_3-cache misses are the measured median values from 500 executions, while ΔC is calculated according to the total cache misses of all used cores when run in parallel divided by the total cache misses of all cores when run sequentially, denoted as ParallelMisses and SequentialMisses in equation (7.7).

$$\Delta C = \frac{ParallelMisses(C_{1..4})}{SequentialMisses(C_{1..4})} \tag{7.7}$$

The ideal value of ΔC is 0% L_3-cache difference which indicates that no thrashing has occurred. If thrashing occurs in the cache, the ΔC will increase. If the difference is negative, it means the memory is efficiently re-used by other threads and produces less L_3-cache misses than the sequential version.

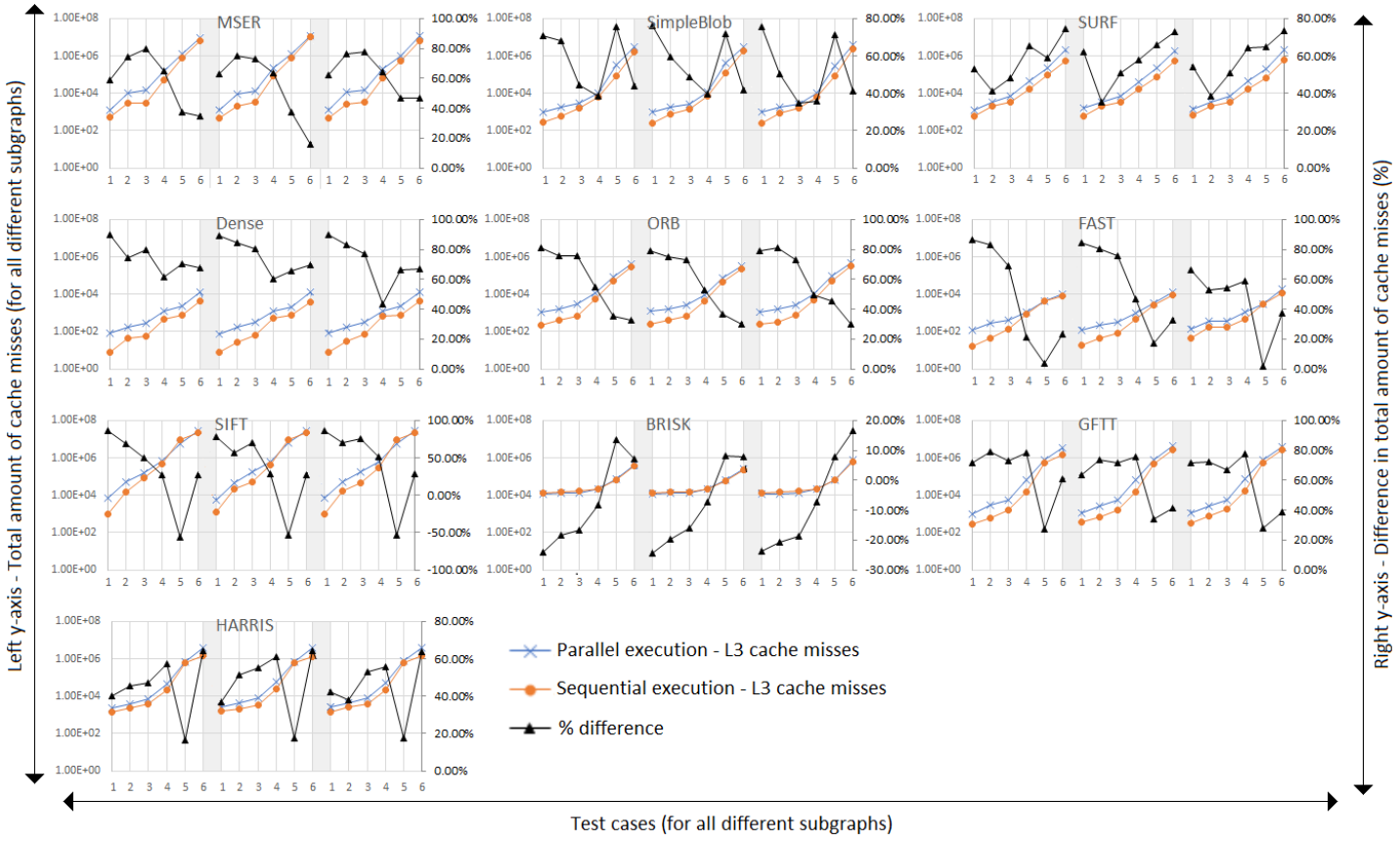

Figure 7.9: L3 misses using parallel and sequential version.

Compared to the other algorithms, FAST has a low L_3-cache usage, see Fig. 7.9, which is proportional to the amount of corners detected. We can also observe that FAST suffers a comparatively low amount of additional cache misses due to memory contention. The largest ΔC are in the smaller frames, but the difference in total is almost negligible. Since the speed-up of FAST is independent on how many cache misses are produced in L_3-cache, we can conclude that FAST is non-cache bound and therefore suitable for parallel executions.

Similarly to FAST, SIFT has a relatively low ΔC at the 6 MB image, which implies that SIFT re-use a lot of the data of the 6 MB variation of the image. The speed-up of SIFT remains unaffected by the ΔC indicating that SIFT is computationally heavy but is not memory bound.

The SURF algorithm has a relatively high ΔC, especially with larger image sizes. L_3-cache misses reveal an increase of 800000 misses in total using the parallel version compared to the sequential one. Concluding that SURF is cache bound is further strengthened by Fig. 7.5, which depicts an insignificant speed-up when executing on the largest image. It is debatable how much the increased amount of corners affect the speed-up; however, Fig. 7.8 reveals a ΔT of almost 1 Second for the largest images, suggesting that the amount of corners detected have small to possibly no effect on the speed-up.

The ORB algorithm has a fairly low ΔC for the larger images and also shows a low ΔT version compared to the other Object detectors. However, the ORB speed-up does not correlate at all with these facts, wherefore we can conclude that ORB is not L_3-cache bound.

The Harris and GFTT algorithms are similar in regards of Speed-up behavior, ΔC and ΔT. However, neither Harris nor GFTT receive a speed-up boost despite the fact that the L_3-cache misses difference is considerably lower for the larger image sizes which indicates that neither Harris nor GFTT are L_3-cache bound.

Dense has a high ΔC for all image variations. Although the total number of cache misses are low, we must also consider the execution time of Dense, which is also low. Since the Dense algorithm presents a ΔT of roughly 3000, it loses 2/3 of its potential execution time when using parallel version. Combining this with the fact that Dense has a high ΔC it is an indication that the Dense algorithm is L_3-cache bound.

BRISK shows a low ΔC as well as a low ΔT even though BRISK has a fairly bad speed-up at the larger images. Due to this fact, we can conclude that

BRISK is not L_3-cache bound.

The MSER algorithm can be considered L_3-cache bound due to the correlation between speed-ups gained in the larger images and the ΔC. In Fig. 7.6, we see a stall in speed-ups from $Img\_1_{6.1MB}$ to $Img\_1_{24.6MB}$ and $Img\_3_{6.1MB}$ to $Img\_3_{24.6MB}$. However, the figure shows a speed-up from $Img\_2_{6.1MB}$ to $Img\_2_{24.6MB}$.

A similar pattern can be detected in Fig. 7.9 whereas the ΔC differs by 40% in $Img\_1_{6.1MB}$, $Img\_1_{24.6MB}$, $Img\_3_{6.1MB}$ and $Img\_3_{24.6MB}$, but only differs 20% for $Img\_2_{24.6MB}$..

SimpleBlob has an irregular behavior according to ΔC. The differences for each test-case are common, but it is hard to find any correlation between the ΔC and the speed-ups gained. Simpleblob, however, has a high total amount of L_3-cache misses, and when adding the fact that SimpleBlob has relatively small ΔT compared to its extensive execution time (830000 microseconds), it indicates that SimpleBlob is not observably bound to the L_3-cache.

7.5 Conclusions

We have evaluated how default configured OpenCV feature-detection algorithms perform when using a data-partitioned parallel programming model for 2,3 and 4 cores. The algorithms performed differently using our data-set. The Harris algorithm obtained the highest speed-up at almost 4 times faster than the original single-core performance. However, this result depends heavily on the image size. SIFT was by far the most stable algorithm showing a speed-up of roughly 3 times the single core performance for all image sizes. SURF, on the other hand, received the worst speed-up, basically insignificant for larger images, which are the most computationally heavy. We have concluded that the parallelizing speed-ups of SURF, Dense, and MSER, are correlated to L_3-cache usage. Our measurements suggest that a system designer should not co-locate these algorithms with other L_3-cache bound tasks. We have also concluded that FAST, ORB, BRISK, HARRIS, GFTT, SIFT and SimpleBlob are not L_3-cache bound indicating that they can be efficiently utilized on multi-core systems, even though other tasks heavily load the L_3-cache. We further conclude that FAST, Dense, Harris, ORB, GFTT and BRISK all suffer from various degrees of overhead penalties when processing smaller frames.

7.5.1 Future work

We have used the default OpenCV parameters in this study, which mean that results from the feature-detection may differ due to different tuning. Therefore, further studies should try to find an optimal tuning for each frame and execute the the parallel feasibility tests described in our study. It is also possible to investigate the feasibility of co-executing feature detection algorithms on different cores. Running SURF which we concluded to be L_3-cache bound on one core and running FAST which is not L_3-cache bound on the three remaining cores could potentially be an efficient approach when the objective of a system is to detect both blobs and corners.

Bibliography

[1] M. Agrawal, K. Konolige, and M. R. Blas. Censure: Center surround extremas for realtime feature detection and matching. In *European Conference on Computer Vision*, pages 102–115. Springer, 2008.

[2] A. R. Alameldeen and D. A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, pages 8–17, 2006.

[3] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *Computer vision–ECCV 2006*, pages 404–417, 2006.

[4] G. Bradski. The opencv library. *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, 25(11):120–123, 2000.

[5] T. P. Chen, D. Budnikov, C. J. Hughes, and Y. Chen. Computer vision on multi-core processors: Articulated body tracking. In *Multimedia and Expo, 2007 IEEE International Conference on*, pages 1862–1865. IEEE, 2007.

[6] C. Ding, X. Xiang, B. Bao, H. Luo, Y. Luo, and X. Wang. Performance metrics and models for shared cache. *Journal of Computer Science and Technology*, 29(4):692–712, 2014.

[7] S. Eranian. What can performance counters do for memory subsystem analysis? In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, pages 26–30. ACM, 2008.

[8] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.

[9] S. Eyerman, K. Hoste, and L. Eeckhout. Mechanistic-empirical processor performance modeling for constructing CPI stacks on real hardware. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 216–226, 2011.

[10] H. Feng, E. Li, Y. Chen, and Y. Zhang. Parallelization and characterization of sift on multi-core systems. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 14–23. IEEE, 2008.

[11] C. Harris and M. Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Citeseer, 1988.

[12] F. Hosseini, A. Fijany, and J. Fontaine. Highly parallel implementation of harris corner detector on csx simd architecture. In *European Conference on Parallel Processing*, pages 137–144. Springer, 2010.

[13] M. Jägemar, A. Ermedahl, S. Eldh, and M. Behnam. A Scheduling Architecture for Enforcing Quality of Service in Multi-Process Systems. In Proceedings of Emerging Technologies and Factory Automation. *Analysis*, ETFA 2017.

[14] S. Leutenegger, M. Chli, and R. Y. Siegwart. Brisk: Binary robust invariant scalable keypoints. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2548–2555. IEEE, 2011.

[15] D. Levinthal. Performance Analysis Guide for Intel ® Core ™ i7 Processor and Intel ® Xeon ™ 5500 processors. *Intel Cooperation*, pages 1–72, 2009.

[16] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[17] J. Matas, O. Chum, M. Urban, and T. Pajdla. Robust wide-baseline stereo from maximally stable extremal regions. *Image and vision computing*, 22(10):761–767, 2004.

[18] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[19] Open Computer Vision. Common interfaces of Feature detectors.

[20] D. Patil, P. Kharat, and A. K. Gupta. Study of performance counters and profiling tools. In *Proceedings of* 21^{st} *IRF International Conference.*, pages 45–49, 2015.

[21] M. J. Quinn. Parallel programming. *TMH CSE*, 526, 2003.

[22] N. Rameshan, R. Birke, L. Navarro, V. Vlassov, B. Urgaonkar, G. Kesidis, M. Schmatz, and L. Y. Chen. Profiling memory vulnerability of big-data applications. In *Dependable Systems and Networks Workshop, 2016* 46^{th} *Annual IEEE/IFIP International Conference on*, pages 258–261. IEEE, 2016.

[23] E. Rosten and T. Drummond. Machine learning for high-speed corner detection. *Computer Vision–ECCV 2006*, pages 430–443, 2006.

[24] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. Orb: An efficient alternative to sift or surf. In *Computer Vision (ICCV), 2011 IEEE international conference on*, pages 2564–2571. IEEE, 2011.

[25] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer. Modeling performance variation due to cache sharing. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE* 19^{th} *International Symposium on*, pages 155–166. IEEE, 2013.

[26] J. Shi et al. Good features to track. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*, pages 593–600. IEEE, 1994.

[27] H. Sugano and R. Miyamoto. Parallel implementation of good feature extraction for tracking on the cell processor with opencv interface. In *Intelligent Information Hiding and Multimedia Signal Processing, 2009. IIH-MSP'09. Fifth International Conference on*, pages 1326–1329. IEEE, 2009.

[28] N. Zhang. Computing optimised parallel speeded-up robust features (p-surf) on multi-core processors. *International journal of parallel programming*, 38(2):138–158, 2010.

[29] Q. Zhang, Y. Chen, Y. Zhang, and Y. Xu. Sift implementation and optimization for multi-core systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

Chapter 8

Paper B: Resource Dependency Analysis in Multi-core systems

Jakob Danielsson, Tiberiu Seceleanu, Marcus Jägemar, Moris Behnam, and Mikael Sjödin. Testing Performance-Isolation in Multi-Core Systems In *2020 IEEE 44th Annual Computer Software and Applications Conference (COMPSAC). Vol. 1. IEEE, 2020*

Abstract

In this paper, we evaluate different methods for statistical determination of application resource dependency in multi-core systems. We measure the performance counters of an application during run-time and create a system resource usage profile. We then use the resource profile to evaluate the application dependency on the specific resource. We discuss and evaluate two methods to process the data, including moving average filter and partitioning the data into smaller segments in order to interpret data for correlation calculations. Our aim with this study is to evaluate and create a generalizeable methods for automatic determination of resource dependencies. The final outcome of the methods used in this study is the answer to the question: "To what resources is this application dependent on?". The recommendation of this tool will be used in conjunction with our last-level cache partitioning controller (LLC-PC), to make decision if an application should receive last-level cache partition slices.

8.1 Introduction

A multi-core processor contains multiple hardware resources, such as a memory management unit, instruction and data caches, translations lookaside buffers (TLB), I/O units such as GPIO's, direct memory access (DMA) controllers, etc. Processors may also have access to several different computational units such as the floating point unit (FPU), the arithmetic and logical unit (ALU) or the graphics processing unit (GPU). Most of these hardware resources are often shared between various computational cores. Several processes located on the same core will also share hardware resources. This can lead to a state called *shared resource contention*, where multiple cores or processes compete for using a hardware resource. Such contention can lead to significant performance degradation or timing unpredictably.

Applications in a computer always utilize a portion of the above-mentioned hardware and are thus *bound* to use these resources at certain times. *Resource-boundness* of an application has a large impact on many different aspects of a system, including how an application should be scheduled together with other applications, how vulnerable an application is to shared resource contention and also gives hints to what source of the shared resource contention the application is vulnerable to.

One possible solution to avoid shared resource contention is to use isolation techniques such as cache partitioning [16], TLB coloring [10], memory bus reservations [15], DRAM bank partitioning [14] and more. Isolation techniques create isolated hardware resource regions within the computer hardware for different cores or processes. One core or process is forbidden to use the hardware resource region of another process. Using such techniques often comes with an overhead performance penalty. It is therefore important to have a clear understanding of when it is beneficial to utilize hardware resource isolation techniques.

In this paper, we investigate methods for analyzing application resource boundness automatically, in order to be able to use the appropriate resource isolation techniques on workloads, in the future.

The rest of this paper is organized as follows: Section 8.2 gives the background information on resource boundness. Section 8.3 proposes a tool skeleton for identifying resource boundness. Section 8.4 shows the characteristics of our baseline measurement case. Section 8.5 discusses different statistical methods for evaluating resource boundness. We present our results and discuss them in Section 8.6, and also provide directions for future work.

8.2 Background

In previous work, we state the importance of finding the appropriate partitioning techniques to avoid wrong resources being partitioned and thus lose unnecessary performance [5]. We have furthermore investigated methods to find the saturation point partition (the point from which an application does not benefit more from additional resources, or increased resource size) of workloads to avoid over-commitment of certain resources such as the cache [3]. The question, however, still remains on how to make the definite decision during runtime on when a workload should be partitioned and what isolation techniques should be applied. While it is possible to create an understanding on where shared resource contention occurs using deductive reasoning [4], the question on making the definitive computer-understandable verdict on what resource boundness is remains.

8.2.1 Application performance

Performance is always related to time, but a full definition of *performance* is more complex, and it may be also unclear. The general term for performance is always that something is executed with a certain quality of service. The quality of service can be defined as many things, such as packets per second (common in network applications) and also total execution time (common in high performance applications). All code is translated into machine code, which are instructions that tell the computer what should be computed. All instructions travel through the memory hierarchy of a computer until, finally, reaching the processor which executes the calculations. The processor is, however, not the final point of actuation for an instruction. All modern processors utilize instruction level parallelism, which means that all instructions are split into smaller segments and executed within a unit called the *processor pipeline* - Figure 8.1 depicts the classic 5-stage RISC pipeline.

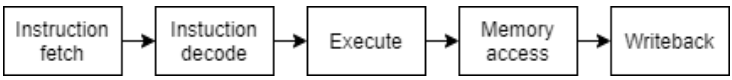

Figure 8.1: Illustration of the classic RISC pipeline

An instruction which has passed the *writeback* stage has passed through the entire pipeline and is now considered completed, or commonly called *retired* by different processor manufacturers such as ARM and Intel. In high performance systems, it is desirable to have as many instruction retired as possible

in a certain time-frame, as a measure of performance.

There are however exceptions to the previous performance measurement rule, which comes with busy-wait programs such as network applications, which measures performance using the packets per second metric. Here, we are given a precise performance indicator, mentioning how many packets two nodes are able to send and receive in a given time-frame. The packets per second metric is based on a sender-receiver scheme where one of the nodes (node A) is the sender, and the other node (node B) is the receiver. Node A will continuously send packets at its maximum capacity while node B will be stuck in a busy-wait loop, doing nothing until a packet has arrived. Since node B is stuck in the loop, the amount of instructions retired will be un-proportionally high, compared to the packets per second, which means that in this typical case - instructions retired is not a feasible performance metric.

8.2.2 Resource boundness

The number of instructions retired can be halted by numerous occurrences within the computer. Such occurrences include CPU suspends while waiting for memory to become available and stalls within the pipeline which happen as a consequence of branch mis-predictions and data hazards. Investigating which resource an application is bound to is, therefore, important, since it enables the system designer to make decisions on how different applications should be handled in the system.

The amount of instructions retired in a given time-frame can become highly dependent on what happens in the memory hierarchy of the computer. For instance, if the memory used by an instruction is not immediately available (i.e., located directly in the processor memory) it needs to be fetched from one of the upper memory layers within the processor, which in turn has its own hierarchy, including the L_1D-cache, L_2-cache and L_3-cache, where the L_1D-cache is fastest and L_3-cache is slowest. Measuring how the performance is bound to a certain resource becomes especially interesting when working with multi-core computers since it enables system architects to adjust the affinity of certain applications which may not fit together on the same core. It is possible, for instance, to assume that applications which are bound to the L_1D-cache should never be placed on the same core since it means these applications potentially can replace each other's cache lines.

Figure 8.2 shows our assumptions on applications allocated on the same core, depending on their resource boundness. In the figure, we use three resources to

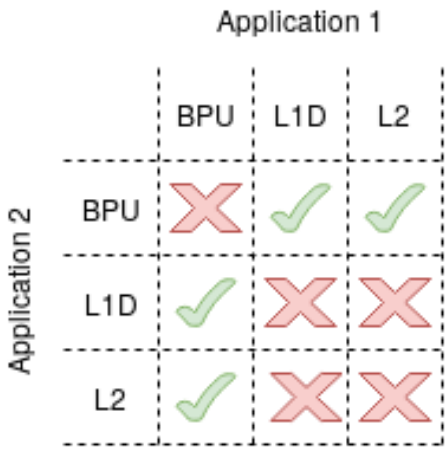

Figure 8.2: Illustration of the classic RISC pipeline

exemplify: branch-prediction unit (BPU), L1 data cache and L2 cache. Crosses in the grid marks that the applications should not be allocated to the same core, while check-marks show that they may. Our assumptions in figure are that it is never a good option to keep two applications which are bound to the same resource on the same core, due to risk of resource contention. For instance, in a hyper-threaded system - where two applications running on two different threads, but still on one core, utilize the same L_1D-cache. If these two applications utilize the local L_1D-cache heavily, they could potentially replace the data of each-other due to the cache replacement mechanisms which cause resource contention. Resource contention makes it a bad suggestion to map L_1D-cache-bound applications to the same core. Similarly, if two applications heavily utilize branches, they put the branch predictor unit to stress since the branch predictor looks at current branch trends and then makes predictions based upon these trends. If two applications contest the branch predictor, there is a risk that the branch predictor may get confused by the branches taken of to independently running applications. The L_2-cache has similar properties to the L_1D-cache, but hosts a greater amount of memory. Two applications bound to the L_2-cache should therefore not be allowed on the same core, unless we are ready to accept relatively heavy losses in performance. We also argue that it might be favourable to not allow for L_1D-cache bound and L_2-cache applications to co-exist on the same core, since L_1D-cache misses directly lead to accesses in the L_2-cache. If the L_2-cache is already working at maximum capacity, more L_2-cache accesses which happens as a consequence of L_1D-cache misses means that the L_2-cache will present even more L_2-cache misses.

Similar considerations should be addressed about the inter-core shared resources, such as the last level cache and the TLB. Since these resources are shared across multiple cores, it is not possible to avoid resource contention through executing applications with LLC- or TLB-related boundness on different cores. However, it is possible to enforce partitioning policies such as cache coloring [13] and TLB coloring [10].

8.2.3 Profiling resource boundness

It is common for large scale industrial applications to have thousands up to millions of lines of code which makes them very hard to analyze in a white-box manner. Therefore it is very temping instead to analyze such applications using a black-box perspective and assume that there is no prior knowledge of the application behavior, and run from that perspective. It is possible to analyze applications in such a manner by using performance counters, which are powerful computer hardware peripherals that measure hardware triggered events, such as branch mis-predictions, cache misses, TLB misses, etc. Using performance counters enables us to create hardware profiles of the applications, without any prior knowledge on their behavior.

8.2.4 Considered resources

Every resource within a multi-core processor can be the origin of shared resource contention. Some resources are, however, more prone to contention than others. Our goal is to create a generalizeable method for characterization of all different computer resources. In this paper we limit, though, the approach to the resources contained within the internal memory subsystem of a processor, due to the limited number of physical hardware counters. We present the targeted resources in Table 8.1.

Performance counter	**Description**
L1_TCM	Total amount of level one cache misses
L2_TCM	Total amount of level two cache misses
L3_TCM	Total amount of level three cache misses
PAPI_TLB_DM	Total amount of data TLB misses
PAPI_SP_OPS	Single point floating operations
PAPI_BR_INS	Amount of branch mispredictions

Table 8.1: Measured performance counters

We believe the items presented in table 8.1 represent some of the most important ones when it comes to resource contented systems. The different performance counters were selected as there are only a number of actual hardware counters on each chip.

We use the *instructions retired* as our performance metric, measured by the performance counter. This further means that it is possible to measure the performance of applications completely ad-hoc, without any complex communication schemes or additional implementation within an application. Measuring the instructions retired, however, limits the application usage, since instructions retired is only one of the relevant performance metrics of continuously running applications, which are not dependent on wait states (such as waiting for I/O's).

8.2.5 Related work

Works including the Scarphase profiling tool [6] are closely related to our research. There are two papers written by Sembrandt et al. [12] discusses how applications can be split into several phases and show the resource usage over time for different processes. In another paper, Sandberg et al. [11] discusses a method for measuring the performance variability of applications due to cache contention. Similarly to our work, they also utilize the performance counters to identify the different phases and can as such visualize the resource usage of specific applications over time. Our work takes this investigation one step further, since we also account for the performance aspect of applications and try to automatically determine how the performance of an application is bound to a specific performance counter.

Other close related work is the characteristics monitor (*Charmon*), developed by Jagemar et al. [7]. Charmon creates advanced profiles of applications using the *perf* interface performance counters and correlates them with a performance metric called System Level Metric (SLM). SLM is a general term for performance of an application, and works more on a system level, where packets per second is an example. This approach is very accurate when the definition of performance is clearly specified, the aim of the application is single objective and also accounts for busy-wait application. Our approach omits busy-wait applications but instead makes it easier to characterize multi-objective applications.

8.3 Method

We propose a three-step methodology for automatically determining the resource-boundness of an application. We describe the characterization procedure in Alrogrithm 1.

```
Initialize_PAPI();
while forever do
    /* Determine the required sampling frequency
       */
    timestamp_before();
    execute_process(application.elf);
    timestamp_after();
    sample_frequency = timestamp_before-timestamp_after;
    /* Fork out process to different thread and
       sample performance counters              */
    pid = fork(application.elf);
    while pid==active do
        measure_performance_counters(pid);
        sleep(sample_frequency);
    end
    /* Calculate characteristics dependencies
       and store them                           */
    calculate_pearson_correlation(pid);
    store_application_characteristics();
end
```

Algorithm 1: LLM-tool pseduocode

We base our algorithm on three steps: firstly identification of sampling frequency; secondly sampling of certain performance counters; finally resource boundness calculations, which quantify how much dependency an application has on a certain performance counter. We list the steps in more detail as follows:

1. *Identification of sampling frequency*. In order to determine how the performance of application moves with a performance counter event, we need a certain amount of samples. We determine the sampling frequency through running the application under test once, and thus determining the execution time. We then divide the execution time with a quota depending on how many samples we need.

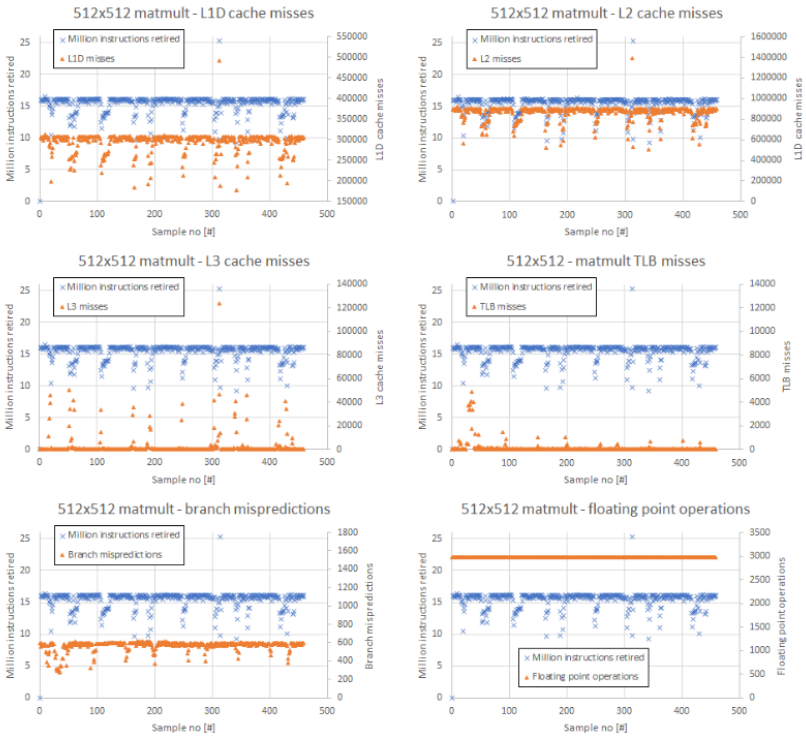

Figure 8.3: Execution characteristics of a 512x512 matrix multiplication

2. *Performance counter sampling*. We sample the performance counters events at the previously defined frequency. All performance counter measurements are stored inside a *struct*, which will be used as a database when calculating the resource boundness of an application.

3. *Resource boundness estimation*. We calculate the resource boundness of an application in our final step using the *Pearson correlation coefficient*.

8.4 Characterizations

In this section, we present experiments to automatically visualize the characterization of an application. Our test platform is specified in Table 8.2.

Feature	Hardware Component
Processor	4xIntel® Core™ i5-8850H CPU (Skylake) 2.6GHz
L_1-cache	32 KB 8-way set assoc. instruction caches/core + 32 KB 8-way set assoc. data cache/core
L_2-cache	256 KB 4-way set assoc. cache/core
LLC	9 MB 12-way set assoc. shared cache
MMU	64 Byte line size, 64 Byte Prefetching, DTLB: 32 entries 2 MB/4 MB 4-way set assoc. + 64 entries 4 KB 4-way set assoc., ITLB: 128 entries 4 KB 4-way set assoc., L_2Unified-TLB: 1 MB 4-way set assoc., L_2Unified-TLB: 512 entries 4 KB/2 MB 4-way assoc.

Table 8.2: Hardware specifications Intel® Core™ i578850H

We use a sample size of 500, which means the periodicity sampling frequency of the performance counters is calculated according to Equation 8.1. We furthermore use the *unistd* Linux library for the sleep timers, which means the execution time of must be measured at the granularity of microseconds.

$$Frequency = \frac{Execution\_time(\mu s)}{Sample\_size} \tag{8.1}$$

Our assumptions when executing a characterisation test is that an application is only bound to a resource when the performance decreases - while the performance counter of the resource increases. We present our definition of resource boundness as follows.

Definition - Resource boundness The *performance* of an application is represented by the number of instructions retired from the pipeline.

A *performance counter event* is represented by an event occurring within a resource of the computer.

When the application suffers performance degradation and at the same time shows an increase in performance counter events, the performance degradation is a direct consequence of the usage of that particular resource.

In previous works [5], we established that a matrix multiplication is bound to the last-level cache and vulnerable to shared resource contention. Since

we know the boundness of a matrix multiplication, we use it as our visualisation example in this paper. Figure 8.3 shows the execution characteristics of a 512x512 matrix multiplication. Each sub-graph plots the instructions retired on the left-hand side y-axis (which is our performance metric), marked with blue crosses. The left-hand side y-axis plots the respective performance counter, see table 8.1 for full description. The x-axis shows the sample number from which these measurements were taken.

We plot the instructions retired on all different sub-graphs, to provide a better visual representation on what resource the matrix multiplication is bound to.

Our definition of resource boundness makes it possible to exclude four measurements as candidates for the resource boundness. The first and second sub-graphs plot the L1D and L2 cache misses, respectively. These two metrics show a direct correlation with the performance - when the performance of the matrix multiplication decreases, the performance counter even also decreases. This is called a positive correlation in statistical terms, and it is precisely what happens when an application is not bound to this particular resource. When the performance counter decreases with the performance, it means the resource is currently operating at maximum capacity and is hindered to performing better by overlying resources such as the L_3-cache and TLB.

The TLB misses show another trend, with a significant spike in the starting phase of the matrix multiplication execution. This spike, however does not last or re-occur later in the program execution and can, therefore, be explained as a new process requesting sufficient address space at the start.

Our most interesting metric is the L_3-cache misses metric, which shows frequent spikes throughout the entire program execution. While these spikes occur, we also see a significant drop in performance, which indicates that the L_3-cache usage has strong claims as being the primary resource bottleneck for the matrix multiplication.

One very interesting data-point also presents itself at point 313 on the y-axis. At this point, the millions of instructions retired has suddenly increased by 67% together with its respective cache metrics. This is an *outlier* value and could have various explanations, including kernel interrupts, or measurement error due to process suspension, or double value measurement, etc. It is possible to just filter this value out and as such just ignore it by using certain thresholds. However, in our methodology, there is the risk that many of our values can be seen as outliers which makes the threshold assignments too arbitrarily. Referring to the L_3-cache misses graph in Figure 8.3, where there are 4 spike values in the beginning - these highly important values could also be

seen as outliers and if we ignore these important values, the resource boundness calculation loses its purpose.

In the following sections we will evaluate how to approach outliers vs, important data spikes.

8.5 Discussion of applicable methods

We have established that the performance of the matrix multiplication shows most correlation to the amount of L3 cache misses produced. These conclusions were however drawn using visual observations of data - not possible to execute automatically in a computer. In order to automatically determine the resource boundness, we will use statistical methods.

In the following, we discuss methods that can be utilized to confirm that the matrix multiplication is L3 cache bound.

8.5.1 Distribution of data

It is important to discuss the data characteristics before analyzing what methods are applicable, because the shape of the curve can impact on how different correlation methods can be used. The majority of the L_3-cache misses measurement data lies in an interval between 100-1000 L_3-cache misses. Figure 8.4 plots the data distribution intervals, where the left-hand side bar shows the amount of samples with a value less than 1000. The right-hand side bar shows the number of samples with a value over 1000. The median amount of L_3-cache misses is 318 and has a standard deviation of 8225.

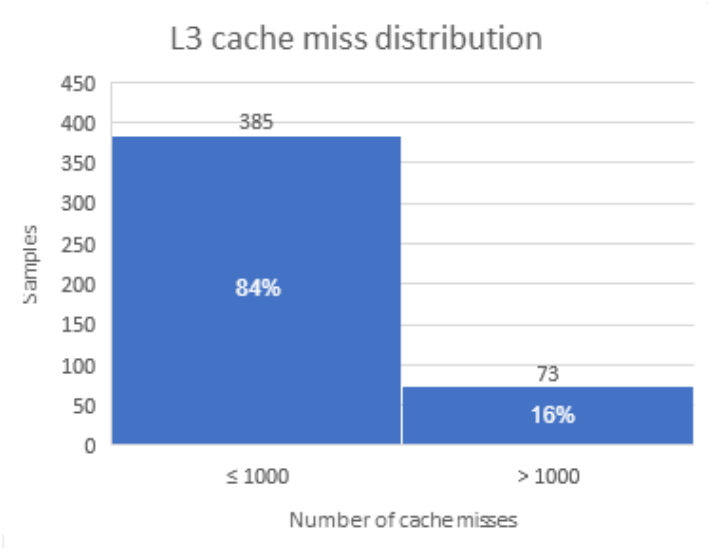

Figure 8.4: Distribution of L_3-cache misses

The data distribution shows that the program execution is characterized by a relatively low L_3-cache misses value, which represents 84% of all measurements. Measurements above 1000 cache misses only represent 16% of the measured data points. The distribution of data means we have a relatively smooth line with large value spikes at certain intervals which happens when the cache memory becomes full. These large value spikes are significantly higher than the median value, which is the reason for such high standard deviation.

8.5.2 Filtering interesting data points

Filtering is a very powerful tool to smoothing out curves and can be very useful when using correlation techniques, especially when there are outliers that can make the correlation calculation inaccurate. Through filter usage, it is possible to ignore data-points which truly are outliers. However, filtering is also something that must be used with utmost caution. Since we are interested in the relationship between two curves with many potential abrupt values, there is a risk that too heavy filtering may classify certain spikes as outliers and just ignore them.

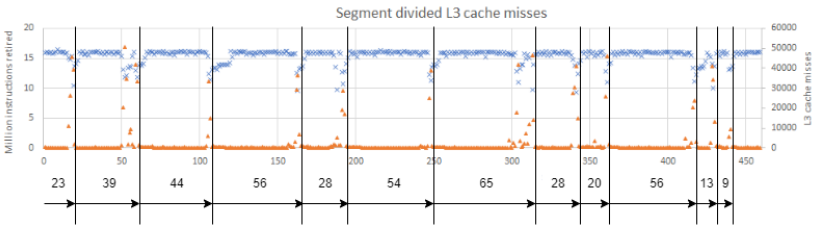

Figure 8.5: Segment divided L_3-cache misses measurements

8.5.2.1 Segmenting the curve

Here, we discuss two filtering techniques, applicable to our case. We firstly discuss partitioning the curve into several sub-curves and analyzing the sub-curves individually. We also apply a moving average filter to smoothing out the curve and filter out extreme outliers.

Our first technique to filter the curve is to split the measurements into several different segments using threshold values. We use the standard deviation of 8225 L_3-cache misses as threshold value for identifying deviations of the curve. Based on these, we then divide the curve into segments and can thus investigate the segments individually rather than investigating the curve as whole. A segment of the curve is characterized as follows.

Firstly, there is a "silent" phase, where the L_3-cache misses lie below the standard deviation value. When the L_3-cache misses measurement trespasses the standard deviation, a burst phase starts, which shows a significant increase in L_3-cache misses. The burst phase ends once the L_3-cache misses measures under the standard deviation once again, and thus marks the end of a segment.

Figure 8.5 enlarges the L_3-cache misses graph from Figure 8.3 and plots the value intervals between different curve partitions.

The reason for employing this technique is to analyze smaller, key-parts of the application execution, rather than the complete application behavior. Once all key-parts are identified, we can calculate the correlation coefficient for each individual key-part. We then use the correlation coefficients of all intervals to calculate the average and median correlation coefficient values to represent the whole curve.

8.5.2.2 Moving average

Smoothing out the curves could make correlation coefficients more susceptible towards the data input, and less vulnerable to extreme outliers. Figure 8.6 plots the moving average for the instructions retired and L_3-cache misses respectively.

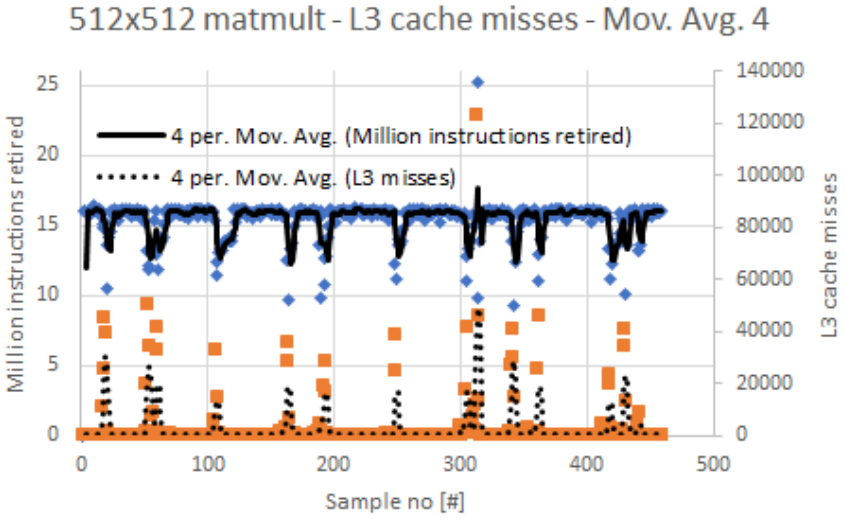

Figure 8.6: L_3-cache curve divided into several segments

The idea of adding a moving average filter to the curve supports smoothing out the rapid changes within the curve, such that the correlation coefficients still recognize these changes. Applying moving average filters can however become dangerous, since there is a risk that important deviations in the dataset are ignored.

8.5.3 Relationship evaluation

In this section we evaluate the standard *Pearson correlation coefficient* [2] - which is highly sensitive to outliers [8] - of our three different methods. There are other correlation methodologies which potentially may be used in our case, such as rank based correlations Kendall [1] and Spearman [9]. These correlations are, however, not as sensitive to outliers as the Pearson correlation, and therefore our approach only covers the Pearson correlation coefficient.

There are two types of correlations. *Positive correlation* means that the curves of two data-sets move in the same direction. In our case, a positive correlation is shown between the L_1D-cache misses and the instructions retired - when the amount of instructions decreases, the amount of L_1D-cache misses also decreases.

The second type is the *negative correlation*, which is shown in in the L_3-cache misses case. Negative correlation means that the curves of two-data sets move in opposite directions - when the amount of L_3-cache misses increases - the amount of instructions retired decreases.

Correlation coefficients provide excellent values for for detecting such relationships and when used in conjunction with absolute values, it is also possible to quantify the magnitude of the trend. An absolute correlation value between 0.2-0.3 is seen as low correlation, and a value of 0.3-0.5 gives a medium correlation. A high correlation comes at values larger than 0.5.

Table 8.3 lists the calculated correlation coefficients using our baseline measurement, segmented measurement and moving average curve. The baseline correlation is just a straight-off correlation coefficient between the instructions retired and L_3-cache misses. The segmented correlation value is based on the average and median correlation coefficients from all segments of the application execution. The final measurement implements a moving average filter to smooth out the curve. The table shows the correlation coefficient depending on the size of the moving average window.

Type	Correlation coefficient
Baseline	-0,3664
Segmented Average	-0,6751
Segmented Median	-0,7695
Mov. Avg. Size 2	-0,4990
Mov. Avg. Size 3	-0,5452
Mov. Avg. Size 4	-0,5706
Mov. Avg. Size 5	-0,5874
Mov. Avg. Size 10	-0,6378

Table 8.3: Correlation coefficients for each method

The table presents the correlation coefficients for each methodology. The baseline measurement, which is the untouched data-set presents a correlation coefficient of 0.3664, which is relatively low, considering how the performance actually moves with the L_3-cache misses. The explanation for this is the ex-

treme outlier in data-point 313, where both the L_3-cache misses and the million instructions retired peaks at an abnormal value.

The moving average calculation presents an increase of correlation, because the worst outliers are smoothed out. A moving average with a window size of 10 presents the highest correlation of 0.6, which is relatively a high value. The question which arises with moving average is, however, to what extent is important data filtered out? Through visual inspection, we come to the conclusion that a moving average of window size 10 has filtered out our worst outlier(s), but many other spikes have also been removed.

Our segmented methodology presents the highest correlation with a value of 0.7695. Comparing it to the above, it would require a substantial increase of the window size with the moving average approach, to reach the same correlation, taking us to a point where values in the curve are almost meaningless.

We argue that using a segmented curve correlation is the best for analyzing since it is usable for also analyzing applications which binds to multiple resources. An example of such an application would be to find the highest value inside a matrix after a matrix multiplication has been performed. The application will firstly have a highly L_3-cache dependent phase during the multiplication of the matrices. Once the matrix multiplication is done, a comparison phase will start, which instead will be highly dependent on the branch prediction unit. It is possible to adapt the segmented correlation to detect such shifts in resource boundness of multiple phases in an application, wherefore it is favourable to use our segmented methodology.

8.6 Summary

We have shown how the performance of a matrix multiplication interacts with several low level resources such as the caches, the TLB, the branch prediction unit and the floating point unit. We have done an extensive evaluation using correlation tests on our measurement data to determine how to filter out outliers from these measurements. Both our filter methods show a significantly greater relationship between the instructions retired and L_3-cache misses compared to just executing correlation coefficient calculations on baseline measurement data. This means, it is favourable to process L_3-cache boundness measurement data with filter methods instead of just correlating the baseline values. Our suggested method of partitioning the data into several segments revealed to show the highest correlation relationship, since we are able to look at multiple, smaller phases of an application and as such decreasing the granularity of correlation calculations rather than just looking at the data set as whole.

8.6.1 Future work

In future work, we will focus on implementing the segmented curve combined with a Pearson correlation coefficient to provide a fully-fledged automatic resource-boundness analysis tool for workloads. Our aim is then to integrate this tool with last level cache partitioning, such that only workloads which need last level cache partitioning will receive cache partition slices. Other interesting directions for future work would be to implement neural networks into the determination process, which possibly could provide an even better estimation of workloads. Our reason for not investigating neural networks in this paper was that we not had precise definition on when an application is resource bound, and could therefore, not train the neural network properly. Now, however, all the necessary tools are available.

Bibliography

[1] H. Abdi. The kendall rank correlation coefficient. *Encyclopedia of Measurement and Statistics. Sage, Thousand Oaks, CA*, pages 508–510, 2007.

[2] J. Benesty, J. Chen, Y. Huang, and I. Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[3] J. Danielsson, M. Jägemar, M. Behnam, T. Seceleanu, and M. Sjödin. Run-time cache-partition controller for multi-core systems. In *IECON 2019-45th Annual Conference of the IEEE Industrial Electronics Society*, volume 1, pages 4509–4515. IEEE, 2019.

[4] J. Danielsson, M. Jägemar, M. Behnam, M. Sjödin, and T. Seceleanu. Measurement-based evaluation of data-parallelism for opencv feature-detection algorithms. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 701–710. IEEE, 2018.

[5] J. Danielsson, T. Seceleanu, M. Jägemar, M. Behnam, and M. Sjödin. Testing performance-isolation in multi-core systems. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 604–609. IEEE, 2019.

[6] https://github.com/uart/scarphase. Scarphase tool.

[7] M. Jägemar, S. Eldh, A. Ermedahl, and B. Lisper. Towards feedback-based generation of hardware characteristics. In *7th International Workshop on Feedback Computing*, 2012.

[8] M. Joseph. Pearson coefficient of correlation explained.

[9] L. Myers and M. J. Sirois. Spearman correlation coefficients, differences between. *Encyclopedia of statistical sciences*, 12, 2004.

[10] S. A. Panchamukhi and F. Mueller. Providing task isolation via tlb coloring. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 3–13. IEEE, 2015.

[11] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer. Modeling performance variation due to cache sharing. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE* 19^{th} *International Symposium on*, pages 155–166. IEEE, 2013.

[12] A. Sembrant, D. Black-Schaffer, and E. Hagersten. Phase behavior in serial and parallel applications. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 47–58. IEEE, 2012.

[13] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: a dynamic cache partitioning system using page coloring. In *Parallel Architecture and Compilation Techniques (PACT), 2014* 23^{rd} *International Conference on*, pages 381–392. IEEE, 2014.

[14] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 155–166. IEEE, 2014.

[15] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE* 19^{th}, pages 55–64. IEEE, 2013.

[16] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102. ACM, 2009.

Chapter 9

Paper C: LLM-shark – A Tool for Automatic Resource-boundness Analysis and Cache Partitioning Setup

Jakob Danielsson, Tiberiu Seceleanu, Marcus Jägemar, Moris Behnam and Mikael Sjödin. In 45^{th} *Computer Society Signature Conference on Computers, Software and Applications (COMPSAC)*. IEEE, 2021.

Abstract

We present LLM-shark, a tool for automatic hardware resource-boundness detection and cache-partitioning. Our tool has three primary objectives: First, it determines the hardware resource-boundness of a given application. Secondly, it estimates the initial cache partition size to ensure that the application performance is conserved and not affected by other processes competing for cache utilization. Thirdly, it continuously monitors that the application performance is maintained over time and, if necessary, change the cache partition size. We demonstrate LLM-shark's functionality through a series of tests using six different applications, including a set of feature detection algorithms and two synthetic applications. Our tests reveal that it is possible to determine an application's resource-boundness using a Pearson-correlation scheme implemented in LLM-shark. We propose a scheme to size cache partitions based on the correlation coefficient applications depending on their resource boundness.

9.1 Introduction

The internal memory subsystem of a processor is often limited, cache memories for instance, often host a memory area that ranges from two digit KB's to single digit MB. The small memory space means it is improbable that an application's entire memory footprint can fit within one of the caches. As such, the caches will, at some point during application execution, become full. When new memory is requested, and when the necessary memory block is not present within the cache, it is instead fetched from the main memory and brought into the cache, replacing old data. Fetching the DRAM data produces a significant delay and will produce processor pipeline stalls while waiting for the memory block to become available.

Modern computers often utilize multi-core processors to increase throughput. The multi-core processors typically implement a shared cache policy, where at least one cache is shared between all cores. When multiple applications execute on different cores while sharing the same cache, there is a high risk that one application's memory requests will repeatedly replace another application's data, causing a resource contention scenario called *cache contention*. Cache contention causes execution-time jitters and performance degradation [7].

We can also find contention in other resources such as the Translation Lookaside Buffer (TLB) [15], the memory bus [24] and even the DRAM [23], but in this paper we focus on the cache related issues. Several mitigation techniques for cache contention exist, including page coloring [22] (a memory allocation scheme implemented in the memory management unit), cache way-partitioning (provided by the hardware manufacturer), and cache locking schemes [21]. The mitigation techniques improve the execution-time jitters, at the cost of implementation complexity and/or execution-time overhead. Due to the disadvantages, it is desirable to only use mitigation techniques when necessary, i.e., when there is a risk for cache contention that would degrade the performance of the system.

Determining the necessity of a mitigation technique is not a straight-forward process, since engineers have to carefully investigate the run-time behavior of each individual application and may also have to inspect the application code. Due to the code complexity of many modern applications this can quickly turn into a time-consuming procedure.

This paper describes our tool, LLM-shark, that determines partitioning techniques and partitioning sizes. LLM-shark monitors the run-time behavior of

applications using hardware performance counters and creates a characteristics profile of an application. We use the characteristics profile to determine how much the performance of an application depends on a certain resource. We make a resource-boundness estimations using the Pearson-correlation coefficient and then suggest the usage of specific mitigation techniques. We, furthermore, utilize the correlation coefficient to determine how much memory the selected mitigation technique must assign to the application. The main contributions in this paper are:

- A method to quantify resource-boundness automatically.
- An evaluation of how applications perform in a cache restricted environment and how the results are in line with our resource-boundness estimation.
- Automatic calculation of the amount of needed cache memory based on the above estimation.
- An implementation of the above aspects in the tool LLM-shark, and a proof of concept study showing the feasibility of the tool.

9.2 Background

9.2.1 Performance counters

Many architectures implement a performance monitoring unit (PMU) [1] to monitor hardware resource usage. The PMU follows a large set of events, including CPU pipeline resources, various internal memory events such as cache/TLB events, and also off-core events such as DRAM accesses and interrupts [25]. The PMU utilizes hardware-implemented performance monitor counters (PMC) that increments each time a specific event occurs. Modern processors typically contain several PMC's to simultaneously measure a set of different events. Our test environment processor is an Arm Cortex-a53 CPU, with capabilities for six simultaneous PMCs measuring six different hardware events simultaneously. In this paper, we utilize the Performance API (PAPI) [14] which is a front-end framework for the standard Linux API for performance counters - Perf [9]. PAPI has extensive default support for the branch unit and the last-level cache.

9.2.2 Resource-boundness

In the context of our paper, we define the performance of an application as the number of instructions completed per unit of time. In Eq. 9.1, P denotes the performance metric, I_t the number of instructions retired of an application and t as the time interval.

$$P = \frac{I_t}{t} \tag{9.1}$$

We use the definition from Eq. 9.1 to trace an application's performance through a PMU event called *instructions retired*, which increments each time an instruction leaves the final write-back stage of the processor pipeline. Thus, we define the application performance as the number of instructions retired at a specific sampling frequency, a higher number meaning higher performance. This definition is not suitable for all types of applications, such as network applications that heavily utilize tight and small busy-wait loops, causing a high instructions retired rate with no perceived system-level performance. The performance metric for such applications instead is packets per second or similar [11] system-level metrics. In this paper, we target non-I/O-bound applications, for which the above performance definition is applicable.

An application is typically built of millions of instructions where each instruction takes at least one clock cycle to complete, assuming an in-order, non-super-scalar processor. In ideal conditions, the processor executes an instruction without any delays, which means a one-cycle instruction will take one cycle to finish, a two-cycle instruction will take two cycles to finish, etc. An application running on a 1.2GHz processor, without any external disturbances will thus execute instructions equivalent to 1.2 billion cycles per second.

It is however unrealistic for an application to execute instructions close to the processor clock frequency, due to cycle-disturbances, such as branch mispredictions, structural data hazards, memory wait operations and also operating system overheads. All cycle-disturbances causes result in stall penalties within the processor pipeline and means that an application's instruction is not allowed to execute for a certain number of clock ticks. One common source for disturbance is the register memory itself, which is typically very small and can therefore not host the complete application data set. When the register memory does not contain requested data, the data needs to be fetched from L_1D-cache and a one cycle penalty stall will be inserted into the processor pipeline, which halts the processor from executing the instruction. This cycle stall penalty, thus, halts the application from executing an instruction, which

means that the application will suffer a performance degradation.

The cycle stall penalty varies depending on the hardware unit. Application data that is not present within L_1D-cache enforces an even greater pipeline stall penalty (e.g., 10 cycles) and needs to be fetched from the L_2-cache, etc,. Similar stall penalties are also present in other various computer components such as the branch predictor unit (BPU) and the Translation Lookaside Buffer (TLB). An application with a high stall cycle penalty count is able to execute less instructions per time interval than an application which contains few stall cycle penalties. Thus, an application's performance builds a dependency towards the stalls, where more stalls infer a decrease in performance.

The resource that causes the most cycle stalls to an application causes the greatest effect on the applications' performance and therefore presents the strongest resource-boundness. Identifying an applications' resource-boundness becomes of great importance in multi-core systems, since some resources are physically shared across different cores. Two applications that display noticeable resource-boundness towards the same shared resource such as the Last-level cache can lead to cache contention, causing both applications to suffer from (potentially severe) performance degradation.

9.2.3 Cache partitioning

Page-coloring is a software approach to partition a cache for mitigating cache contention. Page-coloring creates an allocation scheme for free pages and assigns the pages to a fixed position within the cache. Page coloring, thus, alters an application's data positioning within the cache. Fig. 9.1 demonstrates how page coloring maps the pages of three applications (B, G, R) to different positions in the cache memory.

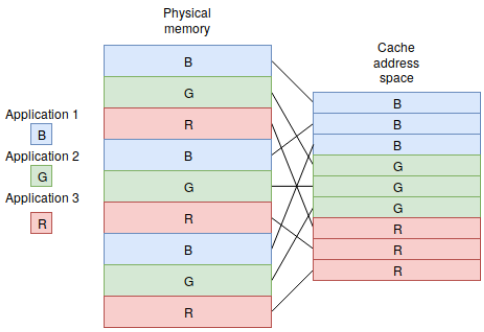

Figure 9.1: Cache coloring

Applications executing within a page-colored environment will not suffer from shared cache contention since page coloring provides a clear border in the cache between where applications are allowed to position data. Page-coloring utilizes a cache's set-associative and presents an additional overhead to memory allocations due to algorithm complexity. The trade-off with cache partitioning, therefore, stands between performance and isolation.

9.2.4 Analyzing resource-boundness

We discuss here a statistical approach to quantify the resource-boundness using the Pearson correlation coefficient.

The Pearson-correlation coefficient has three types of outcomes, 1 - which means a complete positive correlation between two datasets; 0, which means no correlation between the datasets; and -1, which means complete negative correlation. We exemplify the three types of correlation in Figure 9.2.

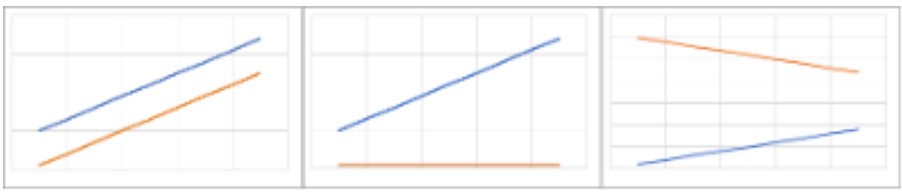

Figure 9.2: Example of positive- (left) zero- (middle) and negative (right) correlation

We analyze resource-boundness by investigating negative relationships between the number of instructions completed and a resource event. Negative relationships mean either the number of instructions completed increase while the number of resource events decrease, or vice-versa. We utilize the PMU - that consists of several performance monitor counters (PMCs) - to monitor an applications' performance. We monitor the *Instructions retired* event, which counts the number of instructions that went through all processor pipeline stages, as a quantifiable performance metric. The second metric defines the kind of resource-boundness the user is interested in (e.g. if interested in cache related boundness we count the cache misses - via counters such as L_1D-cache-misses, etc).

In this paper, we utilize the analysis part of LLM-shark to determine the necessity of placing an application into a Last-Level Cache partition container [22]. Here, we mainly focus on the L_2-cache misses as performance counter-event.

The resource-boundness analysis further covers three actions:

- *Determine the event set.* To determine the resource-boundness, we first need to determine which counter events to sample. The complete counter set (perf+PAPI) lists a total of 116 events. For this paper, we chose to use only the 11 PAPI preset events.

- *Sample performance counters.* We sample selected performance counters during application execution at a fixed rate and save the data for resource-boundness analysis.

- *Assessment of the resource-boundness.* We utilize the performance counter samples from the previous step to calculate the Pearson-correlation coefficient. We quantify the magnitude of the resource-boundness according to the approach proposed by Mindrila and Balentyne [13]: the coefficients are compared using positive and negative values where 0-0.3 is considered none or weak, 0.3-0.5 weak, 0.5-0.7 moderate and greater than 0.7 is considered strong. Since the L_2-cache-misses provide a negative impact on the application, we are only interested in negative correlations. We therefore only consider applications for cache partitions if they present a correlation less than -0.25

9.3 Methodology

In previous work, we discussed the definition of resource-boundness [6] and also the consequences of running resource-bound loads simultaneously on different cores [4] [5]. The main take-away point is that resource-boundness is an important factor to consider when partitioning a system since the resource-boundness is an indicator of what resource an applications' performance depends on.

This section investigates the resource-boundness of six different applications, four of them implementing feature detection algorithms (SUSAN, Harris, SIFT, and FAST), and two presenting synthetic workloads (Matrix multiplication and Bubblesort). We illustrate the effects of resource contention on the execution of the six applications and discuss the relation to the respective correlation coefficient values.

9.3.1 System model

The relevant characteristics of the six mentioned applications are presented in Table 9.1.

Table 9.1: LLC-PC specifics

Application	Data input	type
Harris	2 MB bmp	Corner detection
SUSAN	2 MB bmp	Corner detection
FAST	12 MB bmp	Corner detection
SIFT	256 KB pgm	Object detection
Matmult	200x200 array	Synthetic
Bubblesort	20000 elements array	Synthetic

We use a variety of data input to showcase the usability of LLM-shark. The purpose here is not to create a comparison study on which application executes best in certain circumstances. Instead, our aim is to show that LLM-shark works for a variety of applications, independently on the applications memory footprint. Each application runs within the execution context of the tool, which starts an application, samples the desired performance events during the application execution, calculates a resource-boundness estimation and finally positions the application within a cache partition container. Figure 9.3 depicts

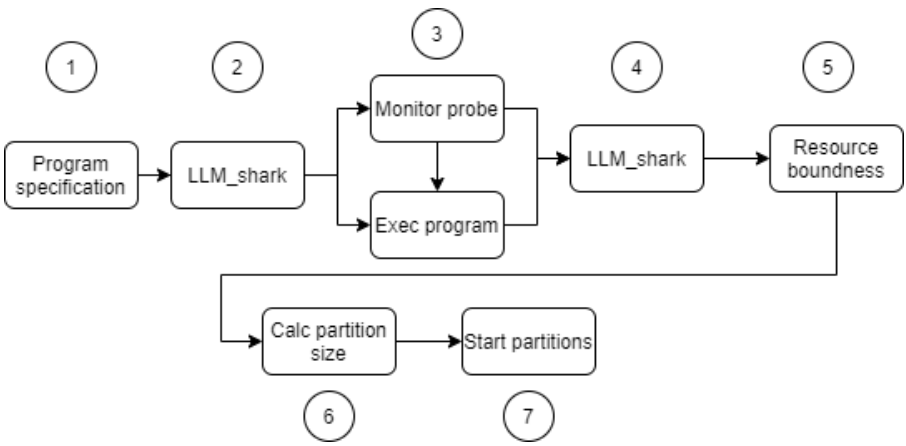

Figure 9.3: LLM shark execution flow

the respective core functionality and execution flow. It shows the seven major execution steps of LLM-shark, which we describe in detail as follows:

1. *Application identification phase* - LLM-shark uses the filepath to an application executable to run the application within an LLM-shark context.

2. *Initialization phase* - initializes the instructions retired performance counter together with the desired counters.

3. *Fork phase* - LLM-shark executes a fork operation and runs the application within the context of child process. During the child process' execution, we monitor the performance counters of the child process continuously at a sampling frequency.

4. *Synchronization phase* - when the forked application has stopped, we store all the performance counter data from the child process and compute the correlation.

5. *Data store phase* - store the resource-boundness of the application so that appropriate actions such as cache partitions can be made.

6. *L_2-cache partition calculation phase* - calculate the L_2-cache partition size for each application based on the application correlation.

7. *L_2-cache partition actuation phase* - execute the applications within their respective L_2-cache partition containers.

LLM shark utilizes performance counters in a context of PAPI, a framework that includes preset performance counter events and the native counters that are specified by the Perf API. In this paper, we use a Xilinx zynq zcu102 evaluation kit which supports 13 preset PAPI counters and an additional 103 native performance counters. For the sake of readability, we only include results containing the preset PAPI counters, as data from 116 events are too much to present in one paper. We list the PAPI preset counter events with a short description in Table 9.2.

We limit this paper to only focus on non I/O-bound applications where the perceived performance of an application is equal to the number of instructions retired per time interval. As such, we omit the PAPI_HW_INT counter since that counter-event defines an application type that we are not interested in. PAPI_TOT_INS defines our performance metric and will be used in all correlation calculations versus another hardware resource. We, therefore, omit this counter from correlation calculation since correlating a value against itself always is one and will not be meaningful. We also omit PAPI_TOT_CYC since it presents the number of active clock cycles for an application and does not hold relevance to the internal memory hierarchy.

Table 9.2: PAPI preset counter events

Event name	Brief explanation
PAPI_L1_DCM	L_1D-cache misses
PAPI_L1_ICM	L_1I-cache misses
PAPI_L2_DCM	L_2-cache misses
PAPI_TLB_DM	DTLB misses
PAPI_TLB_IM	ITLB misses
PAPI_TOT_INS	Instructions retired
PAPI_HW_INT	Hardware interrupts
PAPI_LD_INS	Memory load instructions
PAPI_SR_INS	Memory store instructions
PAPI_BR_INS	Branch instructions
PAPI_TOT_CYC	Processor cycles completed
PAPI_L1_DCA	L_1D-cache accesses
PAPI_L2_DCA	L_2-cache accesses

9.4 Application experiments

We have executed our experiment on two scenarios - baseline and contended. The baseline scenario is defined by the target application running without any deliberately disturbing load. The contended scenario presents the application running simultaneously with a leech application causing artificial L_2-cache contention. We list our test platform in Table 9.3.

Table 9.3: Hardware specifications Xilinx Zynq UltraScale+ MPSoC

Feature	Hardware Component
Core	4xArm Cortex A-53 @ 1.2GHz
L_1I-cache	32 KB 2-way set assoc cache/core
L_1D-cache	32 KB 4-way set assoc cache/core
L_2-cache	1 MB 16-way set assoc. shared Last-level Cache
MMU	L_1ITLB: 10 entries L_1DTLB: 10 entries L_2TLB 512 entries, 4-way set assoc.

9.4.1 Baseline scenario

Table 9.4 shows the median execution-time for each investigated application, taken over 100 measurements. We illustrate the instructions retired and the L_2-cache misses of the Harris algorithm in Figure 9.4 and the FAST algorithms in Figure 9.5 for demonstrative purposes.

Table 9.4: Application baseline execution-time

Application	Median execution-time (ms)
Harris	306
SIFT	750
SUSAN	189
Matmult	224
FAST	133.8
Sort	797

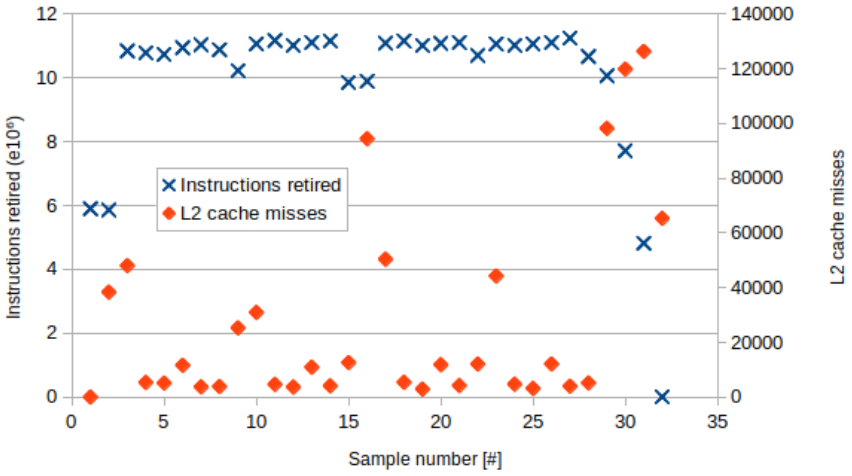

Figure 9.4: Harris execution characteristics

Fig. 9.4 and Fig. 9.5 illustrates the Harris and FAST applications' execution profiles running within the context of LLM-shark at a 200Hz sampling frequency, denoted by the x-axis. The left y-axis denotes the number of instructions retired with blue crosses, and the right y-axis denotes the number of L_2-cache misses with red diamonds. The execution profiles of these two applications are visibly different.The trend for the Harris application is that the number of instructions retired decreases while at the same time the L_2-cache

Figure 9.5: FAST execution characteristics

misses increases. This trend is most notable at the last four measurement values; a similar, weaker trend can also be detected in the rest of the measurements. The FAST application do not have the same trend since the instructions retired continuously increases while the L_2-cache misses remaining the same at a count of 5000 for most of the application. There are small trends between sampling points 4 and 6. The majority of the application is, however, unaffected by L_2-cache misses. We list the correlation coefficients for all different counters for our six applications in Table 9.5.

Table 9.5: Correlation between instructions retired and PAPI preset counters

Counter	Harr	SUS	FAST	Matm	SIFT	Sort
BR_INS	0.69	0.48	0.82	0.19	0.63	0.98
BR_MSP	0.77	0.36	0.71	0.28	0.63	0.04
L1_DCA	0.76	0.65	0.83	-0.89	0.85	0.86
L1_DCM	-0.03	0.26	0.26	**-0.80**	0.75	0.23
L1_ICM	**-0.39**	**-0.29**	0.68	0.49	**-0.3**	-0.06
L2_DCA	**-0.25**	0.28	0.1	-0.83	0.49	0.15
L2_DCM	**-0.49**	0.2	0.12	**-0.84**	**-0.26**	-0.08
LD_INS	0.71	0.61	0.83	**-0.99**	0.87	0.91
SR_INS	0.79	0.43	0.16	-0.06	-0.36	0.79
TLB_DM	**-0.56**	-0.13	-0.05*	0.27	-0.01	-0.02
TLB_IM	**-0.53**	-0.13	0.23	0.31	-0.06	0.02

The table shows the correlation between the preset PAPI counters and the num-

ber of instructions retired for each application. Our theory is that applications that negatively correlate to a performance counter that implies pipeline stalls, such as the L_2-cache misses or DTLB misses will be sensitive in terms of execution-time when other applications utilize the same resources. The sensitivity depends on the correlation value magnitude; a higher correlation means the application will be more prone to performance implications if other applications are using that same resource. We summarize the negative correlation coefficients for each application in Table 9.6 since these counters show an indication for resource-boundness.

Table 9.6: Correlation summary

Application	**Resource-boundness**
Harris	L_1I-cache misses (Weak)
	L_2I-cache accesses (Weak)
	L_2-cache misses (Moderate)
	DTLB misses (Moderate)
	ITLB misses (Moderate)
SUSAN	L_1I-cache misses (Weak)
FAST	No resource-boundness
Matmult	L_1D-cache misses (Strong)
	L_2-cache accesses (Strong)
	L_2-cache misses (Strong)
	L_2-cache misses (Strong)
Bubblesort	No resource-boundness
SIFT	L_1I-cache (Weak)
	L_2-cache misses (Weak)

Out of the six applications, three - SIFT, Harris and the Matrix multiplication - display weak, moderate and strong negative boundness to the L_2-cache misses counter, respectively. Harris furthermore displays a moderate relationship versus both TLBs.

9.4.2 Resource contention

In this section, we empirically show the relationship between the correlation coefficient and how our test applications' execution-time is affected by simultaneously executing artificial loads that utilize the shared cache. We define our hypothesis as follows.

Hypothesis The magnitude of the correlation coefficient indicates how closely tied an applications' performance is with a resource. Reducing this resource's size or capacity will affect the performance of applications with a high correlation towards this resource to a greater extent than applications with a low correlation towards this resource.

We chose to reduce the capacity of the L_2-cache through the execution of a memory-intensive program called leech [10]. The leech executes simultaneously as our our benchmark applications and is positioned on other non-occupied cores to enforce shared cache contention. The leech is built as a memory specific load and executes a read-then-write access pattern on an integer array of variable size. We run the leech using a specific stride pattern in the array to force as many cache line evictions from our benchmark applications as possible. To generate maximum L_2-cache contention, we run three separate instances of the leech on different cores (2,3,4) while the application runs on core 1. Table 9.7 summarizes the leech specifics.

Table 9.7: Leech specifics

Property	**Value**
Iteration sleep	0
Array size	2 MB
Core affinity	Core 2,3,4
Stride Length	64 Byte
Access method	read-then-write

Fig. 9.6 depicts the execution characteristics of the Harris application running on core 1 in a leech contented environment. The red diamonds plot the L_2-cache misses on the right-hand side y-axis, and the blue crosses plot the instructions retired on the right-hand side y-axis.

There are two significant differences between the baseline Harris of Fig. 9.4 and the leech loaded Harris of Fig. 9.6. Firstly, the number of instructions retired in the leech loaded version is significantly less per 50 μs than in the baseline case. Since the number of instructions retired is considerably less, on average 23.7% less per measurement point, the performance becomes significantly worse. The graph also shows an apparent increase in the number of caches misses per time interval compared to the baseline Matrix multiplication with an average of 28% increased cache misses per measurement point.

For comparative purposes, we depict the execution characteristics for the non-L_2-cache-bound application FAST running in a leech setting in Fig 9.7.

The baseline FAST version executes an average of 4.4% more instructions on

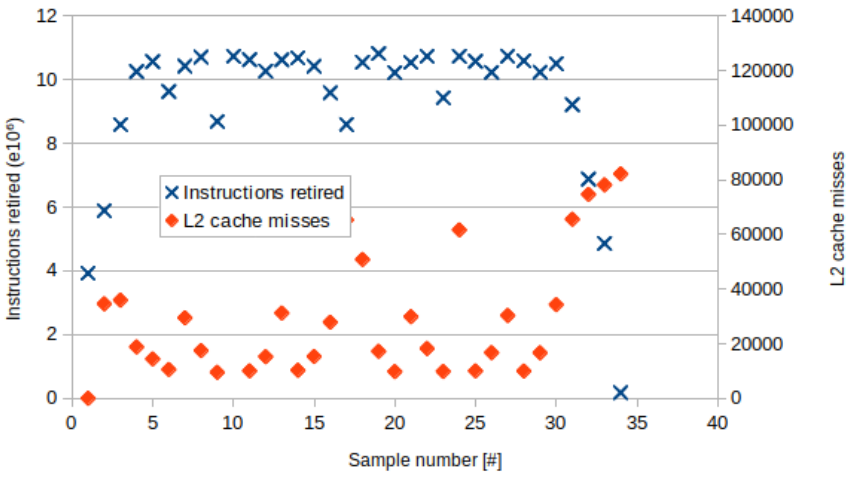

Figure 9.6: The Harris application running together with leeches

average per 50 μs than the leech-loaded version. The difference in L_2-cache is also not overwhelming, (on average 1.1% more) in our leech version versus the baseline version. The small increase in execution-time, small decrease in instructions retired and small decrease in L_2-cache misses means FAST did not suffer notably from heavy L_2-cache contention. This goes in line with our assertion of the previous section that FAST, in fact, is not L_2-cache-bound. Harris, on the other hand, displays notably different behavior; the instructions retired are now jittery, especially in the middle sections of the execution. The L_2-cache misses are also more jittery and counts significantly more than the baseline version. Table 9.8 summarizes the results from our leech tests and shows the application execution-time from our leech loaded version (column 2) and the percentage difference in execution-time compared to the baseline execution (column 3). The table also shows the percentage difference in instructions retired per 50 μs (column 4) and L_2-cache misses (column 5) between the leech loaded version and the baseline execution of the applications.

The matrix multiplication suffered the worst execution-time losses due to resource contention, with a 42.1% increase in total execution-time, which is a drastic performance decrease. The FAST application suffered a 2.96% increase in execution-time, which is also in line with the low L_2-cache correlation listed in the previous subsection. Another stand-out measurement is the number of increased cache misses for the sort application of 7600%. This measurement is not an error but rather a natural consequence of the few misses in the baseline

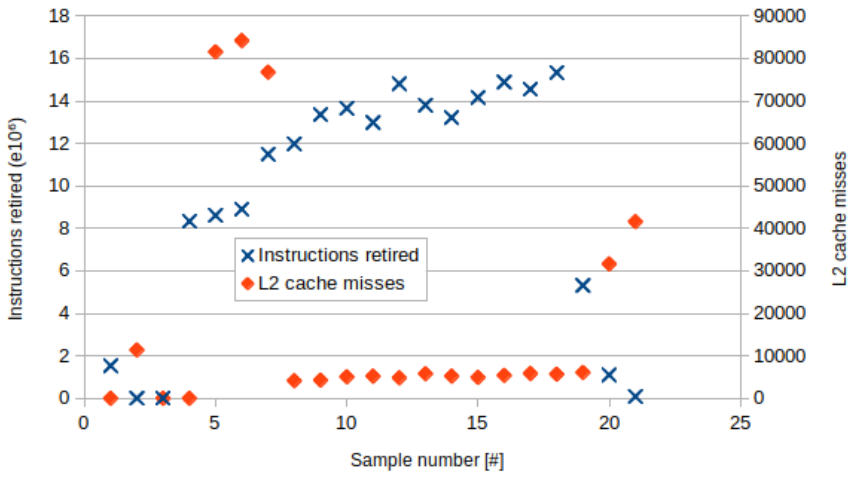

Figure 9.7: FAST execution characteristics with leech

Table 9.8: Summary of performance loss due to L_2-cache contention

Application	Ex. [ms]	Diff.[%]	Instr.[%]	L2.[%]
Harris	328	-7.41%	-5.76%	+64.14%
SUSAN	209	-7.3%	-0.75%	-16.62%
FAST	137.8	-3.02%	-4.4%	+1.1%
Matmult	381	-41.2%	-31.7%	+36.5%
SIFT	799.5	-6.6%	-10.6%	+6.3%
Sort	800.1	-0.38%	-0.93%	+7600%

case (1 L_2-cache-miss on average) compared to the leech case (7600L_2-cache-misses on average). The increase is drastic, but the count is not enough to significantly reduce the performance.

The second worst is the Harris application (moderate correlation), which presents a 7.41% execution-time decrease. The third worst application is SUSAN (no correlation), which also presents an interesting case - the application displays no L_2-cache-boundness according to the correlation calculation. However, it still shows a notable performance decrease, while displaying a less L_2-cache misses average per 50 μs than the baseline case. Since SUSAN shows a performance degradation while simultaneously showing a decrease in L_2-cache misses, it cannot be L_2-cache-bound.

9.5 Partitioning experiments

In previous subsections, we show how we use the Pearson-correlation coefficient to determine the resource-boundness of an application and how L_2-cache-cache contention affects the application's performance. Here, we apply the knowledge of an applications' resource-boundness for assigning L_2-cache-partition sizes.

9.5.1 Cache partitioning performance impacts

In this section, we present experiments on the effects of executing our different applications within a cache partitioned environment. LLM-shark relies on the Palloc framework to implement page-coloring, which replaces the default *buddy* allocator algorithm in the Linux kernel. The page coloring algorithm utilizes the cache set-associative addressing bit for determining the memory location of new data. We list the L_2-cache specifications in detail in Table 9.9.

Table 9.9: L_2-cache specification of ARM cortex-a53

Property	**Size**
Cache size	1 MB
Line length	64 Byte
Set-associativity	16
Set size	1024 Byte
Number of sets	1024
Replacement policy	Pseduo-random

The number of available cache partitions (colors) on a platform depends on the cache size, number of sets and the page size (4 KB), see Eq 9.2.

$$Nr.\ of\ Colors = \frac{Cache\_size}{Cache\_ways * page\_size} \tag{9.2}$$

Our platform provides 16 colors according to the formula and each color provides a 64 KB memory area. We showcase the effects of cache partitioning through a set of experiments where we measure our applications' execution-times under different color assignments, using 2-, 4-, 7-, 10-, 13-, and 16-assigned slices for the application. We assign one color for LLM-shark so that it can operate. We list the median execution-times of 100 measurements for each application using different cache partition sizes in Table 9.10 with the slices translated into the actual L_2-cache partition size.

Table 9.10: Application execution-times in milliseconds of applications using different L_2-cache partition sizes

	Partition size (KB)					
Application	128	256	448	640	832	1024
Harris	320	312	312	309	307	306
SIFT	780	778	777	775	774	771
Matmult	1214	1162	979	876	451	288
FAST	138	138	138	138	138	138
Sort	798	797	797	797	797	797
SUSAN	202	199	198	196	196	196

The table shows the difference in execution-time for each respective application, using different cache partitions where Matmult displays the most execution-time difference due to change in L_2-cache partition size. We further plot the number of instructions retired and the number of L_2-cache misses using different L_2-cache partition sizes in Fig. 9.8 and Fig. 9.9 respectively.

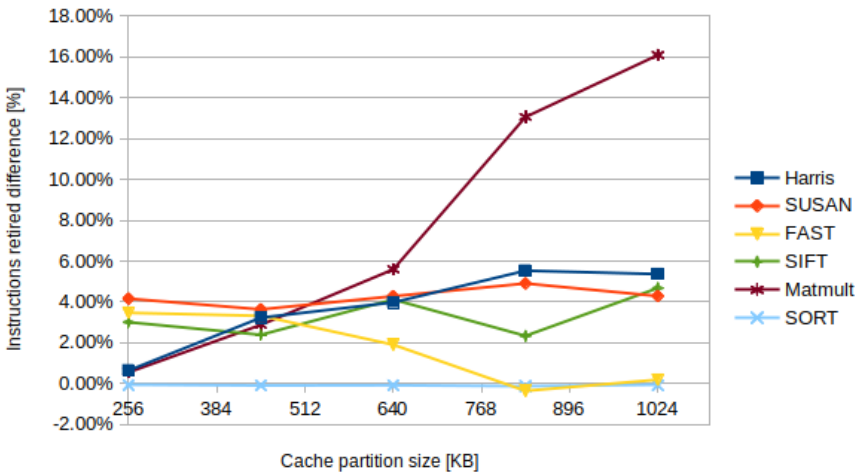

Figure 9.8: Difference in L_2-cache misses using different L_2-cache partition sizes.

Fig. 9.8 plots the percentage difference in the number of instructions retired on the y-axis when scaling up the cache partition size. A high percentage means more instructions retired per 50 milliseconds and is preferable to a low percentage difference. Fig. 9.9 plots the percentage change in the number of L_2-cache misses on the y-axis when increasing the cache partition size. The plots show an inverted scale, which means a positive percentage difference points to a decrease in L_2-cache misses compared to our reference measure-

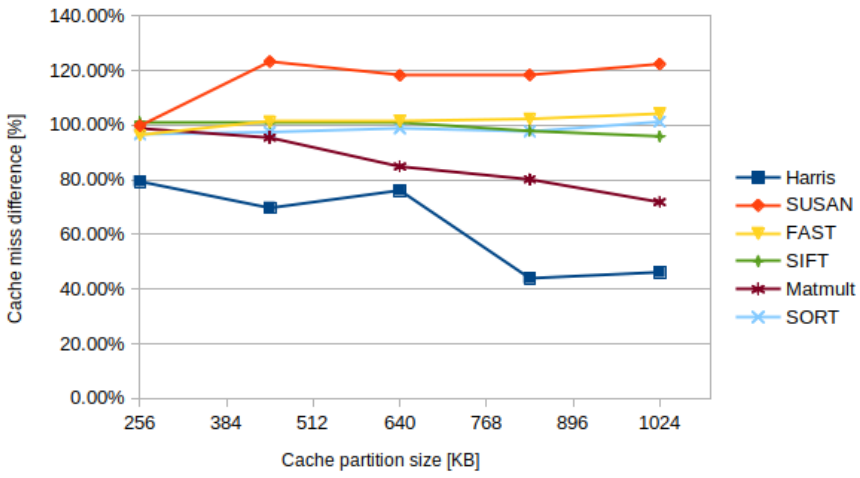

Figure 9.9: Difference in instructions retired misses using different L_2-cache partition sizes.

ment. Since Fig. 9.9 only plots the difference in L_2-cache misses, it is not possible to conclude that higher percentages are preferable to low percentages. Instead, the cache misses must be interpreted in an instructions retired context, where a decrease in the number of cache misses leads to an increased amount of instructions retired.

9.5.2 Initial cache partitions

Previously [3], we used a methodology called LLC-PC which tries to find the best Last-level cache partition size for an application. The methodology is an iterative process that continuously increases the Last-level cache partition for the application until a desired performance has been met. The method utilizes a run-time comparison scheme and measures an application's performance while increasing the cache partition size – if an increase in cache partition size positively affects the application's performance, LLC-PC continues increasing the partition size; if not, then then stop increasing the cache partition size. LLC-PC uses the smallest possible cache partition size (which in our case is 64 KB) for starting point to avoid over-saturation. LLC-PC then increases the cache partition size by one (64 KB) for every iteration. Thus, it may take several iterations before reaching the desired performance, since the starting point is set at the smallest cache partition size possible.

Table 9.11: L_2-cache Initial cache partition suggestions comparison

	LLM-shark			$C_{50\mu s}$			C_{tot}			Execution-times		
Application	**Corr.**	**Norm.%**	**Size**	**Median**	**%**	**Size**	**Number**	**%**	**Size**	**LLM**	**$C_{50\mu s}$**	**C_{tot}**
Harris	-0.49	30.97%	5	11274	15.36%	2	$8.7 * 10^5$	15.36%	2	309.2	313.9	313.9
Matmult	-0.84	52.69%	8	23871	33.95%	5	$2.2 * 10^6$	40.37%	6	897.5	1137.4	1075.3
SIFT	-0.26	16.34%	2	15401	21.9%	3	$1.3 * 10^6$	23.86%	3	780	779	779
SUSAN	**—Omitted—**		1*	2786	3.96%	1**	$4.5 * 10^5$	7.95%	1	212	210	205
FAST	**—Omitted—**		1*	16972	24.14%	4	$7.0 * 10^5$	12.37%	2	145.1	138.9	138.8
SORT	**—Omitted—**		1*	6	0.01%	1**	$4 * 10^3$ %	0.08%	1**	813.7	807.4	797.9

**Executing simulatenously on different cores using the same cache partition*

***Percentage not sufficient to justify a standalone partition, instead use a shared container*

Our optimization proposal is to have a “reasonable" starting point, i.e., an initial L_2-cache partition size from where to start scaling the partition sizes. Here, we illustrate the usage of the correlation coefficient to determine the size of an initial cache partition: applications with high correlations should receive more spacious partitions while low correlation applications should receive less space. In our methodology, we normalize all the correlation coefficient values with a magnitude $\geq$ to weak and partition assign partitions according to the percentage value of our 15 available colors. The initial correlation approach for determining resource-boundness was presented in [6], in here we apply that methodology to assign cache partitions. The normalized values gives a percentage and is used to calculate the cache partition sizes. Since the cache partition space only provides 15 colors, we also need to round-off decimal values to the closest integer values find an appropriate cache partition. E.g., an application that displays a normalization percentage of 30% ($15 * 0.3 = 4.5$) will result in a cache partition size of 4.5 – we round-off 4.5 to the closest integer value, 5, since it is not possible to use fraction numbers as a cache partition.

We discard applications with a lower than weak correlation since additional cache partitions since our assessment is that these application will receive little to no performance benefits from increased cache partition size. We instead execute these applications within a "Junk" container - a partition with space of (64 KB) that holds all low correlation applications, and as such, we do not waste L_2-cache space on these applications. We argue our L_2-cache partition distribution methodology is preferable to other methodologies, such as assigning L_2-cache partitions based on L_2-cache misses or L_2-cache accesses [22] since our methodology includes the resource-boundness factor. We compare our correlation methodology versus two L_2-cache misses distribution policies ($C_{50\mu s}$ and C_{tot}). The table contains results using three methodologies, specified as follows:

1. LLM-shark (column 2-4) – distributes L_2-cache partitions based on correlation.

2. $C_{50\mu s}$ (column 5-7) – distributes L_2-cache partition size based on the median number of L_2-cache misses per $50\mu s$.

3. C_{tot} (column 8-10) – distributes L_2-cache partitions based on the total number of L_2-cache misses per given application.

The table shows the execution-time results and cache partition sizes of the

three different methodologies, LLM-shark, $C_{50\mu s}$ and C_{tot}. The most significant difference is seen from the Matrix multiplication perspective, which receives the most spacious L_2-cache partition size, followed by Harris. SUSAN, SORT and FAST are all assigned to share "Junk" L_2-cache partition container. An application must utilize the locality of reference [20] principle to benefit from the L_2-cache and as such be L_2-cache-bound. Through code inspection of FAST [17], we conclude that FAST cannot be L_2-cache-bound due to a lack of locality utilization – even though the performance counters show a high count (relative to the other applications) of L_2-cache misses. Due to FAST's high L_2-cache count, it receives L_2-cache using the cache miss-based policies, and can be seen as a waste of L_2-cache partition space since it's performance is not affected notably. Due to FAST's low correlation, it does not receive any individual L_2-cache partition space but is instead assigned to the Junk container. We summarize the comparison between the different methodologies in Table 9.12. The columns marks the best solution and specifies to performance degradation for each application compared to the best results. A higher value means an increase in execution time and is therefore a more significant performance degradation than a low value.

Table 9.12: Comparison summary

	Execution-time comparison (ms)		
Application	LLM-shark	$C_{50\mu s}$	C_{tot}
Harris	Best	+4.7	+4.7
Matmult	Best	+239.9	+177.8
SIFT	+1	Best	Best
SUSAN	+7	+5	Best
FAST	+6.2	Best	+0.1
SORT	+15.8	+9.8	Best

Our correlation-based methodology achieved the best execution-times for the matrix multiplication and the Harris applications. SORT displays the most significant downgrade using our approach, which is a fifteen milliseconds performance degradation, comparatively the cache-based distribution policies that display a 239.9ms ($C_{50\mu s}$) and (+177.8 C_{tot}) for our most cache heavy load, the matrix multiplication. The cache misses distribution policies instead focuses on

9.5.3 Discussion

The comparison shows that our correlation-based methodology assigns most cache partitions to the matrix multiplication than the cache misses distribution policies due to its high resource-boundness. The Harris application also receives most cache partitions using our correlation-based approach due to its resource-boundness, resulting in the best performance. SIFT suffers a 1ms performance degradation using LLM-shark. The junk cache partition displays a slight performance degradation for FAST, SUSAN, and SORT, indicating that cache contention occurs within this specific cache partition. However, the performance degradation of these three applications is slight compared to the performance gains of the matrix multiplication in LLM-shark compared to the cache-misses-based distribution policies. The matrix multiplication performance results display the main take-away point from this paper; it is not how frequently an application utilizes the cache that determines how the application responds to a change in cache partition size, but rather *how* an application use the cache. The matrix multiplication has a high data-reusage and tries to access the same cache memory several times during execution. If the cache space is reduced, the matrix multiplication cannot re-use data to the same extent since the cache is smaller and will suffer a significant performance penalty. Comparatively, FAST does not show cache-boundness due to low cache re-usage but still maintains a high cache-miss count. FAST fetches data from the main memory, leading to cache misses, but does not re-access the same data again; therefore, FAST is not cache-bound and does not benefit from increased cache partition space.

We argue that it is more beneficial to assign cache partitions based on their resource boundness rather than the cache-miss count since an increase in cache partition space provides more significant performance benefits to the highly cache-bound applications than non-cache bound applications.

9.6 Related Work

Related work includes papers directed towards investigating the resource-boundness. Work such as Cache Pirating [7] and Bandwidth Bandit [8] are tools for generating shared resource contention and and can empirically pinpoint how much an application suffers from shared resource contention, which is similar to our leech methodology. Even though these works provide a structured methodology for pinpointing resource contention in one particular

resource, the process can become time consuming since a bandit, a pirate or even a leech has to be designed specifically for each individual resource in order to generate and measure contention. Our opinion is that our correlation methodology can significantly decrease the complexity of such tests.

Other works such as Scarphase [19] divides program execution into phases and proposes a method for identifying how the resource usage changes over time in applications using the perf interface for measuring the performance counters. Sembrandt et al. [18] expands on the same direction topic and explains the differences in phase behavior between serial and parallel applications.

The body of cache partitioning papers is relatively large [12]. Coloris [22] is an excellent example, with an approach that splits the cache partitions according to how many cache misses one process is responsible for. As we show in our paper, it is not necessary that the application with the most cache misses benefits the most from cache partitioning, instead we have to look at the significance of the cache misses and take it into relationship on how it affects the performance. Brock et al. [2] discusses how to optimally allocate cache partitions to different processes through exhaustive searches.

Perarnau et al. [16] argue that the cache partition size is best left to the user since it is the user that in the end knows what performance the application requires. The last level cache is, however, a very complex hardware resource due to the contention factor and also due to the limited size. This means the user needs expert knowledge on how the system applications utilizes the cache. Our solution eliminates the need for expert knowledge, through our partitioning scheme.

With LLM-shark, we cover all the aspects of performance improvements based on cache usage: i) we analyze the behavior of an application; ii) we assert potential problems such as resource contention; iii) we actuate mitigation strategies to avoid resource contention before it happens. While all the other similar approaches perform only one or two of these actions, we even provide an automated solution.

9.7 Summary

We show that the LLM-shark tool can assign cache partitions automatically to applications based on their hardware resource-boundness. We have conducted a series of tests to assess the usability of the Pearson correlation coefficient as a tool for determining resource-boundness. We then verify that resource bound

processes benefits to various extent depending on cache partitioning sizes and the level of hardware resource-boundness. Finally, we propose a normalization scheme to assign initial cache partitions to a system, based on the magnitude of the Pearson correlation coefficient. Our tool thus answers the following questions:

1. Which hardware components are limiting factors to application performance?
2. Which hardware resources are potential contention bottlenecks to the application?
3. How spacious should initial cache partitions be?

We focus only on the L_2-cache as a test subject. However, we argue that the correlation methodology is generalizable to all hardware events which imply a negative performance impact, such as TLB-, L_1D-cache-, L_1I-cache-misses and page-misses etc.

9.7.1 Future work

For future work, we plan to integrate our tool to different architectures that provide a more sizeable cache memory. Since our Last-Level Cache (the L_2-cache) is relatively small, the number of the available cache partitions is also small. More spacious cache memory will leave more room for flexibility in the cache partition assignments and, therefore, the execution-times of our L_2-cache-bound applications.

In the paper, we mention several mitigation techniques such as bandwidth reservation and TLB coloring which also can benefit from correlation based resource-boundness estimation. For future work, we intend to implement more mitigation techniques for LLM-shark, utilizing the same correlation-based scheme for determining parititon sizes. We furthermore plan to integrate LLM-shark with LLC-PC [3] to enable the dynamic adaptation of partition sizes once we have them assigned. Other future work includes investigating other metrics for formulating initial partition sizes. Our current strategy only looks at the magnitude of the correlation, which serves as a reasonable starting point. However, to provide an even more accurate partition estimation, it could be possible to model how much instructions retired can be gained from one single cache partition.

Bibliography

[1] ARM. Cortex-a53. `https://developer.arm.com/ip-products/processors/cortex-a/cortex-a53`. 2021-01-06.

[2] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang, and Y. Luo. Optimal cache partition-sharing. In *2015 44^{th} International Conference on Parallel Processing*, pages 749–758. IEEE, 2015.

[3] J. Danielsson, M. Jägemar, M. Behnam, T. Seceleanu, and M. Sjödin. Run-time cache-partition controller for multi-core systems. In *IECON 2019-45th Annual Conference of the IEEE Industrial Electronics Society*, volume 1, pages 4509–4515. IEEE, 2019.

[4] J. Danielsson, M. Jägemar, M. Behnam, M. Sjödin, and T. Seceleanu. Measurement-based evaluation of data-parallelism for opencv feature-detection algorithms. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 701–710. IEEE, 2018.

[5] J. Danielsson, T. Seceleanu, M. Jägemar, M. Behnam, and M. Sjödin. Testing performance-isolation in multi-core systems. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 604–609. IEEE, 2019.

[6] J. Danielsson, T. Seceleanu, M. Jägemar, M. Behnam, and M. Sjödin. Resource depedency analysis in multi-core systems. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 87–94. IEEE, 2020.

[7] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache pirating: Measuring the curse of the shared cache. In *2011 International Conference on Parallel Processing*, pages 165–175. IEEE, 2011.

[8] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Bandwidth bandit: Quantitative characterization of memory contention. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE, 2013.

[9] T. Gleixner. Linux Performance Counter announcement, 2008.

[10] M. Jagemar, A. Ermedahl, S. Eldh, M. Behnam, and B. Lisper. Enforcing quality of service through hardware resource aware process scheduling. In *23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 329–336. IEEE, 2018.

[11] R. Jain. *The Art Of Computer Systems Performance Analysis: Techniques For Experimental Measurement, Simulation, And Modeling*. john wiley & sons, 2008.

[12] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14. IEEE, 2019.

[13] D. Mindrila and P. Balentyne. Scatterplots and Correlation.

[14] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[15] S. A. Panchamukhi and F. Mueller. Providing task isolation via tlb coloring. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 3–13. IEEE, 2015.

[16] S. Perarnau, M. Tchiboukdjian, and G. Huard. Controlling cache utilization of hpc applications. In *Proceedings of the international conference on Supercomputing*, pages 295–304. ACM, 2011.

[17] E. Rosten and T. Drummond. Fusing points and lines for high performance tracking. In *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, volume 2, pages 1508–1515. Ieee, 2005.

[18] A. Sembrant, D. Black-Schaffer, and E. Hagersten. Phase behavior in serial and parallel applications. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 47–58. IEEE, 2012.

[19] A. Sembrant, D. Eklov, and E. Hagersten. Efficient software-based online phase classification. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 104–115. IEEE, 2011.

[20] W. Stallings. *Computer organization and architecture: designing for performance*. Pearson Education India, 2003.

[21] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 272–282. ACM, 2003.

[22] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: a dynamic cache partitioning system using page coloring. In *Parallel Architecture and Compilation Techniques (PACT), 2014* 23^{rd} *International Conference on*, pages 381–392. IEEE, 2014.

[23] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 155–166. IEEE, 2014.

[24] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE* 19^{th}, pages 55–64. IEEE, 2013.

[25] G. Zellweger, D. Lin, and T. Roscoe. So many performance events , so little time. *APSys '16*, 2016.

Chapter 10

Paper D: Testing Performance-Isolation in Multi-Core Systems

Jakob Danielsson, Tiberiu Seceleanu, Marcus Jägemar, Moris Behnam, and Mikael Sjödin. Testing Performance-Isolation in Multi-Core Systems In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC). Vol. 1. IEEE, 2019*

Abstract

In this paper we present a methodology to be used for quantifying the level of performance isolation for a multi-core system. We have devised a test that can be applied to breaches of isolation in different computing resources that may be shared between different cores. We use this test to determine the level of isolation gained by using the Jailhouse hypervisor compared to a regular Linux system in terms of CPU isolation, cache isolation and memory bus isolation. Our measurements show that the Jailhouse hypervisor provides performance isolation of local computing resources such as CPU. We have also evaluated if any isolation could be gained for shared computing resources such as the system wide cache and the memory bus controller. Our tests show no measurable difference in partitioning between a regular Linux system and a Jailhouse partitioned system for shared resources. Using the Jailhouse hypervisor provides only a small noticeable overhead when executing multiple shared-resource intensive tasks on multiple cores, which implies that running Jailhouse in a memory saturated system will not be harmful. However, contention still exist in the memory bus and in the system-wide cache.

10.1 Introduction

While great advancements in virtualization and partitioning techniques nowadays allow logical and functional partitioning of a system into a set of independently executing subsystems (referred to as partitions) [2], there exists no practical and efficient methods to guarantee that different partitions have no negative impact on each other's performance. That is, contemporary techniques give logical isolation but not performance isolation. In this paper we propose a method for testing the performance isolation between different subsystems running on different cores in a multi-core architecture. Furthermore, the method tests isolation of different computing resources such as CPUs, caches and memory-system. Thus, it allows to pinpoint any sources of breached isolation and it enables mitigation of such breaches by introduction of specific isolation techniques for specific resources. With the introduction of multi-core architectures as the standard platforms for performance-critical application-domains like embedded systems and real-time systems, the issues of performance guarantees on these architectures becomes paramount. In multi-cores, isolation is hampered since a wealth of computing resources are shared between cores, such as caches, TLBs (Translation Lookaside Buffers), memory controllers and memory banks.

Our work is a step towards allowing empirical evaluation of performance isolation in complex multi-core architectures. We demonstrate the use of our model by evaluating performance isolation obtained by the Jailhouse hypervisor [15] and comparing it with running a non-partitioned Linux system.

Isolation is a complex topic and a clear terminology needs to be defined, for example: what is shared resource isolation?

The *performance isolation* is defined here by the slowdown in execution of an application while running in a context where access to resources is contended by other applications, too. An application that runs with a specific performance without any disturbing processes (in *isolation*) runs at a *baseline performance*. An application running with deliberately disturbing processes is running at a *loaded performance*. If the loaded version runs with the same performance as the baseline version, the application is performance isolated. Performance isolation of applications targeting specific hardware can be accomplished by using methods such as page coloring [10], hypervisors [6], bus-scheduling [19]. Many different techniques are available for isolating hardware from disturbances generated by other processes, but most techniques cover only one or two parts of the hardware resources. The resource partitioning hypervisor Jail-

house developed by Siemens can become one significant step towards achieving full isolation in multi-core systems. Due to its small code size, it is now much easier to understand the hypervisor and therefore implement new partitioning strategies into it.

The main contributions of this paper are:

- We present a methodology for measuring performance isolation of a system.
- A study on the performance isolation gained using the Jailhouse hypervisor.

Related work. We here identify previous studies that analyze shared resource contention caused by multiple cores, or address performance measurements on the Jailhouse hypervisor on ARM processors. Bansal et al. [1] investigated resource contention of the memory subsystem of the Xilinx ZCU 102 and proposes a Jailhouse based architecture to solve the contention. The authors effectively show a latency performance degradation of their benchmark when using multiple cores and propose mitigation techniques. In our work, we employ a different methodology, using the performance counting unit as a tool for identifying the sources of the performance degradation. Toumassian et al. [16] investigate the overhead of the Xen and Jailhouse hypervisors, where overhead is defined as Hypervisor performance/Linux performance. We complement this work, by deliberately adding the disturbing loads for estimating resource contention effects, while looking for application performance isolation. As listed by Deshane et al. [3], there exist a large body of reporting the impact of hypervisors on performance. However, since the Jailhouse hypervisor is relatively new, there is not so much research done on this subject. Up to our knowledge, there is no reporting of work investigating cache contention and memory bus contention in a Jailhouse environment, such effects being described as "yet to be measured" in a Linux Journal article [15]. Furthermore, there is no reported work trying to verify what Jailhouse can accomplish in the area of task isolation, wherefore we research here the performance degradation on a Linux system caused by CPU sharing.

10.2 Background

Shared resource contention has become an increasingly important topic due to the phasing out of single-core systems and the adoption of multi-core systems. Important shared resources can be divided into three categories: CPU, memory, and I/O [17] which may all be subject to contention. The CPU sharing takes place in the scheduling level, where two or more processes share the execution capacity of the same CPU. If one process executes and a higher priority task interrupts, the swapped out process will not get to execute anymore, and may, therefore, expose an increased latency. The second level of resource contention occurs in the memory layer of a computer and can come in the form of *thrashing* - a state where much of the processing time is spent on handling cache misses or page faults due to several processes/threads continuously replacing each other. The third level of resource contention occurs in the I/O layer and can be illustrated very well by the ARM v8 case where a generic interrupt controller (GIC) handles all general purpose interrupts (such as general purpose I/O interrupts).

Partition-based virtualization is one of the solutions that addresses the sharing of resources across multiple processes [17], [5], [12]. Hypervisors such as Xen [2] and KVM [6] can effectively partition the cores of a system such that the resource is protected from usage of processes which do not belong to the specific partition. These hypervisors come with an overhead [8] and a significant code size. New virtualization techniques such as the Jailhouse hypervisor give promise of better task isolation through statically disallowing inter partition sharing of resources and also come with a relatively small code size.

10.2.1 Jailhouse hypervisor

The Jailhouse hypervisor (version 0.1 released in august 2014) partitions hardware resources through virtualization, and enables asymmetric multiprocessing on top of the Linux system [14]. It also enables the insertion of *cells* through a kernel module. A cell is a virtual machine that is created in a partitioned environment. Once created, the host operating system loses knowledge of the core where the cell is created. In a similar fashion, programs running within the Jailhouse cell do not know that they run within a virtual machine, nor have they any knowledge of cores outside of the cell.

Fig. 10.1 shows a regular Linux system - a) and a Jailhouse partitioned system

which runs one Linux partition (core 0, 1, 2) and one real-time (RT) partition (core 3) - b).

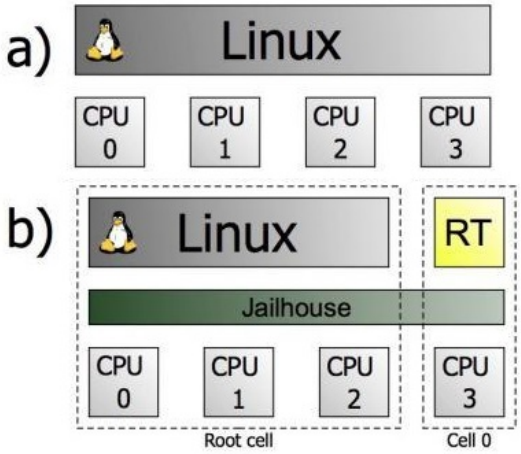

Figure 10.1: a) Usual Linux deployment. b) Linux with Jailhouse configuration [15].

10.3 Shared resource contention

We describe the performance degradation of a process in Equation 10.1, where performance is equal to the execution time of an application.

$$I = \frac{P}{C} - 1 \tag{10.1}$$

We denote I as the *isolation coefficient*, representing the resulting slow-down of the execution of a task in the presence of other tasks. P denotes the loaded performance of an application, and C is the baseline performance. Both C and P values are measured in time units; moreover, it is expected that the P will always be higher than C, that is, the execution time of an application will always be longer in the presence of additional load as compared to the "ideal" case when the application executes alone on the computing platform. It is also important to note that the measured values of both C and P are platform dependent. Measurements are relying on processor specifics such as cache memory mechanisms, clock frequency and bus bandwidth, but also on the operating system. Therefore, C should not be seen as an absolute value of the best achievable performance (that is, cross-platform), but instead, the

highest performance achievable using the respective setup. We refer to C as **baseline** in subsequent sections of the paper.

As an example, consider an application running on one core of a multi-core processor, exposing a baseline of 100ms. To perform tests on cache memory isolation, we apply a heavy cache intensive load, which runs on a different core than the application, and re-execute the application in these conditions. Both cores have a shared LLC. In case the loaded performance is observed to be 100ms, the isolation coefficient $I = 100\text{ms}/100\text{ms} - 1 = 0$. Hence, and the application is isolated from LLC disturbances. Alternatively, if the loaded performance is 110ms (for exemplification purposes), the isolation coefficient becomes $I = 110\text{ms}/100\text{ms} - 1 = 0.1 = 10\%$ which means that the application has suffered a 10ms performance penalty due to cache contention.

In the following subsections, we will discuss resource contention on shared resources, including CPU, cache, memory bus. We will discuss each shared resource in the context of a Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit using 4 Cortex A-53 cores, specified in Table 10.1.

Feature	Hardware Component
Core	4xArm Cortex A-53 @ 1.5GHz
	2xArm Cortex-R5 @ 1.4GHz
L_1I-cache	32 KB 2-way set assoc cache/core
L_1D-cache	32 KB 4-way set assoc cache/core
L_2-cache	1 MB 16-way set assoc. shared platform cache
MMU	L_1ITLB: 10 entries
	L_1DTLB: 10 entries
	L_2TLB 512 entries, 4-way set assoc.

Table 10.1: Hardware specifications Xilinx Zynq UltraScale+ MPSoC

10.3.1 CPU utilization

Two applications sharing the same CPU can have dramatic effects on either applications response time. When sharing the CPU, one task may get to execute up to 50% of compared to the non-shared situation. Thus, the response time of the application could increase to at least the double of the baseline. We can avoid the CPU sharing effect by not scheduling other applications to the same core. However, if all cores are currently loaded, it is not possible to enforce such a policy, since the newly created application needs an execution environment. Consider our ARM system with 4 cores, running App_1..App_4 on

core 0..3 respectively. In case a 5th application, App_5, enters the scheduling queue, there is no un-occupied core, which means App_5 has to share one of the cores with one of the other applications. This will increase the response time of both applications. This situation may not become a problem in real-time systems since tasks with high importance often are given a higher priority and will therefore not share execution time with other tasks during their respective time quanta. Thus, scheduling applications properly is usually a solution to this problem. Another solution can be static partitioning of the system, where the cores of one partitioned sub-system are hidden from another partitioned sub-system [11], disallowing partitions from using each other's designated cores.

10.3.2 Internal Memory Contention

The internal memory is often a source of execution time unpredictability - the so-called *jitter* - in multi-threaded systems [4]. Whenever the data requested by applications is not in the L_1D-cache or the L_2-cache, we need to fetch the data from the main memory. If the L_2-cache is already full, a cache-line is evicted from the cache to make space for the incoming data. Since the L_2-cache is shared between multiple cores, processes scheduled on different cores can evict the cache-lines of each other whenever the shared cache becomes full.

Within our ARM system, with a 1 MB L_2-cache, cache contention is exemplified as follows: App_1 and App_2 with a memory footprint of 1 MB each are executing on core 0 and 1 respectively. The applications are each using 1 MB of data, which, combined, is above the limit of L_2-cache - 1 MB. If the tasks are continuously running on different cores, App_1 will continuously try to write 1 MB of data into the shared cache. Since the cache is not large enough to contain the total amount of 2 MB data requested by both tasks, 1 MB of data will continuously have to be replaced according to the cache replacement policy. Cache coloring can be applied here, to restrict cache access of different applications to assigned cache lines only. Thus, one may mitigate problems such as performance losses [10], jitter [18], and even energy efficiency [9]. In our example though, this limits the amount of L_2-cache available to either of the applications.

10.3.3 Memory bus contention

The memory bus that interconnects the cache memory with the main memory is also a subject for contention. It is used for serving read and write requests from each core, which can become problematic when multiple memory intensive tasks are running on several cores. The bus can become a significant bottleneck concerning throughput, and a source of jitter.

Once again, consider the ARM system which has a measured bandwidth capacity of roughly 4.7 GB/s. The system hosts four applications (App_1, ..., App_4 running on core 1..4 respectively) which executes write operations at 2 GB/s individually. If the data is not present in the cache, it has to be fetched from the main memory via the memory bus. The bus, however, can only handle a certain amount of writes per second, as specified. Since we use multiple cores executing writes at 2GB/s, the bus bandwidth will be fully saturated. If any of the applications were the only one executing memory transactions, it could operate at the intended 2MB/s capacity. However, since multiple applications are executing, the bus has to distribute the capacity over the set of cores, which can dramatically decrease the individual memory throughput and increase the jitter of each application. It is possible to limit the effects of bus contention by restricting processes to execute under a certain memory bandwidth budget [19] [20] - with potential important overhead for each budgeted application.

10.4 Performance isolation

We have used a matrix multiplication of various sizes as the application to benchmark the isolation that can be achieved using the Jailhouse hypervisor. The execution time of the application is measured by inserting wall-clock timestamps at the start and at the end of the multiplication. Further, the matrix multiplication is co-executed with additional load programs denoted *leeches* to enforce shared resource contention. We use the previously defined Xilinx Zynq ZCU 102 platform (Table 10.1) running a Petalinux 4.9 kernel and reserving 2 GB of RAM for the Jailhouse hypervisor using the *mem* kernel argument.

In the following subsections we show isolation measurements for the CPU, L_2-cache and memory bus resources with the matrix multiplication running in unfavourable (leech-disturbed) execution environments and compare them to the baseline executions.

10.4.1 CPU isolation test

We devised a test including a kernel module to serve as a CPU stealing leech and a matrix multiplication to show the contention problems in a CPU. We exemplify the problems using the following scenario, assuming equal application priority.

1. Applications P_0, P_1, P_2 and P_3 are ready to execute.
2. The applications are pinned as following $P_0 \rightarrow C_0$, $P_1 \rightarrow C_1$, $P_2 \rightarrow C_2$, $P_3 \rightarrow C_3$.
3. Kernel application KP_5 becomes ready to execute, all cores are currently occupied.
4. The kernel has to chose one available core for KP_5, in this case, C_3 is chosen.
5. P_4 and P_5 now share the same core and execute

To instantiate the above contention scenario, we co-run a 256x256 matrix multiplication as workload together, with a calculation-heavy program called a CPU leech, implemented as a kernel module. Kernel modules often are executed at seemingly random times and also at a higher priority than user-space modules. The CPU-stealing leech performs 100000 random number calculations, searches for the highest value read and then goes to sleep for a specified amount of time. This process takes between 79-80 milliseconds to execute. Since the time measurement of the matrix multiplication is dependent on context switches from another workload, we will call the time measurement *response time* in this test case. We statically set the core affinity of the matrix multiplication and the CPU leech to the same core C_3.

We also execute the same tests using the Jailhouse hypervisor, where the matrix multiplication is run within a Jailhouse Linux cell executing on C_3. The results of the CPU isolation tests are depicted in Fig. 10.2 where the y-axis shows the response time of the matrix multiplication run under Linux (blue dash) compared to a matrix multiplication run within a Jailhouse Linux cell (orange dash). Each data point is the median response time of 50 executions. The y-axis is a logarithmic scale of the response time measured in milliseconds, and the x-axis shows the sleep timer of the kernel module - the period between executions. A low value on the Y-axis - meaning a low response time - would be better than a high value. The calculated isolation coefficient of the matrix multiplication is listed in Table 10.2.

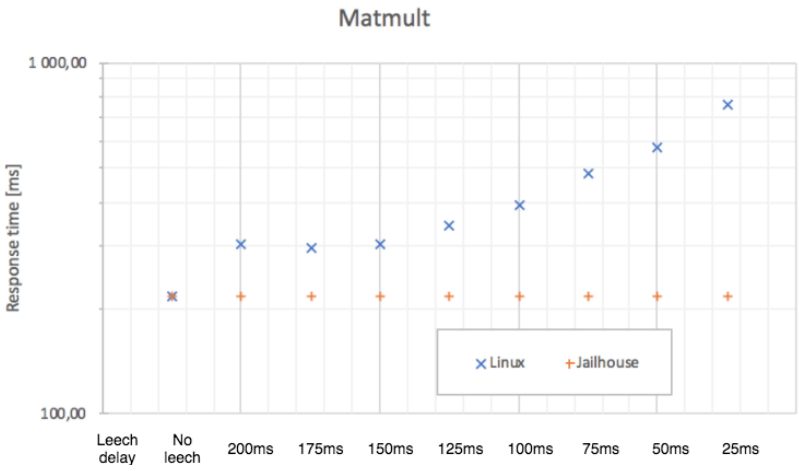

Figure 10.2: CPU isolation test

Sleep	I_{Linux}	I_{Jhouse}	**Sleep**	I_{Linux}	I_{Jhouse}
200	41,22%	0,62%	100	81,99%	0,79%
175	38,25%	0,15%	75	124,00%	0,50%
150	40,76%	0,57%	50	166,47%	0,86%
125	59,88%	0,57%	25	250,79%	-0.15%

Table 10.2: *I* coefficient in CPU contention test (percentage)

Fig. 10.2 shows a Linux matrix multiplication which suffers heavily from the CPU stealing caused by the leech, even at the relatively large sleep periods of 200 ms. In these conditions, according to Table 10.2 and using Equation 10.1, Linux alone offers an isolation coefficient of 0.40, which is an indicator of significant resource contention. The CPU leech will always get a high priority when ready to execute, running with kernel priority. Hence, when the associated sleep period goes under a certain value, the isolation coefficient even surpasses 0.50. When running the matrix multiplication within a jailhouse partition, however, the response time is almost constant, with an isolation coefficient of 0.0086, which is in the range of an error margin.

Concluding, the Jailhouse hypervisor performs as promised regarding the CPU isolation, while the Linux system shows a significant downgrade in the performance of the matrix multiplication, as expected, too.

10.4.2 L2-Cache isolation test

Here, we intend to provide a measurement of the isolation coefficient for the matrix multiplication, verifying to what extent it suffers of L_2-cache cache contention.

We use a 512x512 matrix multiplication for benchmarking workload, and a tweaked version of a maximum bandwidth benchmark called Tinymembench [13] as a leech, for loading the L_2-cache. The Tinymembench load continuously reads 32-bit integers from a N-sized buffer and writes them into another N-sized buffer. The isolation test was conducted as follows.

1. Run baseline execution of the matrix multiplication
2. Initialize cache load process with size N (initially 64 KB)
3. Assign cache load process to C_0
4. Start matrix multiplication on C_3
5. Re-iterate from step 1 and multiply size N by 2

The results of the matrix multiplication running within a regular Linux environment are depicted in Fig. 10.3, and the results of running it within a Jailhouse Linux cell are shown in Fig. 10.4. The graphs point the execution time (blue dash) on the left-hand side y-axis and the L_2-cache misses (orange dash) on the right-hand side y-axis. The x-axis marks the leech buffer size. The graphs also include error bars where the upper dash shows the maximum value, and the lower dash shows the minimum value of 50 measurements. As previously, low values are better than high values of the execution times. Also, a large error bar is worse than a small one, since small variability in both L_2-cache misses and execution time is preferable. Table 10.3 lists the calculated isolation for the matrix multiplication when co-run with the Tinymembench load.

We observe a typical "knee" effect, i.e., the performance degradation of the matrix multiplication halts at a certain point. This halt occurs when the matrix multiplication co-run with a L_2-cache leech cannot produce more cache misses, as every cache line request will be a miss. This comes to a full effect when N is 1 MB, which is aligned with the 1 MB-sized L_2-cache. From the isolation coefficient values- Table 10.3, we see almost no difference between the Jailhouse measurement and the Linux measurement. This is motivated by the fact that the Jailhouse hypervisor (in the reported version) does not mitigate

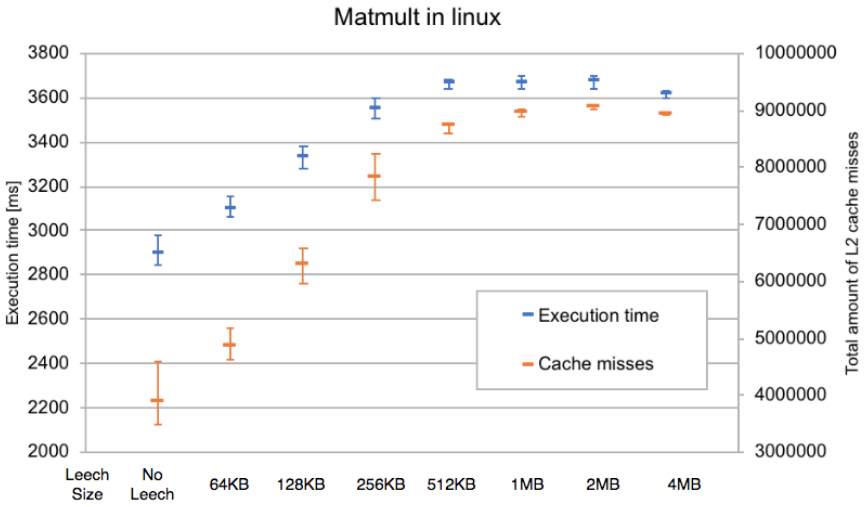

Figure 10.3: Linux L_2-cache isolation test

Size	I_{Linux}	I_{Jhouse}
128 KB	7,17%	7,74%
256 KB	15,27%	15,84%
512 KB	22,78%	22,33%
1 MB	26,62%	26,69%
2 MB	26,92%	26,87%
4 MB	25,14%	25,51%

Table 10.3: *I* coefficient in L_2-cache contention test (percentage)

this problem. Also, there is almost no difference in execution time, nor cache misses. This suggests that it is potentially is possible to migrate tasks from regular Linux system to a Jailhouse partition without having to re-calculate the execution characteristics of the algorithm.

10.4.3 Memory bus isolation test

In this section, we describe memory bus contention which occurs due to multiple processes on different cores requesting non-cached memory. In the previous test, we discovered the knee effect occurring at a buffer size of 1 MB, which means all data requested by a process will be a cache miss and it has to be fetched from the main memory through the bus. If multiple processes from different cores request data from the main memory, the bus has to arbitrarily

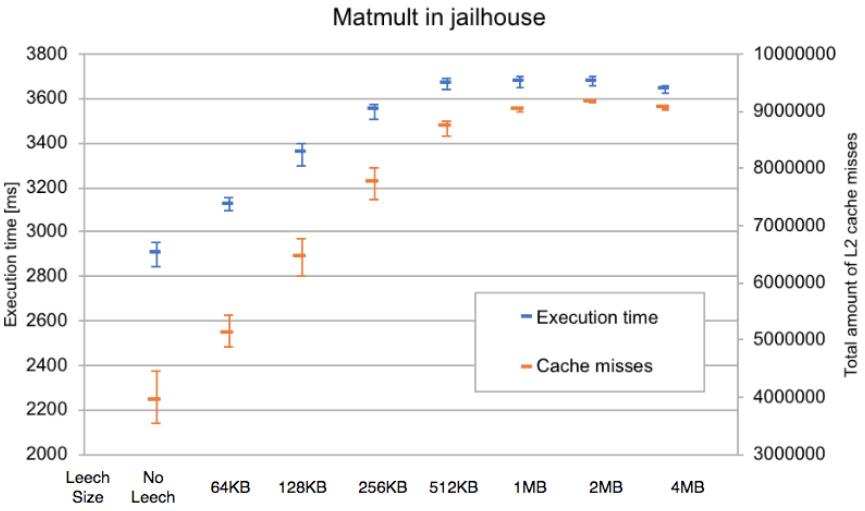

Figure 10.4: Jailhouse L_2-cache isolation test

chose which process gets the access. This may lead to further performance degradation. To investigate memory bus contention, we run a test as follows, where we employ the same kind of leech as previously, with a buffer size of 8 MB (or any size larger than the 1 MB limit described above).

1. Start a 512x512 matrix multiplication on C_3
2. Insert one memory bus leech on a non-occupied core
3. Repeat step 3 until all cores are occupied

To ensure that full cache contention occurs during the entire execution of the test, we measure the L_2-cache misses of the system. Their number should remain constant - any change reflecting the fact that there were also some cache-hits, which is to be avoided.

Fig. 10.5 depicts the results of the regular Linux matrix multiplication execution, and Fig. 10.6 depicts the results of the execution under Jailhouse protection. The left-hand side y-axis plots the calculated median execution time of 50 measurements, the x-axis shows the number of leeches inserted into the system and the right-hand side y-axis shows the L_2-cache misses of the system. The graphs also include error bars where the upper dash shows the maximum value and the lower dash shows the minimum value of the 50 measurements.

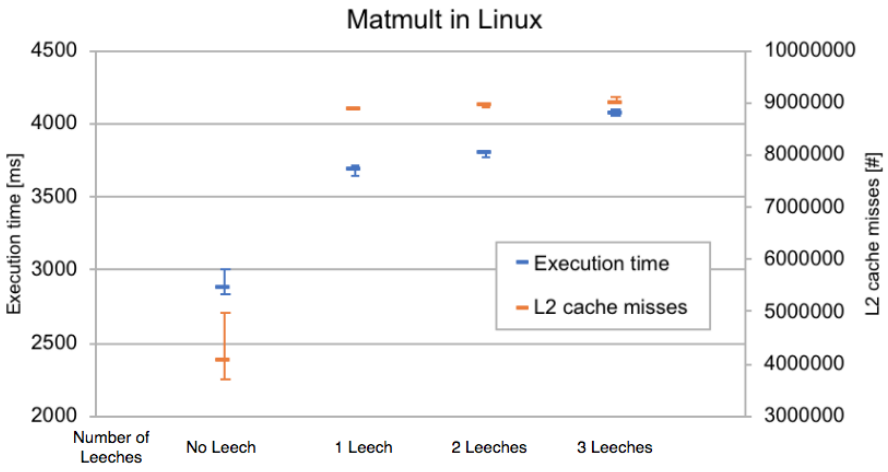

Figure 10.5: Linux memory bus isolation test

We list the calculated isolation coefficient for the matrix multiplication using regular Linux and Jailhouse in Table 10.4.

The graphs show a significant performance degradation of the matrix multiplication due to memory bus contention running in Linux as well as in Jailhouse.

The baseline execution time remains the same as in the matrix L_2-cache isolation case, since we used the same matrix size. Furthermore, the observed effects when using one leech are also similar to the L_2-cache isolation test, as the cache is fully loaded. However, the interesting effects on execution times occur when inserting two or more leeches. Firstly, we can read an isolation coefficient of 0,3168 and 0,326 for the Linux and Jailhouse matrix multiplications, respectively. The values mean that the Jailhouse hypervisor does not provide any sorts of bus isolation, as expected. In addition, the execution time of the matrix multiplication will be increased with any added leech. Once again, the performance impact of using the Jailhouse hypervisor is within a measurement error margin, suggesting that using the Jailhouse hypervisor does not come with any overhead penalties.

Table 10.4: *I* coefficient in Memory bus contention test, (Percentage)

Size	I_{Linux}	I_{Jhouse}
1 Leech	28,91%	25,96%
2 Leeches	31,75%	34,12%
3 Leeches	41,30%	43,50%

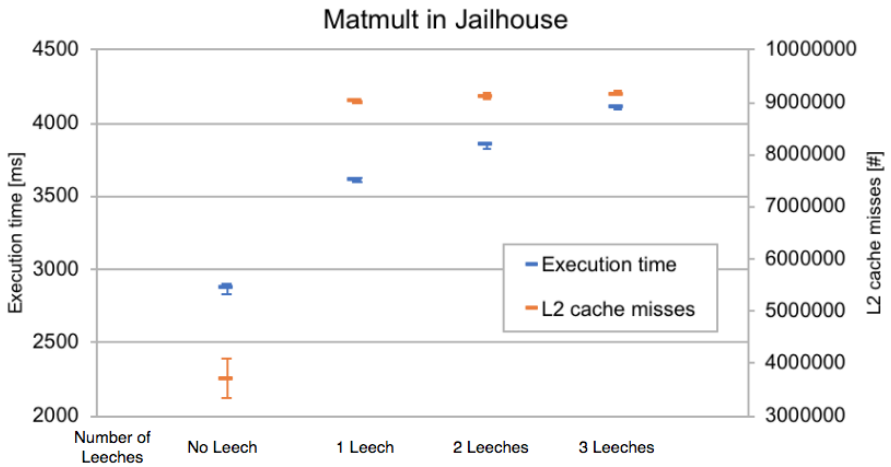

Figure 10.6: Jailhouse memory bus isolation test

10.5 Conclusion

We have measured the effects of contention on computing resources such as CPUs, L_2-cache and memory bus. As an example of an application with high need for both CPU and memory, we used a matrix multiplication. We executed the application in a standard Linux context and compared it with the execution in a Jailhouse hypervisor cell context. In order to test the isolation, we disturbed the application by executing leeches designed to consume particular computing resources.

Our measurements focusing on the CPU resource show that the Jailhouse hypervisor provides isolation between different partitions, enabling the application to exhibit a performance very close to the baseline even in the presence of leeches. Jailhouse does not, however, provide any memory bus or L_2-cache isolation. These said, there is a very small difference in performance degradation for the application execution between the Jailhouse hypervisor and a standard Linux system during heavy shared resource congestion. This further suggests that using Jailhouse in a heavily loaded shared resource environment provides an at least as performant execution context as Linux.

We leave investigating TLB, DRAM bank and I/O contentions for future work. There also exists a newly published patch [7] for Jailhouse which provides a cache coloring configuration for Jailhouse cells. Investigating the Jailhouse's page coloring mechanisms using our methodology is also relevant future work.

Bibliography

[1] A. Bansal, R. Tabish, G. Gracioli, R. Mancuso, R. Pellizzoni, and M. Caccamo. Evaluating the memory subsystem of a configurable heterogeneous mpsoc. *OSPERT 2018*, page 55, 2018.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[3] T. Deshane, Z. Shepherd, J. Matthews, M. Ben-Yehuda, A. Shah, and B. Rao. Quantitative comparison of xen and kvm. *Xen Summit, Boston, MA, USA*, pages 1–2, 2008.

[4] FAA. Addressing cache in airborne systems and equipment. accessed: 2019-11-04.

[5] S. Han and H.-W. Jin. Full virtualization based arinc 653 partitioning. In *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*, pages 7E1–1. IEEE, 2011.

[6] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.

[7] J. Kizka. Jailhouse google groups. accessed: 2019-11-04.

[8] J. Li, Q. Wang, D. Jayasinghe, J. Park, T. Zhu, and C. Pu. Performance overhead among three hypervisors: An experimental study using hadoop benchmarks. In *Big Data (BigData Congress), 2013 IEEE International Congress on*, pages 9–16. IEEE, 2013.

[9] S. Mittal, Z. Zhang, and Y. Cao. Cashier: A cache energy saving technique for qos systems. In *VLSI Design and 2013* 12^{th} *International Conference on Embedded Systems (VLSID), 2013* 26^{th} *International Conference on*, pages 43–48. IEEE, 2013.

[10] S. Perarnau, M. Tchiboukdjian, and G. Huard. Controlling cache utilization of hpc applications. In *Proceedings of the international conference on Supercomputing*, pages 295–304. ACM, 2011.

[11] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer. Look mum, no vm exits!(almost). *arXiv preprint arXiv:1705.06932*, 2017.

[12] S. Santos, J. Rufino, T. Schoofs, C. Tatibana, and J. Windsor. A portable arinc 653 standard interface. In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA* 27^{th}, pages 1–E. IEEE, 2008.

[13] S. Siamashka. https://github.com/ssvb/tinymembench. *Retrieved January*, 2019.

[14] V. Sinitsyn. Understanding the jailhouse hypervisor, part 1. *https://lwn.net/Articles/578295/*, 2014.

[15] V. Sinitsyn. Get to know jailhouse. *https://www.linuxjournal.com/content/jailhouse*, 2015.

[16] S. Toumassian, R. Werner, and A. Sikora. Performance measurements for hypervisors on embedded arm processors. In *Advances in Computing, Communications and Informatics (ICACCI), 2016 International Conference on*, pages 851–855. IEEE, 2016.

[17] S. H. VanderLeest. Arinc 653 hypervisor. In *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA* 29^{th}, pages 5–E. IEEE, 2010.

[18] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: a dynamic cache partitioning system using page coloring. In *Parallel Architecture and Compilation Techniques (PACT), 2014* 23^{rd} *International Conference on*, pages 381–392. IEEE, 2014.

[19] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE* 19^{th}, pages 55–64. IEEE, 2013.

[20] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, 2016.

Chapter 11

Paper E
Automatic Quality of Service Control in Multi-core Systems using Cache Partitioning

Jakob Danielsson, Tiberiu Seceleanu, Marcus Jägemar, Moris Behnam, and Mikael Sjödin. In *45^{th} Annual Conference of the IEEE Industrial Electronics Society (IECON)* IEEE, 2019.

Abstract

In this paper, we present a last-level cache partitioning controller for multi-core systems. Our objective is to control the Quality of Service (QoS) of applications in multi-core systems by monitoring run-time performance and continuously re-sizing cache partition sizes according to the applications' needs. We discuss two different use-cases; one that promotes application fairness and another one that prioritizes applications according to the system engineers' desired execution behavior. We display the performance drawbacks of maintaining a fair schedule for all system tasks and its performance implications for system applications. We, therefore, implement a second control algorithm that enforces cache partition assignments according to user-defined priorities rather than system fairness. Our experiments reveal that it is possible, with non-instrusive (0.3-0.7% CPU utilization) cache controlling measures, to increase performance according to setpoints and maintain the QoS for specific applications in an over-saturated system.

11.1 Introduction

Hardware resources are often limited for automotive control-systems. Especially when executing multiple applications on electronic control-unit, resource allocation must be carefully considered before deployment to achieve the desired Quality of Service (QoS). Multi-core computers are gaining increased popularity for the automotive industry due to increased available resources (processor cores) on the same chip. Multi-core processors offer higher computational capacity than their single-core predecessors while utilizing less size, weight, and power (SWaP) than older single-core ones. Multi-core computers often utilize a shared resource principle, where the ownership of multiple resources such as cache memories and the memory bus are shared simultaneously across different cores. The resource-sharing principle makes multi-core's prone to a state called shared resource contention, which causes severe execution-time fluctuations for applications and is seen as one of the major bottlenecks for bringing multi-core's into time-critical computing.

It is possible to counter shared resource contention using partitioning techniques such as cache coloring for the cache [12] and thus make multi-core systems more time-predictable. However, the partition boundaries are hard to assign appropriately at system boot since an application can change run-time behavior during the lifespan of the system. As such, too small partition sizes can cause an application to display lower QoS than desired, and a too large partition sizes wastes hardware resources without any QoS gain.

In this work, we try to automate the allocation of cache memory to meet QoS needs. We present experiments containing two different distribution policies; fair and prioritized. Fair distribution prioritizes assigning cache partitions based on the current performance of all system applications and tries to optimize the allocated cache size such that all applications reach as close as possible to their maximum performance. Priority distribution instead distributes cache memory based on an application's setpoint QoS and prioritizes this application to receive cache partition space while the setpoint QoS is not met. Our experiments are done for over-saturated systems where it is impossible to reach maximum QoS for all applications. We demonstrate that our controller can instead reach and maintain a setpoint QoS for our system using our fair strategy. We further demonstrate how to prioritize an application and meet QoS needs for one specific, prioritized application. We list our contributions as follows:

- An automated process of monitoring application performance continuously, without the need of a complex communication scheme.
- A cache partitioning control scheme that automatically adjusts the cache partition size of monitored applications to meet their respective QoS.
- A working implementation in Linux using the above mentioned contributions.

11.2 Background

11.2.1 Application Quality of Service

We define *QoS* as a function of the number of instructions retired in a time interval. One instruction retired means that the instruction has passed through all stages within the processor pipeline. This means that for higher QoS, an application will execute/retire a higher number of instructions. We can configure the performance monitor counters (PMC) of a CPU to monitor the instructions retired rate for a process ID (pid), and thus monitor the performance of an application online. However, the measurement approach means that we put requirements on the application's functionality – network and I/O applications that utilize busy-wait loops typically display a high number of instructions retired in the loops but not doing practical work. The prerequisites for our performance measurement approach to work is that the applications are not utilizing busy-wait loops but instead continuously doing "actual" processing. We assume our the QoS of our applications is correlated to the number of instructions retired, where an increase in number of instructions retired leads to a decrease in response time.

11.2.2 Cache contention

Cache memories are relatively small, temporary memory storage units that affects the system's overall performance. Cache memories in multi-core systems are prone to reach a state called cache contention, which causes dramatic execution-time fluctuation of system applications [3] and can cause problems to a system that expects execution-time predictability. The main reason for cache contention is the small memory size of the cache combined with simultaneous utilization from multiple tasks on different cores. The cache's are so small that it is *very* improbable that an entire application's memory foot-print

fits within the cache and it is almost a certainty that the cache will become full at some point during an application's execution.

Cache memories implement a data eviction policy to mitigate out-of-cache memory scenarios and replaces old data with new data when the cache is full. The cache selects one data block (cache line) according to a policy (e.g., LRU, random), evicts the selected cache line from the cache and then finally inserts the new data onto the address of the previously evicted data. Data replacement is necessary to counter the small memory space of a cache, but is also the main reason for severe execution-time fluctuations and QoS decrease.

Execution-time fluctuation often appears when two or more cache dominant applications utilize the same cache memory [2]. Consider the following scenario; two applications App_1 and App_2 executes simultaneously in a dual-core system with a 4 MB cache. App_1 runs on core 1 and App_2 runs on core 2. Both applications require a memory footprint of 4 MB– i.e. the same size as the available cache and both application have a cache usage that is linear with the execution. The cache will be full and start to evict data that belongs to either App_1 or App_2 once the applications have executed roughly half of their execution. The data evictions means the data is no longer present within the cache and needs to re-fetched from the main memory into the cache if referenced again, which has a significant latency.

Cache contention causes dramatic execution-time fluctuations for applications in multi-core systems [2] and is one of the major bottlenecks for introducing multi-core chips in to time-critical computing. In this paper, we focus of the shared last-level cache (LLC) as the location of the contention which can be partitioned according to the page-coloring algorithm that mitigates cache contention [12].

11.2.3 Cache partitioning

The main idea behind cache partitioning is to reserve a portion of the cache memory to only certain processes such that shared cache contention never occurs. There exist a variety of solutions to implements cache partitioning, including the static, hardware-supported cache way-partitioning, MMU-based page coloring [12] [5] and also the programmatic cache locking solution [10]. In this paper, we utilize page coloring; an MMU-implemented a policy that redirects how page addresses are translated into the cache memory. There exists a large body of variations on page coloring including Palloc [13], Coloris [12], Jailhouse hypervisor adaptation [5] etc. Page coloring creates bor-

ders in the cache memory disqualifying processes from accessing certain data blocks in the cache memory (cache-lines).

We exemplify page coloring using three applications (A,B, and C) in Figure 11.1 with a cache that contains nine cache-lines. The figure shows how the

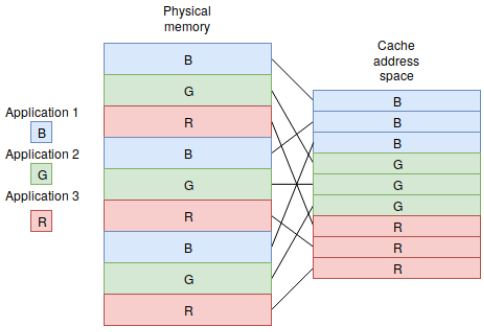

Figure 11.1: Cache coloring

MMU maps addresses to the cache in a page-colored environment. Memory requests that belong to application 1 are only allowed to access cache lines 1-3 while application 2 is only allowed to access the cache lines 4-6. Page-coloring thus means an application can only evict its' data from the cache memory and not by other applications on different cores.

We showcase the effects of cache contention in Fig. 11.2 and illustrate how these effects are countered using page coloring. The figure shows two experiments that runs 150 executions of one 256x256 matrix multiplication on core 0. The blue squares marks the execution time of a matrix multiplication running in a non-isolated environment, while the orange crosses marks the the execution time of a matrix multiplication running in a cache partitioned environment. We generate cache contention by starting another 256x256 matrix multiplication at iteration 75 on core 1.

Cache contention has a dramatic effect on the non-isolated matrix multiplications' (blue squares) execution time. The figure shows how the execution-time for the matrix multiplication increases by 17 milliseconds, just from running another matrix multiplication on another core. The cache partitioned version is roughly 5-6 milliseconds slower than our un-partitioned case, but remains unchanged when the new matrix multiplication. Cache partitioning however comes with drawbacks in terms of complexity which causes execution-time overhead as (5-6 ms in the example). The cache memory is also very small, which means the partition sizes for different applications must be handled with utmost care to avoid wasting a valuable resource. The total number of available

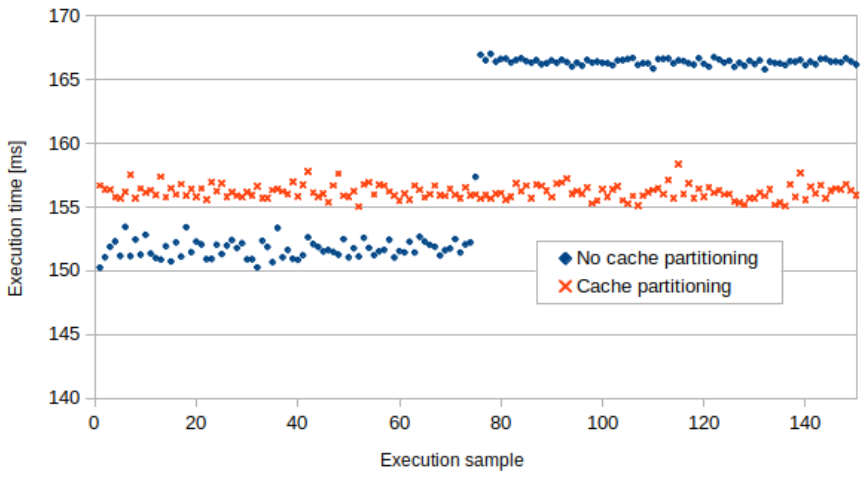

Figure 11.2: Cache contention

colors depends on the way-set associativity and also the total amount of cache space. The number of available cache partitions for a processor is calculated according to Equation 11.1 [12].

$$Nr.\ of\ Colors = \frac{Cache\_size}{Cache\_ways * page\_size} \tag{11.1}$$

11.2.4 Related work

There exists a large body of dynamic cache partitioning studies [12, 13, 6, 8, 9] that investigates how to optimize cache partitions to achieve maximum performance of the SPEC CPU benchmark suite. However, it is hard to reach maximum performance in an over-saturated system where all cache-partitions are assigned; wherefore, we focus on creating a controller that lets the system engineer decide the performance thresholds. Other related works focuses on enforcing quality of service [4] and isolation [14] through bandwidth restrictions. Kloda et al [5] introduces page-coloring into the Jailhouse hypervisor, which also introduces an entirely new dimension to solving cache contention as the cache partitioning spans overall application that belongs to a specific guest OS. However, the solution is limited to re-partitioning at guest OS boot, which makes it dynamic but less flexible. Our work differs from the previous work since we introduce a priority fashion-based assignment policy into the

cache allocation policy. Our goal is not to optimize overall system throughput but to provide isolation for applications while prioritizing performance for specific applications. Xu et al. [11] presents CaM, a resource partition allocation scheme using the intel's built-in cache partition utility called CAT and combines it with the memory bus reservation scheme memguard [14]. CaM proposes an algorithm that contains multiple procedures that optimizes task schedulability in a system. The main similarities between our work and CaM lie in resource allocation and load balancing. CaM takes the approach of allocating the minimum partition size to tasks executing on different cores and then re-allocates partitions until all tasks are schedulable. CaM also executes load balancing by migrating tasks from unschedulable cores to schedulable cores. CaM presents WCET guarantees and focuses on the schedulability of tasks. Our work instead focuses on tweaking the performance of specific applications in an oversaturated system in an online fashion. Our controller does not evaluate all possible task permutations in a system but instead focuses on tweaking the cache partition size of already running applications to satisfy performance needs.

11.3 Cache partition distribution

Cache partitioning offers isolation and counters execution-time fluctuations that happen as a consequence of cache contention. Cache partitioning, however, comes at the expense of performance degradation due to a complex memory management mechanism. Allocating cache partitions statically to applications is the most simplistic distribution policy. However, it can be very non-optimized as it is hard to assign suitable partition sizes beforehand unless performing time-consuming exhaustive searches [1]. Various online approaches instead tune the cache partition sizes to optimize application execution time [12] and maximizes system throughput.

There are industrial use-cases where the maximum throughput of the entire system is not the primary goal but is to instead maintain the QoS for one or perhaps two particular applications. Additional performance benefits to the system are just bonuses. Consider a simplistic multi-core system for an autonomous vehicle that contains three applications executing on different cores on the same chip; App_1 – feature detection algorithm that detects visual obstacles; App_2 – stores log-metrics in a database. App_3 – DPDK that sends and receives log-data packets over the network.

App_1 has the most critical task in detecting obstacles for the autonomous ve-

hicle, while App_2 and App_3 posts log data. These applications, however, run in the same multi-core system and thus share the same cache. Assigning cache partitions based only on increasing system overall performance (that is, increasing the number of instructions retired in a time interval) can lead to a scenario where App_2 and App_3 reserves all available cache partitions, while App_1 only gets one cache partition. Distributing the system fairly based on cache usage can also lead to the same scenario if App_2 and App_3 utilizes the cache more than App_1. Both scenarios will lead to a decreased QoS for the feature detection algorithm while the system's overall performance increases. We target our use-case towards systems that prioritizes QoS for specific applications above an overall system throughput.

11.4 Implementation

We implement our QoS cache partitioning controller in C using a petalinux 4.14 kernel. The controller utilizes the performance API (PAPI) [7] for monitoring the PMC; the cgroup interface for controlling an application's core affinity, and also the palloc [13] interface for adjusting the cache partition sizes. Our primary focus lies not in optimizing performance but to meet a specific application's QoS. We present our controller architecture in Fig 11.3.

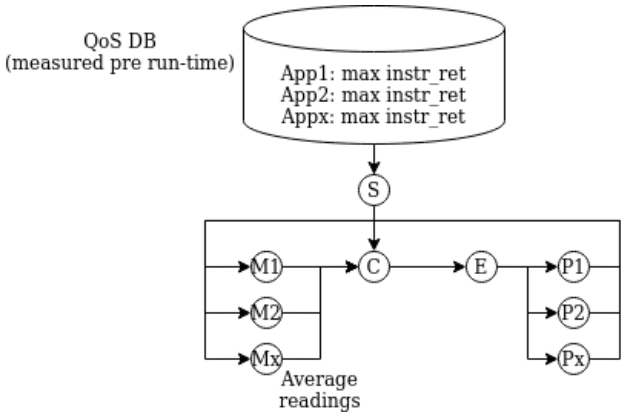

Figure 11.3: Cache partition controller architecture

The controller samples the number of instructions retired (QoS) of all system applications (M1..Mx) every 20 ms and calculates the average QoS every 100 values. We assume each application to run in parallel on different cores. Our controller utilizes a QoS database to provide an estimate setpoint for the con-

troller. The QoS database contains instructions retired measurements of all system applications and is measured pre-execution. However, the purpose of the QoS database is only to provide estimates for QoS setpoints, as they might not be theoretically achievable due to the limited cache space on-chip. The system engineer has to make the final verdict for each application on valid setpoints and use the database values as a reference for deciding a setpoint QoS. The controller compares the setpoints (S) of all running applications to the current QoS and selects two applications, one application that receives cache partition space and one application that loses cache partition space in step C. We implement two different controller modes that select applications for cache partition re-distribution; fair-oriented and priority-oriented.

Fair-oriented In this mode, we compare the differences between an application's current performance and its setpoint. The controller will increase the cache partition size for the application that displays the greatest difference between the setpoint and the current performance and decrease the cache partition size for the application that displays the least difference.

Priority-oriented In this mode, the system engineer needs to assign priorities to each application actively. The application that has the highest priority will always receive cache partition size first-hand. The controller will assign cache partition space to lower priority applications only when the higher priority applications display a QoS equal to the setpoint.

The change in cache partition size for the two selected applications is always 1 and the minimum cache partition is one for each application. If an application already has the minimum cache partition size, another application will instead be selected. The controller actuates the cache partition space of the two selected applications in step E utilizing the palloc API. The outcome from the cache partition actuation is a cache partition space for each application (P1..Px).

11.5 Experiment setup

We utilize the Xilinx-zynq zcu102 evaluation kit as testbed platform, with the processor specifics as of Table. 11.1. Our chip provides a 16-way set associative cache, which means we may consider 16 ($1\ \text{MB}/16{*}4\ \text{KB} = 16$) available colors in the system according to equation 11.1.

Table 11.1: Hardware specifications Xilinx Zynq UltraScale+ MPSoC

Feature	Hardware Component
Core	4xArm Cortex A-53 @ 1.2GHz
L_1I-cache	32 KB 2-way set assoc cache/core
L_1D-cache	32 KB 4-way set assoc cache/core
L_2-cache	1 MB 16-way set assoc. shared Last-level Cache
MMU	L_1ITLB: 10 entries L_1DTLB: 10 entries L_2TLB 512 entries, 4-way set assoc.

11.5.1 Test applications

Our execution scenario is inspired from industrial use-cases that execute applications on different cores. The system contains resource draining applications, that will drain the entire cache and as such cause severe execution-time jitters for other applications in the system. We exemplify the industrial use-case with three continuously running applications, two cache draining matrix multiplications that are common in computer graphics as synthetic loads and one feature detection algorithm, SUSAN to serve as a realistic load, listed as follows:

1. Matmult$_{ijk}$ (200x200) - Naïve implementation of the matrix multiplication that utilizes the traditional IJK traversion strategy.

2. Matmult$_{ikj}$ (100x100) - Cache prefetcher friendly traversing strategy of a matrix multiplication, designed to generate a higher cache hit rate than the naïve version. We chose a different size of this matrix multiplication to show more diverse results.

3. SUSAN - This application represents our realistic application and is used to detect corners in a frame. It is commonly used combination with other algorithms to identify visual obstacles for autonomous vehicles.

11.5.2 Controller setup

The controller is run as a standalone process that is running on its own core. It continuously monitors the applications' performance counters (instructions retired and L_2-cache misses) every 20 milliseconds – the sampling rate is a trade-off value. More frequent sampling rate reduces the controller's sleep

time and thus results in a significant CPU utilization increase. Less frequent values will instead decrease the controller's responsiveness since we are dependent on average samples to estimate the current performance. In this paper we wanted to maintain a CPU utilization below 1% while still being able to re-partition on a second basis, wherefore we chose 20ms. The controller stores the performance counters in a history database, which is used for calculating the average readings. We calculate the average performance counter readings based on 100 samples from the history database and use these average readings as basis for the re-partitioning decision. Table 11.2 summarizes the controller variables for our tests.

Table 11.2: Controller configuration

Property	Value
Sampling frequency	50HZ
Average window size	100

11.6 Partitioning experiments

In this section, we perform several experiments to show the benefits of an online partitioning controller. We perform four different experiments, including a baseline experiment, a proof-of-concept experiment focusing on application fairness, a QoS-focused cache distribution policy, test and finally a priority-based cache distribution policy. We affine each application to different cores; Cache partition controller (core 0), Matmult$_{ijk}$ (core 1), Matmult$_{ikj}$ (core 2) and SUSAN (core 3). The controller has a CPU utilization of 0.3-0.7% and always runs using one cache partition. Due to the controller's CPU low utilization, it is possible to run other other applications on the same core as the controller if 0.3-0.7% loss of CPU utilization is acceptable. In this paper we focus only on partitioning the cache, wherefore we opt out of optimizing scheduling applications together with the controller.

11.6.1 Initial experiment

Here, we present the setpoint QoS of the applications utilizing the maximum available cache partitions for each application. This value will be our reference QoS and used to compare the quality of a cache partition. We measure the maximum QoS by monitoring the number of instructions retired while all available

partitions are assigned to an application running in isolation. We sample the instructions retired every 20ms for 10 seconds and then calculate the average instructions retired. We use 10 seconds as interval to capture PMC events of at least 10 full iterations of each application. We present the reference values in Table 11.3.

Table 11.3: Cache partition maximum configuration

Application	Partition size	Reference QoS
Matmult$_{ijk}$	15	$5.4 * 10^6$
Matmult$_{ikj}$	15	$8.22 * 10^6$
SUSAN	15	$10.4 * 10^6$

The table shows the average number of instructions retired per 20 milliseconds of our applications, we denote this metric as **reference QoS**. However, the conditions of this experiment are not possible in a real system with concurrently running tasks, as we only have 15 available cache partitions and cannot distribute 15 colors to all concurrently running tasks without risking cache contention through cache-partition sharing.

11.6.2 Naïve cache partitioning

We can statically assign cache partitions in a naïve fashion by distributing the available cache partition space equally to all concurrently running applications, see Table 11.4.

Table 11.4: Initial setup

Application	Partition
Controller	1
Matmult$_{ijk}$	5
Matmult$_{ikj}$	5
SUSAN	5

In our naïve scenario, we split all available cache partitions among our different applications, which means our test applications receives 5 cache partitions while the controller receives 1. Table 11.5 shows the number of instructions retired per 20 ms (denoted as QoS), the L_2-cache misses per 20 ms, and the difference in QoS compared to the reference QoS for each application.
The table shows how an initial cache partition setup changes the QoS of our applications compared to the reference QoS in our previous experiment. Matmult$_{ijk}$ performs worst (due to nature of the naïve traversion strategy) comparing to the reference QoS, and SUSAN performs best.

Table 11.5: Performance comparison: reference versus equal partitions

Application	QoS	L_2-cache misses	Diff
Matmult_{ijk}	$3.23 * 10^6$	27901	41 %
Matmult_{ikj}	$7.19 * 10^6$	45380	13%
SUSAN	$9.83 * 10^6$	81073	6.2%

11.6.3 Fair distribution

The equally shared cache distribution experiment shows a significant QoS degradation as compared to the reference QoS. We introduce a control mechanism to regulate the cache partition sizes according to the distance to the reference QoS. The controller balances the QoS of the applications to minimize the difference between the application's current QoS, and their reference QoS. We list the controller steps as follows:

1. Monitor current QoS of all applications in the system
2. Select application with highest difference compared to the reference QoS (App_{high})
3. Select application with lowest difference compared to the reference QoS and cache partition size > 1 (App_{low})
4. Increase partition size of (App_{high}) by one and decrease partition size of (App_{low}) by one
5. Go to step 1

The above algorithm embraces fairness, prioritizing poorly performing applications over better-performing applications. Figures 11.4, 11.5 and 11.6 depicts the cache partitioning assignments done by the controller over a time-period of 90 seconds. The red line marks the current average QoS on the left-hand side y-axis, the green line marks the reference QoS as measured in the initial experiment, and the blue line marks the cache partitioning sizes on the right-hand side y-axis.

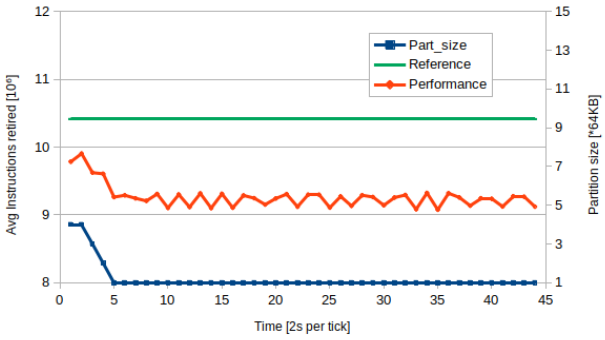

Figure 11.4: Susan fair cache partitioning

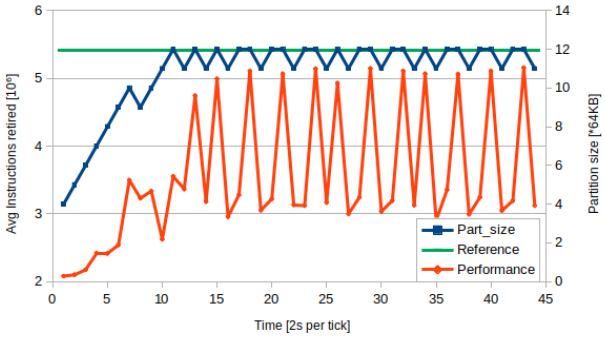

Figure 11.5: Matmult$_{ijk}$ fair cache partitioning

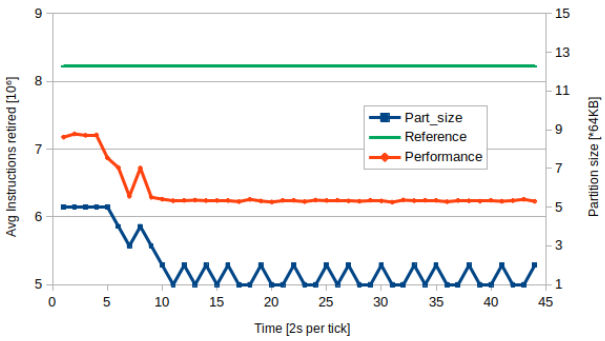

Figure 11.6: Matmult$_{ikj}$ fair cache partitioning

The three figures show an example of an over-saturated system as the controller cannot assign partitions that meets any reference QoS. The difference of Matmult$_{ijk}$ remains the highest until a cache partition size of 11. Once this mark is met, the controller starts continuously change partition size between Matmult$_{ijk}$ and Matmult$_{ikj}$. The algorithm partitions the system fairly, but fails to meet the QoS requirements of any application. Figure 11.6 displays high performance fluctuations due to the sensitivity of the performance to the cache size, we discuss this further in Section 10.5.

11.6.4 Reference distribution

An application's desired QoS does not necessarily have to be the maximum achieve-able QoS. A system engineer can, for example, decide that it is acceptable that a task is operating at a percentage value of its maximum capacity when cache isolation is more prioritized. In this subsection, we tune down the expectations of our two matrix multiplications and instead use a "desired QoS" as metric for the controller to chase. We leave SUSAN's desired QoS unchanged at 10.4 million instructions per 20 ms. We show these new desired QoS values in Table 11.6 and compare them with our reference values.

Table 11.6: Initial setup

Application	Reference	Desired	% Difference
Matmult$_{ijk}$	$5.4 * 10^6$	$3 * 10^6$	55%
Matmult$_{ikj}$	$8.22 * 10^6$	$7.5 * 10^6$	91%
SUSAN	$10.4 * 10^6$	$10.4 * 10^6$	100%

The table shows that we have tuned down the QoS requirement of Matmult$_{ijk}$ by 45% to an average of 3 million instructions per 20ms. We have also tuned down the requirement of Matmult$_{ikj}$ by 9%, to 7.5 million instructions per 20ms. In Figures 11.7, 11.8 and 11.9 we show how our controller operates with these new QoS requirements.

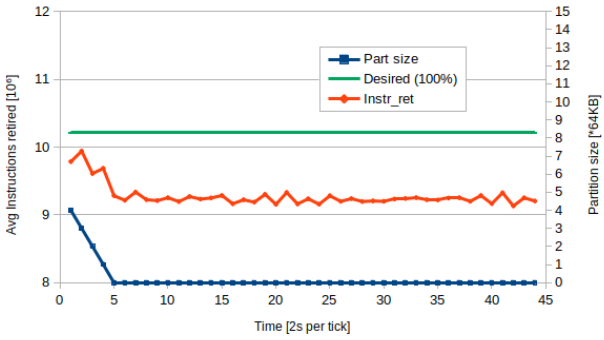

Figure 11.7: SUSAN 100% target performance

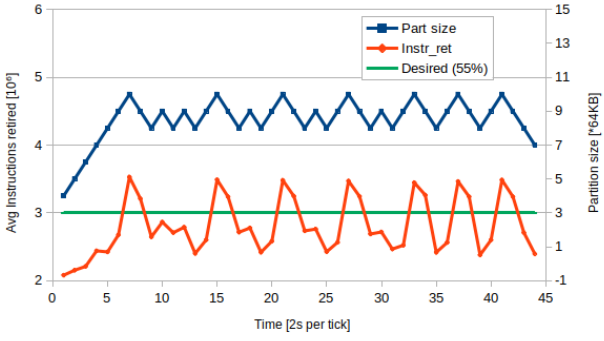

Figure 11.8: Matmult$_{ijk}$ 55% target performance

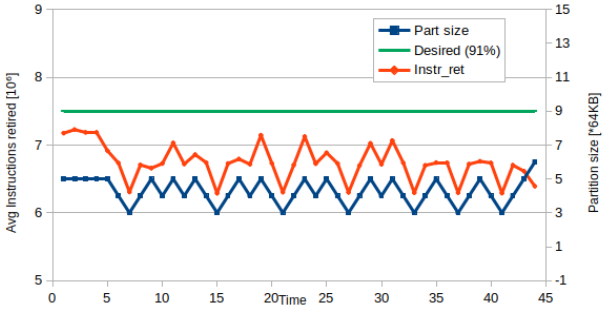

Figure 11.9: Matmult$_{ikj}$ 91% target performance

The figures show how the controller adapts the partitions according to the new desired QoS values. SUSAN still gets the minimum number of partitions, but Matmult$_{ijk}$ and Matmult$_{ikj}$ present a different scenario. Matmult$_{ijk}$ gets priority on receiving partitions first-hand since the distance to the desired QoS is highest. Matmult$_{ikj}$ starts to receive partitions from Matmult$_{ijk}$ at controller iteration six.

11.6.5 Priority distribution

Different applications in a system can be of different importance. Our system utilizes two matrix multiplications as synthetic loads and one "real" scenario application, SUSAN. In this experiment, we assign priorities to our applications to force partitions into a specific application. We chose SUSAN to receive the highest priority, Matmult$_{ikj}$ to receive medium priority, and Matmult$_{ijk}$ to receive low priority. Introducing priorities means we also shift our distribution rules, presented as follows:

1. Monitor the QoS of active tasks in the system
2. Select the highest priority application (App$_{high}$) that does not have a current QoS higher than a desired QoS
3. Select the lowest priority application that has cache partition size > 1 (App$_{low}$)
4. Distribute one cache partition from App$_{low}$ to App$_{high}$
5. Go to step 1

Once the high-priority application meets its target QoS, the controller will actively shift focus to the second-highest priority task and so on. Our priority policy means a medium priority task will only get partitions once the high priority task has its QoS requirements fulfilled etc. We exemplify the priority distribution policy using a QoS threshold in our applications. The controller will shift cache distribution focus once an application runs at a higher QoS than its threshold. Table 11.7 presents the experiment setup and contains application priorities and QoS threshold values. Matmult$_{ijk}$ has a non-applicable threshold since it is the lowest priority.

SUSAN has the highest priority, Matmult$_{ikj}$ has medium priority and Matmult$_{ijk}$ has low priority. Once SUSAN counts a presents a higher count

Table 11.7: Initial setup

Application	Priority	Threshold	Value
Matmult_{ijk}	Low	N/A	N/A
Matmult_{ikj}	Medium	95%	$7.5 * 10^6$%
SUSAN	High	95%	$9.95 * 10^6$

of instructions retired than $7.5 * 10^6$ (95% of measured max), Matmult_{ikj} will start to receive partitions. Figures 11.10, 11.11 and 11.12 shows the cache partition distributions for SUSAN, Matmult_{ijk} and Matmult_{ikj} using our prioritization policy.

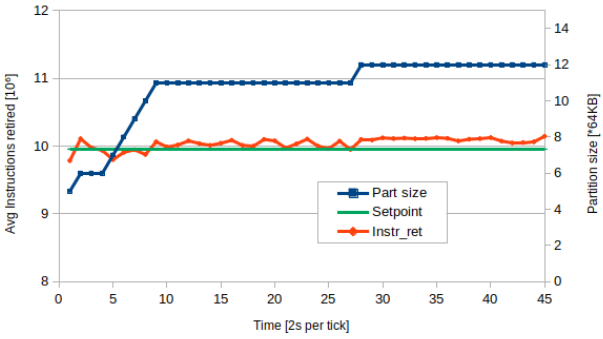

Figure 11.10: SUSAN high priority

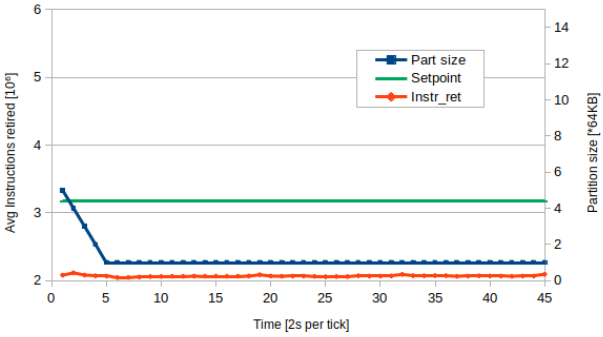

Figure 11.11: Matmult_{ijk} low priority

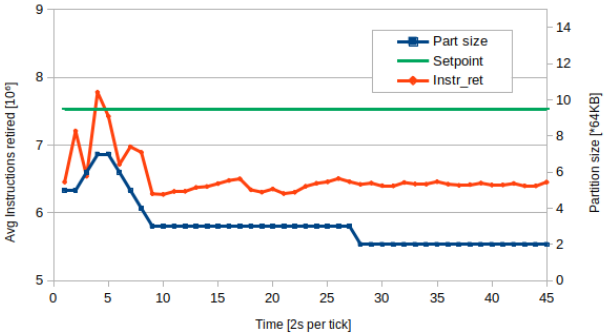

Figure 11.12: Matmult$_{ikj}$ medium priority

These experiments show a different cache partitioning distribution compared to the previous two experiments. SUSAN, now on high priority,receives a size increase at the first controller iteration, which increases the QoS to above the threshold. Since SUSAN now is above the threshold, Matmult$_{ikj}$ receives cache partitions from Matmult$_{ijk}$ for two iterations while also increases the QoS above the desired threshold. SUSAN, however, displays a QoS degradation during this time and is prioritized once again for cache partitions. This time, SUSAN takes cache partition size from Matmult$_{ijk}$ for six iterations, and the QoS finally hits the QoS threshold again. At iteration 30, SUSAN's QoS once again is below the threshold and thus receives another cache partition from Matmult$_{ijk}$. From this point, the controller does not change the cache partition distribution.

11.6.6 Equal priority distribution

Our last experiment presents our prioritization policy when applications run the same priority. This policy presents the most complex problem since we here combine both fairness and priority. When two applications have the same priority, we trigger the fairness calculation and calculate the application's distance to its desired QoS. In this experiment, we assign SUSAN the same priority as in the previous experiment (high), but we lower the threshold by 0.2% to create a more interesting execution scenario. We furthermore lower the priority of Matmult$_{ikj}$ to low, see Table 11.8 for experiment specification. We maintain the desired QoS from our previous experiment.

Table 11.8: Initial setup

Application	Priority	Threshold	Value
Matmult$_{ijk}$	Low	N/A	N/A
Matmult$_{ikj}$	Low	N/A	N/A
SUSAN	High	93%	$9.95 * 10^6$

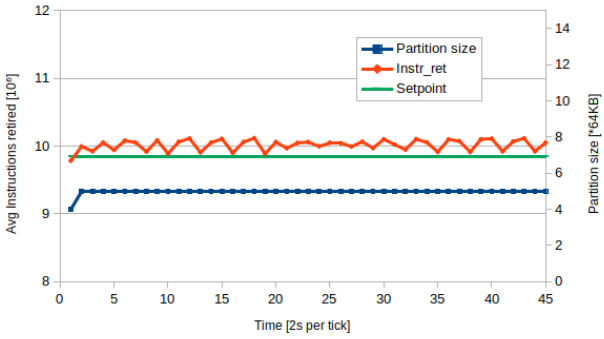

Figure 11.13: SUSAN high priority

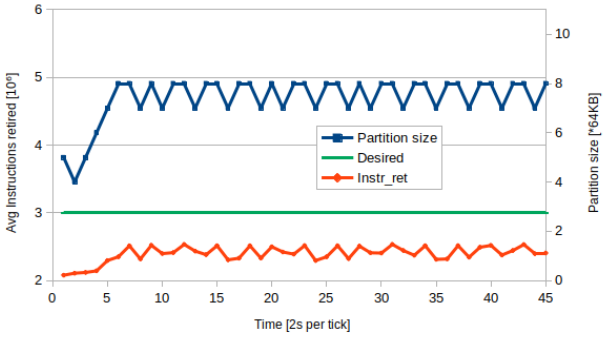

Figure 11.14: Matmult$_{ijk}$ low priority

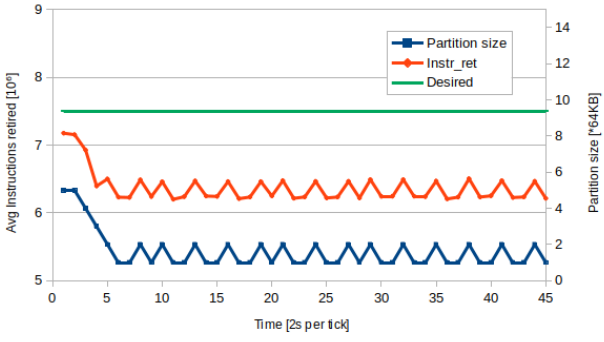

Figure 11.15: Matmult$_{ikj}$ low priority

The graphs show the re-distribution policy when Matmult$_{ijk}$ and Matmult$_{ikj}$ run with the same priority (low). The controller immediately assigns one cache partition to SUSAN, which increases SUSAN's QoS to above the 93% threshold. The controller then triggers the fairness calculation for both matrix multiplications. Matmult$_{ijk}$ has the most significant distance to the desired QoS and gets cache partitions from Matmult$_{ikj}$ for five controller iterations. The increased cache space results in an increased QoS for Matmult$_{ijk}$ but also a decreased QoS for Matmult$_{ikj}$. The controller starts to fluctuate at iteration 8, since Matmult$_{ikj}$ has now the furthest distance to its desired QoS. The controller thus assigns one cache partition from Matmult$_{ijk}$ to Matmult$_{ikj}$, a behavior maintained throughout the rest of the experiment execution.

11.6.7 Discussion

Our results show that it possible with relatively non-intrusive algorithms (0.3-0.7& CPU utilization) to control an application's QoS using only the means of cache re-partitioning. The first and most engaging discussion point is when to stop assigning cache partitions. All of our experiments, except for the all-different priority case, display a self-fluctuating state of the controller, which repeatedly decreases/increases the partition size of the same two applications. The fluctuating behavior is a consequence of a traditional constant controller that is working, but that case might not be practical. It is possible to add sanity checks within the controller that detects such behaviors since we have access to historical data and stop the re-partitioning procedure once detecting a fluctuating behavior. Stopping the re-partitioning procedure will increase the complexity of the controller significantly, since we then also need to add decision making for starting a stopped controller again.

The SUSAN's fluctuating performance - Fig. 11.10 - can be explained as follows. SUSAN trespasses the performance threshold setpoint already at controller iteration 1. SUSAN reaching the performance threshold mark this early is however an outlier and could be a result of SUSAN executing a couple of "lucky" executions. SUSAN goes below the threshold QoS setpoint again at controller iteration 4, wherefore the controller re-starts to assign partitions to SUSAN. Implementing a freezing functionality for the controller could, in this particular case, have led to a scenario where the controller freezes the partitions for SUSAN at partition size 1, while the current detected performance was a result due to a measurement anomaly.

11.7 Summary

We presented here the idea of building an online cache partitioning controller that focuses on maintaining QoS for prioritized applications. We presented two primary cases; maintaining QoS based on a user-defined reference value and maintaining QoS through prioritization. Our results show that it is possible to control the execution time of several cache-bound tasks in a multi-core system by adjusting cache partition sizes. We introduced two controller modes: *fair* and *prioritized* and execute experiments using three applications. Our fair partitioning algorithms display favoritism towards the matrix multiplications because the difference between their current QoS and their setpoint performance is always greater than SUSAN's. The two matrix multiplications also show a more significant sensitivity towards an increased cache space which results in SUSAN never receiving cache partition space according to the fair policy. We therefore implement a priority policy that will assign partitions for applications with higher priority on first-hand. We show that our priority scheme prioritizes the QoS of SUSAN and increases its performance by 5% compared to a fairly partitioned system.

11.7.1 Future work

Our controller uses a minimum cache partition size of one, but there is also the possibility of investigating cache partition sharing such that applications which are not important share the same cache partition. Sharing the same cache partitions will cause cache partition contention and reduce the QoS dramatically for the affected applications but will on the other hand free more cache partitions for the applications that do not share cache partitions. We also envision using more sophisticated controller techniques with other hardware that provides more available cache partitions. More available cache partitions means it can be possible to include a proportional element to the controller and change the cache redistribution to more than just one per iteration. Other interesting works include investigating effective ways to freeze the system and thus counter the self-fluctuating effect resulting from our controller operating and mechanisms that unfreezes the system once again at appropriate points.

Bibliography

[1] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang, and Y. Luo. Optimal cache partition-sharing. In *2015* 44^{th} *International Conference on Parallel Processing*, pages 749–758. IEEE, 2015.

[2] J. Danielsson, T. Seceleanu, M. Jägemar, M. Behnam, and M. Sjödin. Testing performance-isolation in multi-core systems. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 604–609. IEEE, 2019.

[3] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache pirating: Measuring the curse of the shared cache. In *2011 International Conference on Parallel Processing*, pages 165–175. IEEE, 2011.

[4] M. Jagemar, A. Ermedahl, S. Eldh, M. Behnam, and B. Lisper. Enforcing quality of service through hardware resource aware process scheduling. In *23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 329–336. IEEE, 2018.

[5] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14. IEEE, 2019.

[6] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *14th International Symposium on High Performance Computer Architecture*, pages 367–378. IEEE, 2008.

[7] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[8] S. Muralidhara, M. Kandemir, and P. Raghavan. Intra-application cache partitioning. In *International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[9] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *39th Annual International Symposium on Microarchitecture (MICRO'06)*, pages 423–432. IEEE, 2006.

[10] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 272–282. ACM, 2003.

[11] M. Xu, L. T. X. Phan, H.-Y. Choi, Y. Lin, H. Li, C. Lu, and I. Lee. Holistic resource allocation for multicore real-time systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 345–356. IEEE, 2019.

[12] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: a dynamic cache partitioning system using page coloring. In *Parallel Architecture and Compilation Techniques (PACT), 2014* 23^{rd} *International Conference on*, pages 381–392. IEEE, 2014.

[13] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 155–166. IEEE, 2014.

[14] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE* 19^{th}, pages 55–64. IEEE, 2013.

Chapter 12

Paper F
Run-Time Cache-Partition Controller for Multi-Core Systems

Jakob Danielsson, Marcus Jägemar, Moris Behnam, Tiberiu Seceleanu and Mikael Sjödin. In *45th Annual Conference of the IEEE Industrial Electronics Society (IECON)* IEEE, 2019.

Abstract

The current trend in automotive systems is to integrate more software applications into fewer ECU's to decrease the cost and increase efficiency. This means more applications share the same resources which in turn can cause congestion on resources such as such as caches. Shared resource congestion may cause problems for time critical applications due to unpredictable interference among applications. It is possible to reduce the effects of shared resource congestion using cache partitioning techniques, which assign dedicated cache lines to different applications. We propose a cache partition controller called LLC-PC that uses the Palloc page coloring framework to decrease the cache partition sizes for applications during run-time. LLC-PC creates cache partitioning directives for the Palloc tool by evaluating the performance gained from increasing the cache partition size. We have evaluated LLC-PC using 3 different applications, including the SIFT image processing algorithm which is commonly used for feature detection in vision systems. We show that LLC-PC is able to decrease the amount of cache size allocated to applications while maintaining their performance allowing more cache space to be allocated for other applications.

12.1 Introduction

Recent trends in the automotive industry show an increasing interest in high-performance computational machines. A common way to address the increased demand for computational capacity is the use of multi-core CPUs, which is a significant benefit to the autonomous industry due to the reduced size, weight, and power (SWaP) area [3]. Increasing the number of cores adds additional computational capacity, however, it also increases the system complexity. Multi-core systems are infamous for performance variations, which can become problematic in time-sensitive systems [8]. These variations often occur due to inter-core resource sharing, such as shared caches, shared memory bus, Translation Lookaside Buffers (TLB), shared DRAM-banks and others. These resources can be shared between cores, which means an application (e.g. app_0), executing on one core, does not have exclusive ownership of a single resource, instead it shares the resource with another application, (e.g. app_1), executing on an adjacent core. Such scenario can lead to shared resource contention where app_0 unexpectedly stalls, since app_1 has access to the resource.

The shared last level cache (LLC) has been a performance bottleneck in multi-core systems for a long time because of simultaneous accesses from multiple cores. In recent years, several studies have proposed methods aiming to mitigate LLC contention through isolation. Some examples are cache partitioning which partition the LLC so that accesses from one application do not affect the performance of another [11]. An additional technique is cache locking [12], that forces applications to use only certain cache lines. Another example is cache scheduling [6] that schedules applications to minimize conflicts in the cache memory. Isolating the cache memory can however be a costly process in terms of lost memory space and increased overhead.

We have devised a new way to optimize LLC partition allocation, during runtime. We implement a controller that continuously reads the instructions retired event from the Performance Monitoring Unit (PMU) [5] to estimate the application's performance. This paper focuses on the LLC, but the PMU supports a broad set of events [15], and our method can be applied to other shared resources as well - to be investigated in the future. The controller correlates the performance metrics and the cache partition size, and decides if an application needs more cache memory to achieve the desired performance or Quality of service (QoS). Our main contribution is:

- Propose a method to *automatically* select the minimum cache-size to be

allocated to an application for achieving a desired QoS.

The rest of the paper is structured as follows. We give background information in Section 12.2 and describe the LLC partition controller we have implemented in Section 12.3. An empirical study of the correlation coefficient and also a comparison study of our LLC partition controller versus statically assigned LLC partitions is described in Section 12.4. Section 12.5 describe work related to ours and we conclude the paper in Section 12.6.

12.2 Background

In the following, we discuss cache partitioning and it's relations to application performance.

12.2.1 Partitioning to avoid LLC contention

LLC contention occurs when multiple applications compete for the same cache lines. This can drastically degrade the execution time. Page-coloring, a.k.a cache coloring [13] or cache partitioning, is a way of disqualifying applications from using certain cache lines. LLC partitioning in Linux can be done by replacing the standard Buddy allocator [14], forcing applications to take a subset of the total number of cache lines. Forming LLC partitions is often done by assigning colors to an application. The colors are then used to control where data requests from the physical memory should be put in the cache, see Fig 12.1.

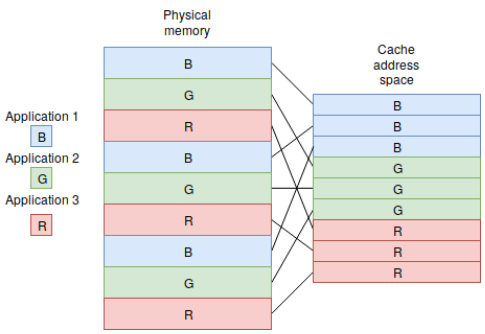

Figure 12.1: Cache coloring

The Figure shows three applications which split the cache memory equally.

The applications are assigned three different colors in the physical memory which are then used to map memory rows to cache line locations. Cache colors are referenced using the set-associative bits of the LLC, calculated according to Equation 12.1 [13].

$$Nr.\ of\ Colors = \frac{Cache\_size}{Cache\_ways * page\_size} \tag{12.1}$$

We have used the combined DRAM-bank partitioning and LLC coloring tool called Palloc [14] to create LLC partitions. Palloc is a kernel module which runs partitions at the granularity of a page and replaces the regular Linux Buddy allocator with a colored page approach.

12.2.2 Cache partitioning effect

Page coloring can be very efficient for reducing the execution time oscillations of applications executing in a memory contentious environment [13]. We have illustrated such environment in Fig. 12.2 where one 512x512 matrix multiplication application runs iteratively 100 times on core 0. The blue pluses show 100 iterations of the matrix multiplication without page coloring. The red crosses show 100 iterations of the matrix multiplication using palloc page coloring with a cache partition size of 60. Another matrix multiplication starts at iteration 20, running on core 1. The purpose of the newly inserted matrix multiplication is to cause LLC contention, which happens as a consequence of sharing the same LLC.

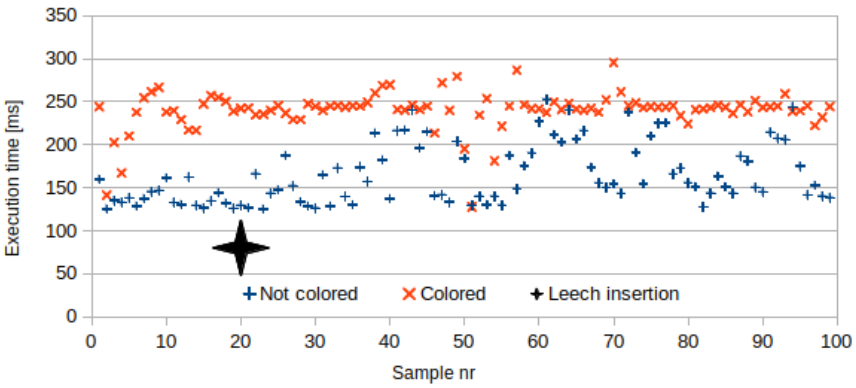

Figure 12.2: Matrix multiplication - isolation example

Fig. 12.2 depicts a typical LLC contention scenario, where the execution time

of the no-page-colored matrix multiplication starts to oscillate, after inserting the leech. The page-colored matrix multiplication is, on the other hand, undisturbed by the leech. It is, however, apparent that page coloring comes with an increased overhead due to extra latency in page allocations. Such trade-off can be worthwhile in time-critical systems when application time-predictability is essential. Overhead evaluations and Real-time performance impacts of the Palloc tool using bank partitions is extensively discussed in the Palloc paper.

Dimensioning the LLC partition sizes is one of the critical aspects when running multiple applications simultaneously. Assigning too small LLC partitions can significantly decrease the application performance. Fig. 12.3 shows the performance difference of the same matrix multiplication using various amount of LLC partition size.

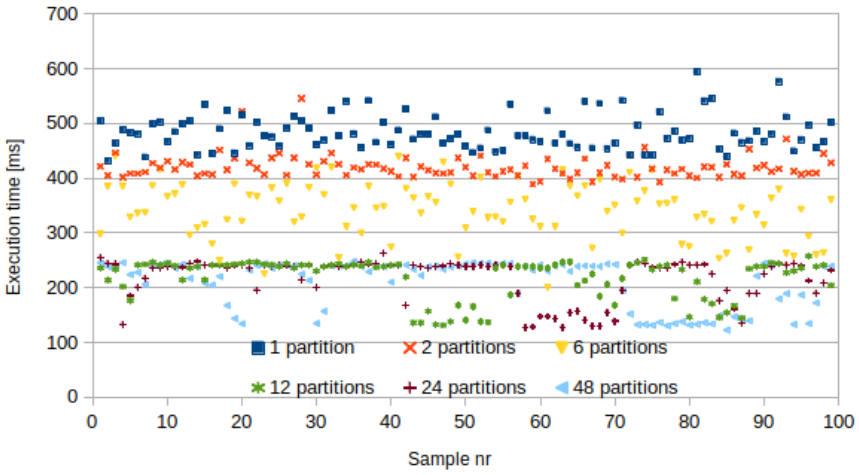

Figure 12.3: Matrix multiplication using different cache partition sizes

Assigning only 1 LLC partition to the matrix multiplication significantly reduces the performance, compared to the execution in Fig. 12.2, which uses 60 LLC partitions. Increasing the LLC partition size to 2, significantly increases the performance compared to the 1 LLC partition assignment and so on. Fig. 12.3 also illustrates an "above LLC saturation point" scenario - when an application does not gain performance from being assigned more cache memory, which is a consequence of fully saturating the temporal locality of the matrix multiplication. For this dataset size, the number of cache misses cannot be reduced anymore and all data which can be re-used is being re-used. Thus, there is no increase in performance from increasing the LLC partition

size further. In this case, the saturation point occurs at the 12 LLC partitions assignment. Further increasing the available LLC partitions, does not produce a significant performance impact on the application. Increasing the LLC size for this application will only allocate unnecessary resources. As a comparison, we could adopt a static partitioning strategy: for instance, assigning a 4^{th} of all cache partitions to each core in a 4 core system. In many cases, this may be a waste of valuable resources. Thus, we argue that it is beneficial to find the LLC saturation point at run-time, rather than statically assigning partitions.

12.3 Cache partition decision

There are many ways to create efficient LLC partitions. One possibility is to use exhaustive offline profiling for tasks, distributing the available cache partitions optimally to different tasks [2]. Offline profiling, however, needs complete knowledge of the applications running in the system. Changing the application set requires a complete re-profiling procedure before deploying new cache partitions. These limitations make offline cache partitioning not feasible for most dynamic systems. In addition, some applications may also change their respective workload during execution, which can be very difficult to foresee at design-time.

This paper focuses on LLC-bound workloads, meaning that the respective performance is bound tightly with the amount of LLC misses, where more LLC misses equals less performance. It is possible to assume that an LLC-bound workload will benefit from receiving more LLC partitions and opens up ways for constructing re-partition methodologies.

For an app_0, the performance is denoted by the number of retired (reached the final step in the instruction pipeline) instructions. In the context of the used example, our theory is that:

- The performance of an LLC-bound process is strongly correlated to the number of LLC misses.
- Enlarging the corresponding partition size available for app_0 increases the performance and decrease the LLC misses.
- The correlation between performance and increased LLC partition size decreases as the number of LLC partitions increase, until a *LLC saturation point*, where other resources (may) become the bottleneck

Based on our theory, we propose a correlation-based cache partition controller, *LLC-PC*, that tries to find the LLC saturation point - Fig. 12.4.

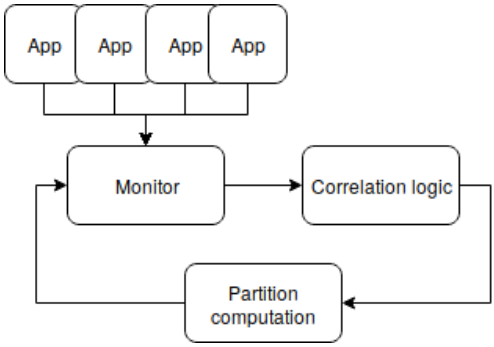

Figure 12.4: LLC-PC

The cache controller is a correlation based control loop which regulates the cache partition size according to the correlation between a performance metric and the increase in cache size for a specific application. The controller will continuously increase the cache size for as long as the correlation between the increase in amount of cache partition size and the performance metric is high. Once the correlation starts to decline and reaches a certain threshold, an LLC saturation point has been found and the controller will stop assigning additional cache partitions to the specific application.

The correlation scheme to find the LLC partition saturation point is based on the *Pearson correlation coefficient* [1] - a statistics methodology to quantify the relationship between two datasets. The pearson correlation coefficient is calculated according to Equation 12.2, where r is the pearson correlation coefficient estimate, n is the number of samples, x is the first sample vector, $\overline{x}$ is the mean of the first sample vector, y is the second sample vector, $\overline{y}$ is the mean of the second sample vector and i is the iterator.

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2(y_i - \overline{y})^2}} \tag{12.2}$$

The correlation coefficient ranges from values between -1 and 1. The absolute value of the correlation coefficient represents how strong the correlation is, where a higher value represents a stronger correlation. Correlation coefficients between 0.1 to 0.3 generally show a weak correlation, 0.4-0.5 show a medium correlation and greater than 0.5 show a strong correlation [4].

12.3.1 Controller implementation

We have implemented the LLC partition controller - *LLC-PC* - as a user-space application in the Linux operating system. LLC-PC employs the Palloc page-coloring interface, described in Fig. 12.5.

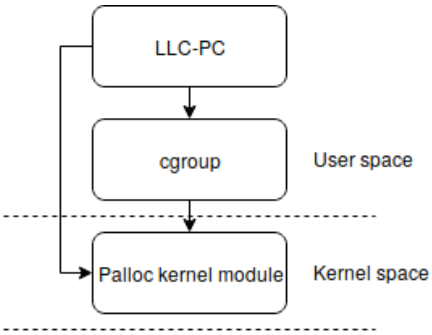

Figure 12.5: System connections

LLC-PC handles application connections through message queues and assigns LLC partitions to the connected applications using the `cgroup` interface. The `cgroup` interface has an implemented file-system called `palloc`, which uses the LLC set associative bits for configuring LLC boundaries. The `palloc` kernel implementation creates the cache colors based on the information provided by the `cgroup` file-system. LLC-PC has also a connection to the `palloc` kernel space user interface to enable `palloc`.

The controller, see Fig. 12.4, consists of three parts. The monitor part, the correlation part and the partition computation part. The controller implementation is described in Algorithm 2. The first forall block of the algorithm shows the connectivity part of LLC-PC, i.e., how the program deals with connected applications through message queues. Applications connect to LLC-PC by sending the application pid to a message queue. Applications furthermore notify LLC-PC of execution iteration ends by sending a "done" message to the same message queue. If the system does not currently recognize the pid posted by an application, the create_application function is triggered. This function initializes an application variable and stores the newly created application to an array. If an "end" message is received, an average value of instructions retired for the application is calculated, and the amount of average samples for the application is increased by 1.

```
Initialize_palloc();
Initialize_PAPI();
while forever do
    /* Handle application connections          */
    forall messages in message_queue do
        if message == new_application then
            initialize_application();
            tasks_in_system++;
        end
        if message == task_iteration_ended then
            calculate_avg_instructions_retired();
            avg_samples++;
            done = 1;
        end
    end
    /* Control loop segment                    */
    forall applications in tasks_in_system do
        /* Monitor application characteristics */
        instr_retired = read_pmu(pid);
        if avg_samples <= 3 then
            /* Calculate correlation           */
            correlation = pearson(avg_instructions_retired[i..end],
              cache_partition_size[i..end]);
            /* Make partition decision         */
            if correlation > 0.8 then
                partition_size++;
            end
        else
            /* Insufficient amount of data to
               calculate correlation           */
            partition_size++;
            done = 0;
        end
        resize_cache_partition();
    end
    sleep();
end
```

Algorithm 2: LLC-PC pseudocode

The second forall block shows the actual LLC-PC controller part and starts with an application monitor part. The monitor continuously reads the instructions retired PMU event for all application pids which exists within the applications array. The instructions retired event is stored within another array, used for calculating the average instructions. If the amount of average samples for an application is less than 3, a correlation calculation will not be performed, since it is not possible to detect trends with so few values. Thus, if there are less than three available average samples, LLC-PC will increase the partition size by 1. If on the other hand, the amount of average samples is at least 3, LLC-PC will start to perform the correlation calculation. The correlation calculation uses the average instructions retired and partition history for one application as input data and provides a Pearson correlation coefficient as output data. The application input data to the Pearson calculation is provided as a sliding window filter ranging from i to the end of the vector. This window is implemented to ensure that only the most recent values are accounted for in the Pearson calculation, to provide a faster response of LLC-PC. Once the correlation calculation is complete, a partition decision can be made. If the correlation is over 0.8, the partition size of the application is increased by one. If not, the saturation point of the application has been found, and LLC-PC will not increase the partition size further.

The third step is to actuate the resize cache partition method, which goes through all currently active applications in LLC-PC and calls cgroup/palloc to create partitions accordingly. Finally, the sleep variable dictates the periodicity of the monitor loop and therefore controls the number of values given as input to the average performance calculations — the average overhead of the LLC-PC monitor- and control loop averages at 73 μs. Decreasing the sleep timer will increase the amount of control-loop iterations per application samples and will thus increase the overhead while expanding the sleep timer will reduce overhead.

12.4 Experiments

We here describe the experiment on the identification of a feasible correlation threshold, to be used to determine the LLC saturation point. We evaluate how well LLC-PC perform compared to a static LLC partitioning.

Our experiment platform is a desktop Intel® Core™ i5 computer, with specification details as in Table 12.1.

Feature	Hardware Component
Processor	4xIntel® Core™ i5-8850H CPU (Skylake) 2.6GHz
L_1-cache	32 KB 8-way set assoc. instruction caches/core + 32 KB 8-way set assoc. data cache/core
L_2-cache	256 KB 4-way set assoc. cache/core
LLC	9 MB 12-way set assoc. shared cache
MMU	64 Byte line size, 64 Byte Prefetching, DTLB: 32 entries 2 MB/4 MB 4-way set assoc. + 64 entries 4 KB 4-way set assoc., ITLB: 128 entries 4 KB 4-way set assoc., L_2Unified-TLB: 1 MB 4-way set assoc., L_2Unified-TLB: 512 entries 4 KB/2 MB 4-way assoc.

Table 12.1: Hardware specifications Intel® Core™ i578850H

12.4.1 Point of saturation - Correlation threshold

Finding the right correlation threshold value is essential to LLC-PC, since a too low threshold value can cause the LLC-PC to act too slowly and therefore assign too many LLC partitions to an application. A too high threshold value may, on the other hand, force LLC-PC to act too quickly, and to assign not-enough LLC partitions to an application. The following experiments describe how the correlation coefficient between performance and LLC partition size changes over time, using different workloads while increasing the LLC partition size.

The correlation-based approach is able to identify which resource has the dominant effect on the performance of the applications, and this might change after allocating a certain amount of that particular resource, such as the LLC. Due to the space limitation, we will leave the management of multiple resources as future work and focus on a single resource which is the LLC.

Matrix multiplication. This experiment exemplifies what happens when a cache intensive workload runs on different partition sizes. We chose a 512x512 matrix multiplication, which is a well-known cache optimization problem [7] to run, using an increasing amount of LLC. Fig. 12.6 depicts the matrix multiplication instructions retired on the left-hand side y-axis and the correlation relationship between the instructions retired and the cache partition size on the right-hand side y-axis.

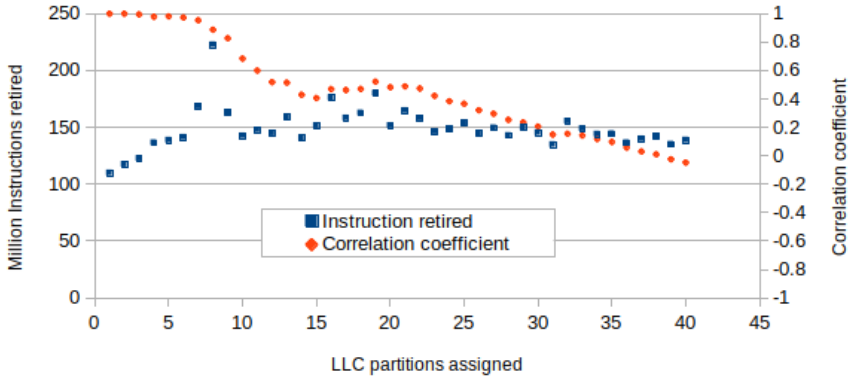

Figure 12.6: 512x512 matrix multiplication execution

The figure shows a gradually decreasing correlation curve and also a clear relationship between increased LLC partition size and instructions retired. The matrix multiplication reaches saturation at a partition size of 10.

SIFT. We test the SIFT algorithm, a commonly used feature detection algorithm to illustrate that our correlation theory works for not only synthetic workloads. Fig. 12.7 show as an execution of the SIFT algorithm run on a 4MB image with different cache partition sizes from 1 to 40.

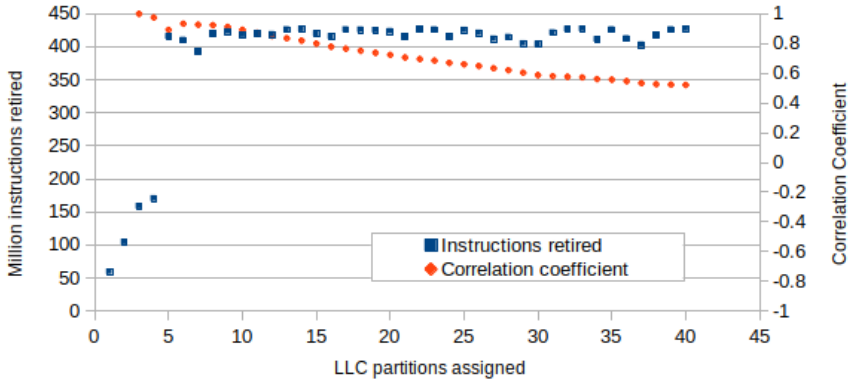

Figure 12.7: 4 MB SIFT execution

The figure shows an upwards going performance curve, with an absolute peak when assigned 37 cache colors. This peak is however very minor and can be explained as local deviation due to "lucky" executions. The majority of the peak values are, however, within the 405 million - 425 million instructions

retired interval, which is reached at a correlation coefficient of roughly 0.9 and continues to scale down.

Random Calculation. The purpose of this experiment is to exemplify what happens when a load is not LLC-bound. The random calculation program executes a set of random number requests and stores the random value into a variable. The variable is compared with another variable to find the highest value gained from the random number requests. We set the random number requests to 10^8 random number requests with a modulo of $5 * 10^5$ and increase the number of cache partitions assigned to this application by one each time the application is finished executing. Fig. 12.8 depicts the correlation coefficients from the random calculation test.

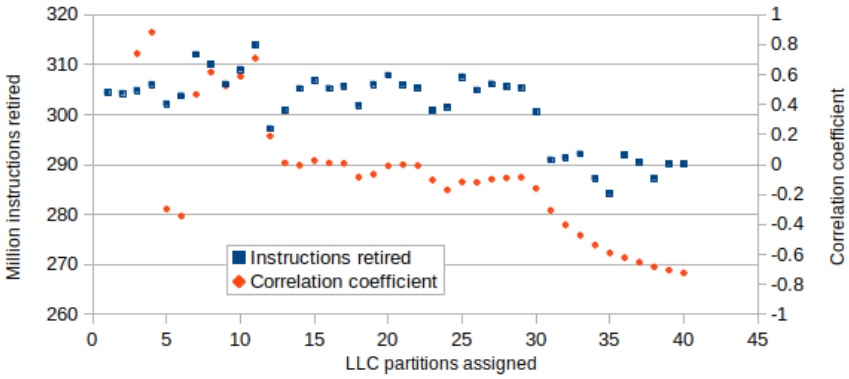

Figure 12.8: Random calculation execution

The figure shows an entirely different result from the matrix multiplication correlation graph. Instead of a continuously decreasing correlation, the correlation values are irregular at first but then saturates on iteration 13 to a correlation coefficient of 0.

12.4.2 Summary of experiments

There are two common nominators for the LLC-bound applications in these experiments. Firstly, the number of instructions retired increase when increasing the LLC partition size. The increase in instructions retired is reasonable since the application gets significantly more LLC. Secondly, there is a point where the instructions retired curve levels off to a stable state. The curve levels out when the application is assigned a certain number of LLC partitions.

Thus, we have found the LLC saturation point for this given application. We can conclude that in our experiments, the LLC saturation point of the curve is a certainty at a correlation coefficient of 0.8. Using this conclusion, we set the correlation threshold to 0.8 in the subsequent LLC-PC experiments, which is the point from which LLC-PC will not assign more cache partitions to an application. Using a correlation over the entire dataset at all time, however, makes LLC-PC slow to saturate. The saturation of the system can, however, be hastened through introducing a sliding window, which only tracks the most recent cache partition and instructions retired measurements. Using a sliding window means the system will only react to current execution trends, not considering the earliest stages of the system execution.

12.4.3 LLC-PC evaluation

One static way of assigning LLC partitions is to split all available LLC partitions equally between the cores. Our test environment has 4 cores and 128 available cache partitions, thus each core gets $128/4 = 32$ static LLC partitions as a first reference value. We also use 16 partitions per core as a second reference value. Below, we show an evaluation of static partitioning vs. LLC-PC, using different sizes of the previously introduced LLC-bound workloads. We ran each test a total of 5 times. LLC-PC runs the experiment setup listed in Table 12.2.

Property	**Value**
Available LLC partitions	128
Correlation window size	5
Correlation threshold	0.8
Control loop sleep	50ms

Table 12.2: LLC-PC specifics

For the sake of test simplicity, re-partition regulations are made once each application iteration, however, in theory a re-partition decision could be made each time a memory manager call is made. We execute each test sequentially for a more straightforward interpretation of the results. The control loop address each task individually, which means that it is possible for the controller to handle multiple tasks concurrently at the same time. It can also be argued that the control loop sleep time would be a coefficient of the execution time such that the sampling occurs only a certain amount of times every iteration, however since the execution time can be very hard to predict, we chose to go

for a statically set sleep timer. Such a solution, however, requires accurate execution time prediction of an application, which becomes very troublesome since the execution time of each application can change dramatically due to cache re partitioning and would possibly mean more overhead to LLC-PC. We chose 50 ms as control loop sleep in order to get at least 100 measurement values for the average calculation for all application variations.

Matmult and SIFT running under LLC-PC. We evaluate LLC-PC versus a static partition based solution which uses a LLC partition size of 16 and 32. Fig. 13.6 and Fig. 12.10 depicts the execution flow of a 756x756 matrix multiplication and a 8MB sift execution respectively, using LLC-PC. The left-hand side y-axis of the graphs plots the median instructions retired (i.e., performance) per 50 milliseconds of the application using LLC-PC (blue squares), 16 statically assigned cache partitions (orange cross) and 32 statically assigned cache partitions (yellow plus). The right hand-side axis show the correlation over time using LLC-PC. A higher value on the left-hand side axis means more instructions executed per 50 milliseconds and is, therefore, better than a low value. The x-axis shows the number of partitions used, where a lower value is preferred since more cache partitions can be given to other applications.

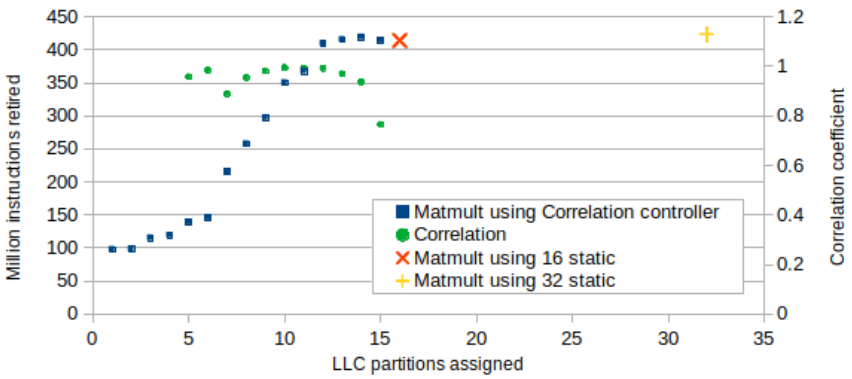

Figure 12.9: Comparison of 756x756 matrix multiplication executions

Fig. 13.6 shows a full LLC-PC run of a 756x756 matrix multiplication, where the system saturates at 16 partitions, with comparable performance to that of the static partitions. For this particular matrix multiplication size, the static partition size was equal to the correlated size. Statically increasing the LLC sizes to 32 does not improve the matrix multiplication performance significantly. Furthermore, Fig. 12.10 show SIFT operating within the LLC-PC, with a final assignment of 13 LLC partitions at which point the correlation value has

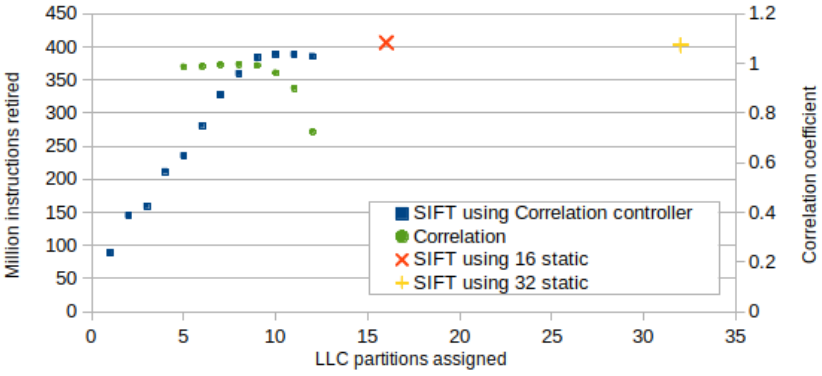

Figure 12.10: Comparison of 8MB SIFT executions.

dropped from 0.89 to 0.72. The correlation-based methodology almost reaches the same performance achieved by the static LLC partition allocations.

Table 12.3 and Table 12.4 further compares LLC-PC with a static partitioning strategy using different sizes of the workloads. W_{size} is the workload size and C_{size} is the LLC partition size assigned to the application, C_{instr}, S_{16} and S_{32} show the median million instructions retired per 50 milliseconds of the matrix multiplication using LLC-PC, 16 statically allocated LLC partitions and 32 statically allocated LLC partitions respectively.

W_{size}	C_{size}	C_{instr}	S_{16}	S_{32}
256x256	7	432.73	428.17	419.49
512x512	13	420.11	432.78	428.54
756x756	16	414.36	424.63	428.39

Table 12.3: Matrix multiplication tests

W_{size}	C_{size}	C_{instr}	S_{16}	S_{32}
1MB	6	395.38	406.56	408.79
2MB	7	384.96	413.01	405.85
4MB	9	387.80	410.61	405.87
8MB	13	385.53	406.31	402.58

Table 12.4: SIFT tests

Table 12.3 shows the benefit of LLC-PC, especially using the smallest matrix multiplication size of 256x256, which saturates at a partition size of 7. Increas-

ing LLC partition size to 16 and 32 does not increase the performance, and would thus be a wasteful LLC assignment since other applications could have used the LLC partitions. The larger 512x512 matrix multiplication size saturates at an LLC partition of 13, which is 3 LLC partitions less than the static 16 allocation, which does not notably change performance. Table 12.4 further compares LLC-PC with the static partitioned strategy using different image sizes, where W_{size} is the image size used by the SIFT application and C_{size} is the LLC partitions assigned to SIFT by LLC-PC. C_{instr}, S_{16} and S_{32} show the median million instructions retired per 50 milliseconds for using LLC-PC, 16 statically allocated LLC partitions and 32 statically allocated LLC partitions respectively. The table shows a close-to static performance for all different image sizes using less LLC partitions. The 8MB image receives 13 LLC partitions from LLC-PC and is which is relatively close to the S_{16} allocated partitions, which saves 3 LLC partitions from waste. Increasing the image size further could potentially trespass the S_{16} allocation using the correlation controller.

12.5 Related Work

Our work is based on the PALLOC [14] page coloring framework, which can be used for partitioning both the cache and DRAM banks. While the authors show that Palloc efficiently can be used to counter resource contention where all cores gain the same amount of cache partitions, they do not consider to optimize the cache assignments for each application. We aim to further extend this approach by using correlation-based partitioning decisions and therefore gain more efficient cache partitions. Ye et al. [13] presented the Coloris cache coloring engine which uses a threshold scheme, based on performance counters. The Coloris approach forms cache partitions based on how many cache misses one process contributes to the total amount of cache misses of all processes. Our approach differs from Coloris, as we look at how the performance of a process correlates to the cache misses of the same process. Perarnau et al. [10] presents another cache coloring scheme and argues that creating feasible cache memory partitions is best left to the user, since they have most knowledge of the application. We argue that it is difficult to know beforehand how much cache an application needs, in order to achieve a certain performance level. It is therefore beneficial to use a method that makes the cache partition decision automatically at run-time.

12.6 Conclusion

We have created a correlation based LLC partition controller, called LLC-PC, which can be used to find LLC partition sizes for workloads with unknown cache usage. We evaluate LLC-PC using two LLC heavy loads, a Matrix multiplication, and a SIFT feature detection algorithm. The results show that LLC-PC can be used for this set of workloads to reduce the amount of cache size given to an algorithm compared to a static 32 cache LLC partition assignment, and also in most cases a 16 LLC partition assignment - while still maintaining similar performance. We can probably find better cache partitions through thorough offline measurements and code analysis; however, our aim is not to find the absolute optimal cache partitions but rather find sufficient cache partition sizes during runtime of an algorithm.

Our prime focus has been to create a generalizable correlation model. We can apply the correlation model on any shared resource that has a performance counter event and a partitioning strategy which affect the shared resource, e.g., TLB partitioning [9]. Our future work includes introducing new partitioning strategies. We would also like to create a methodology for solving the multi-objective control problem when balancing multiple shared resources usage.

Bibliography

[1] J. Benesty, J. Chen, Y. Huang, and I. Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[2] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang, and Y. Luo. Optimal cache partition-sharing. In *2015* 44^{th} *International Conference on Parallel Processing*, pages 749–758. IEEE, 2015.

[3] A. Bucaioni, S. Mubeen, F. Ciccozzi, A. Cicchetti, and M. Sjödin. Technology-preserving transition from single-core to multi-core in modelling vehicular systems. In *European Conference on Modelling Foundations and Applications*, pages 285–299. Springer, 2017.

[4] J. Cohen. *Statistical power analysis for the behavioral sciences*. Routledge, 2013.

[5] T. Gleixner. Linux Performance Counter announcement, 2008.

[6] N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multicores. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 245–254. ACM, 2009.

[7] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 63–74. ACM, 1991.

[8] A. Mazouz, D. Barthou, et al. Study of variations of native program execution times on multi-core architectures. In *2010 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 919–924. IEEE, 2010.

[9] S. A. Panchamukhi and F. Mueller. Providing task isolation via tlb coloring. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 3–13. IEEE, 2015.

[10] S. Perarnau, M. Tchiboukdjian, and G. Huard. Controlling cache utilization of hpc applications. In *Proceedings of the international conference on Supercomputing*, pages 295–304. ACM, 2011.

[11] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[12] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 272–282. ACM, 2003.

[13] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: a dynamic cache partitioning system using page coloring. In *Parallel Architecture and Compilation Techniques (PACT), 2014* 23^{rd} *International Conference on*, pages 381–392. IEEE, 2014.

[14] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 155–166. IEEE, 2014.

[15] G. Zellweger, D. Lin, and T. Roscoe. So many performance events , so little time. *APSys '16*, 2016.

Chapter 13

Paper G
Modelling Application Cache Behavior using Regression Models

Jakob Danielsson, Janne Suuronen, Marcus Jägemar, Moris Behnam, Tiberiu Seceleanu and Mikael Sjödin. In *45th Annual Conference of the IEEE Industrial Electronics Society (IECON)* IEEE, 2019.

Abstract

In this paper, we describe the creation of resource usage forecasts for applications with unknown execution characteristics, by evaluating different regression processes, including autoregressive, multivariate adaptive regression splines, exponential smoothing, etc. We utilize Performance Monitor Units (PMU) and generate hardware resource usage models for the L_2-cache and the L_3-cache using nine different regression processes. The measurement strategy and regression process methodology are general and applicable to any given hardware resource when performance counters are available. We use three benchmark applications: the SIFT feature detection algorithm, a standard matrix multiplication, and a version of Bubblesort. Our evaluation shows that Multi Adaptive Regressive Spline (MARS) models generate the best resource usage forecasts among the considered models, followed by Single Exponential Splines (SES) and Triple Exponential Splines (TES).

13.1 Introduction

Cache memories in multi-core systems are prone to resource contention, most notably the last-level cache since it is commonly shared across multiple cores and allows for simultaneous usage [13]. The cache is a small, finite memory storage area and will evict data when its' capacity limit is met; the evictions are called cache misses. In contrast, references to a cache memory block are called cache accesses. Resource contention typically occurs when two memory-intensive applications execute on different cores, continuously executing cache accesses to new memory blocks. The cache memory will become full at some point during execution and, therefore, needs to evict cache lines to make space for new data requests. A vicious cycle can in the worst cases occur, where the applications' cache accesses continuously triggers cache evictions from each others' data, leading to performance degradation's and execution time fluctuations. One popular way to avoid such a scenario is to disqualify simultaneous usage of certain cache blocks through page coloring, also known as cache partitioning [30]. Page coloring removes resource contention through assigning specific cache blocks to specific processes at the cost of overhead performance penalties [9].

The execution characteristics of applications different depending on the application functionality. Applications are typically split into several phases [23], such as cache-heavy phases, arithmetic-heavy phases and floating-point heavy phases. A cache-heavy phase means the majority of the instructions leads to an access in the cache memory. In contrast, an arithmetic phase means the majority of the instructions causes an operation within the Arithmetic Logic Unit (ALU), etc. The most vicious scenario for cache contention is when two applications run simultaneously on different cores while executing their most cache heavy phases and stresses the cache to the capacity limit. We should not run applications that execute their most cache heavy phases simultaneously because of resource contention. Instead, we should schedule applications according to their shared resource usage, so the resource-specific phases (e.g., cache-heavy phases) never collides with each-other, thus mitigating the resource contention to a small degree. Making such a schedule requires modelling techniques that estimate the applications' resource usage trends for offline scheduling. To further adapt the methodology for reactive, run-time scheduling, we also need the model to predict the future resource usage.

Regression modelling is a mathematical process used to analyze trends in time-varying processes such as stock prices. In this paper, we benchmark different regression models with respect to the computing realm. We aim to create suit-

able regression models that can formalize an application's resource usage and create a prediction model for future resource usage.

We use the computer's performance counters to generate resource usage models. Our models look at what hardware is triggered by a software application and estimate its future resource usage. Performance counters are widely used in modern computers, making the modelling approach scale-able to all hardware that has the performance counter utility. In this paper we exemplify the modelling process using a set of three applications, including Bubblesort (non-cache heavy), matrix multiplication (cache heavy) to serve as synthetic workloads, to show resource forecasting applicability. We also use the Feature detection algorithm Scale Invariant Feature Transform (SIFT) [17] to serve as a realistic workload for resource forecasting.

Our contributions are:

- Resource forecast modelling on the three previously mentioned algorithms using the forecast models available in the Statsmodel module [22].
- A comparison evaluation on the accuracy of the different forecast models using the Root Mean Square Predicted Error (RMSPE) as a comparison metric.

The rest of this paper is organized as follows: Section 13.2 presents relevant notions, including measurement strategy, computer resource usage, and regressive performance analysis. Section 13.3 introduces our method for evaluating different regression processes. In section 13.4 we show the comparison results of the nine regression processes and discuss the results. Section 13.6 concludes the paper and presents opportunities for future work.

13.2 Background

It is challenging to predict bottlenecks for a particular hardware resource (such as L_1-cache, L_2-cache or similar) since the hardware resource usage may vary significantly during the execution time. For demonstrative purposes, we show the L_3-cache usage for the SIFT algorithm using an 8MB image executed on a single CPU, Figure 13.1. The y-axis plots the total number of L_3-cache accesses, while the x-axis shows the measurement points over the entire execution.

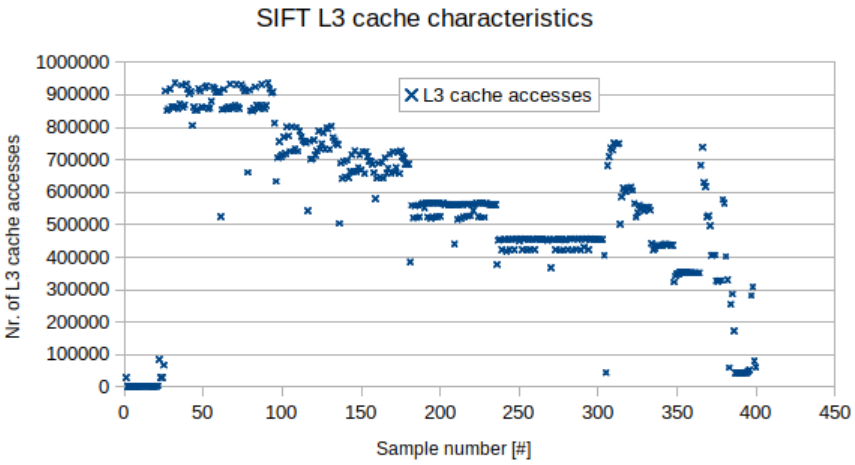

Figure 13.1: Illustration of SIFT using an 8MB image.

The L_3-cache usage of the SIFT algorithm varies by a significant amount over time. The L_3-cache usage is low at the start of the algorithm and rapidly increases after 0.5 seconds. The L_3-cache accesses count is significantly reduced at the 6-second mark.

The SIFT application takes 9.1 seconds to execute, which itself is not a very long time. Still, a complete software system often consists of 10's to 1000's of tasks, which could have similar execution time to that of SIFT. Accurately forecasting resource usage can significantly decrease run-time applications' testing time since they do not need to run for their full duration.

Forecasting and predicting the hardware resource usage also helps system designers in making three significant decisions, listed as follows:

- Hardware evaluation: The system designer will be able to distinguish sooner if a specific platform has enough capacity to run the software.
- System scheduling: Forecasting will enable scheduling the system runtime in a resource-efficient way so that hardware resources can be utilized at their maximum capacity without interference from other tasks.
- Resource bottlenecks: Accurate resource usage forecasting can also indicate if an application will be affected by resource capacity limits in the future casing resource bottleneck.

In the following sections, we discuss how different resources affect an application's performance and how to measure interesting resources using the Performance Monitor Unit (PMU). We also discuss different regressive models for forecasting application resource usage.

13.2.1 Computer resource usage

Computers consist of a vast set of resources, such as cache's, memory management unit (MMU), main memory (DRAM), I/O's, etc. These resources add functionality to the processor, such as memory access speed through temporary memory storage areas (cache's), increase instruction-level parallelism (processor pipeline), increase process parallelism (processor cores), etc. All applications utilize at least one computing resource during execution and are therefore dependent on this resource to complete their execution - we call this resource boundness [10]. However, many applications are often complex and thus bound to several resources simultaneously - a memory-bound application, for instance, typically utilizes the entire spectra of the memory chain: TLB's, cache's, DRAM, and instruction memory.

In this paper, we mainly investigate how to generate and forecast cache resource usage models. We limit ourselves to constructing resource usage forecasting models of cache for memory-bound applications, and, at the moment, we exclude other resource boundness situations.

13.2.2 Measurement strategy

Measuring the resource usage of applications can be done using the Performance Monitor Unit (PMU), which is included in most modern hardware. The PMU hosts a large set of counters - the Performance Monitor Counters (PMC), which are event-triggered hardware counters that trace the various resource usage within a computer. We use here the Performance API (PAPI) [18], a performance counter library that utilizes the built-in Linux *perf* headers [26] for measuring the performance counters. Further, we take a sampling-based approach to generate resource forecasts of applications. This means that we continuously measure the performance counters during runtime of an application with a certain frequency, instead of measuring the total count. Figure 13.2 depicts our measurement strategy.

In this way, we can generate resource usage forecasts on the individual application, since we have multiple sampling points of the cache resource usage.

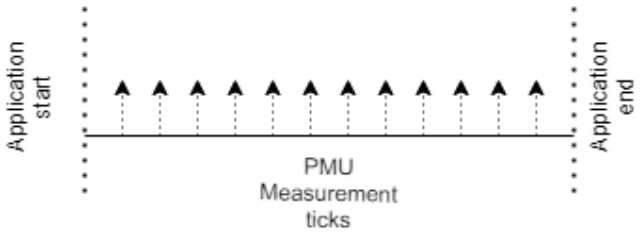

Figure 13.2: Periodic measurement of performance counters.

13.2.3 Regressive performance analysis

Regressive analysis is a method for modeling relationships between dependent and independent variables through a statistical process. Dependent variables are what a regressive model tries to predict or model. Independent variables are factors that have an impact on the dependent variables we are investigating. For example, a dependent variable can correspond to the execution time of any given process. A potential independent variable is the number of cache misses within that process, which may negatively impact that process's execution time.

There exists a broad spectrum of different regression tests, e.g., *autoregressive* and *spline modeling* processes [19]. Regressive modeling requires a dataset to construct models and approximate dependent variables by estimating independent variables' functions alongside an error term. The result of this procedure is an estimation model of the relationship between different variables of interest. The final product of estimating a mathematical function is the ability to forecast dependent variables.

We list here the major steps to construct a regressive model:

1. Select a modelling process suitable for the observed patterns in the data
2. Fit the parameters of a model to training data according to what is dictated by the modelling process.
3. Evaluate the prediction accuracy, for example by examining Root Mean Square Prediction Error (Equation 13.4), of the fitted model using a validation dataset.

There are two types of models that regressive modeling processes estimate:

- *Parametric models* consist of a finite number, specified before the model is generated, of parameters.
- *Non-parametric models* can theoretically include an infinite number of model parameters.

We limit this work only to include first-order parametric models. Which means the number of parameters (i.e., the amount of variables within the model) is limited to only one. The quality of the generated model is measured by evaluating how the polynomial function of the model “fits" the measured dataset. There exists three outcomes, *good* fits, *overfits* and *underfits*. Good fits accurately describe the shape of the dataset and can also detect the trends in the dataset. Therefore, they can forecast future values of the dataset. Overfits means the model has generated too many parameters and will be an almost exact representation of the dataset. Overfits also means that it is impossible to make any forecasts of the dataset since the model is an exact representation of the values in the current dataset. Underfit means the model has too few parameters to make any forecast prediction. Sharma et al. [24] exemplify over-, under- and good-fitting in Fig. 13.3:

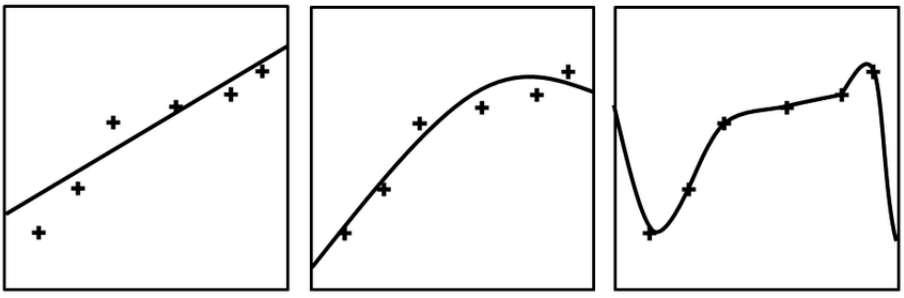

Figure 13.3: Example of under- (left), good- (middle) and over-fitting (right) [24]

Other than the problems of over-and under-fitting curves, adopting regressive analysis to construct forecasting models of hardware resource usage is a relatively straightforward process. Regressive models require datasets to be available at the moment of construction. Thus, we sample the cache resource usage of processes before initiating the model construction step and feed the dataset to our regressive models once the sampling is finished. Our regressive resource usage models are, therefore, offline representations of resource usage.

13.2.4 Related work

Existing research addresses methods that investigate how to create hardware resource usage models and how they can be applied to predict application performance. Several studies evaluate statistical regression models as suitable candidates for building performance forecasts [25, 14]. These works show how different performance values can be predicted using different regressive models. Other related work which is closest to our investigates resource usage through a novel autoregressive model called *Threshold Autoregressive* (TAR) [6]. The authors show that it is possible to create resource usage forecasts of any given application using regressive models, with a relatively low prediction error. Our research expands on this topic, and we evaluate the applicability of resource forecasting on different regressive models in the Statsmodel module [22] and *pyearth* for MARS models. Other results on predicting software performance and resource usage using autoregressive models are presented by Schneider et al. [8].

While some of the previously listed works employ similar regressive models to ours, we introduce the use of hardware performance counters' in conjunction with regressive models. To the best of our knowledge, no other research works utilize performance counters as a mean for forecasting and detecting hardware capacity bottlenecks.

Other relevant works investigate how to use PMU's to evaluate application performance models without in-depth knowledge of application code [27, 2, 7, 15, 12, 21]. However, these current methodologies strictly rely on measured data, which requires an application to be run until completion at least once to completion. Resource bottlenecks can only be discovered offline after application execution. Since our paper targets forecasting, our additions to this domain enable us to discover potential hardware resource bottlenecks before they occur.

13.3 Method

13.3.1 Model System Behaviour

We primarily focus on constructing forecast models of an application L_2-cache and L_3-cache usage. We start by gathering hardware resource usage samples during the application runtime to generate forecast models. We use a time slot sampling strategy which is similar to a frequency based measurement strategy.

One PMU measurement sample is taken at the end of each sampling timeslot. We determine the time-length of a time slot according to Equation 13.1, where T is the time slot length in a time unit, app_e is an applications' execution time and s is the desired number of samples.

$$T = \frac{app_e}{s} \tag{13.1}$$

Assuming an application execution time (app_e) of 1 second and the desired number of samples (S) is 100, the timeslot length (T) is equal to 10 milliseconds, which means a sampling rate of 100Hz. We denote the set of all measurement samples as y_c. Here, y denotes the application, and c denotes the performance counter. In this paper, we focus only on L_2-cache and L_3-cache accesses, thus, c will indicate either the L_2-cache or L_3-cache accesses. We furthermore denote the individual performance counter sample of an application y as t, we can have the complete execution characteristics of c, with 100 samples, given in Equation 13.2.

$$y_c = \{t_0, .., t_{100}\} \tag{13.2}$$

Next, we use y_c set to generate the forecast model $\hat{y_c}$, see Equation 13.3, where model represents a regressive model process.

$$\hat{y_c} = model(y_c) \tag{13.3}$$

The populations y_c and $\hat{y_c}$ describes the actual (y_c) and modelled ($\hat{y_c}$) cache resource usage at specific time points. We access data within the respective population using discrete time points t. $y_c(t)$ returns the actual cache resource usage at timepoint t and $\hat{y_c}(t)$ returns the modelled cache resource usage at timepoint t.

We exemplify a resource usage forecasting scenario using matrix multiplication as a test application and a MARS model as forecasting generation model, Figure 13.4. The first 50 samples in the model show a tendency of overfitting but still generates a close estimation of the measured L_2-cache usage in the last 100 samples. The y-axis shows the amount of L_2-cache accesses, and the x-axis shows the sample number. One blue dot corresponds to the data samples at a specific time point (t_i), which means all blue dots builds the population y_c where c is equal to L_2-cache accesses. The green line depicts the MARS model's fit, and the red line depicts the forecast model produced by MARS

($\hat{y_c}$). The figure visualizes the purpose of resource forecasting, i.e., the ability to forecast the L_2-cache usage.

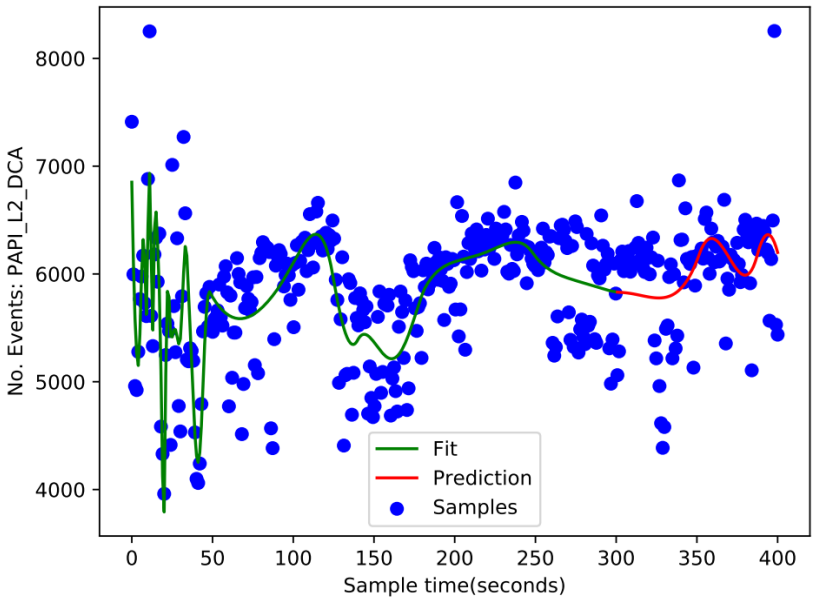

Figure 13.4: Example of an L2 cache usage function estimated using a MARS model [19].

The figure shows the model generation of MARS, where the red line depicts the actual forecast model. We then evaluate the model's applicability using RMSPE, see Section 13.3.2. In this work, we investigate the applicability of multiple different regression models, including Auto-regressive, Auto-regressive Moving Average, Auto-regressive Integrated Moving Average, and Spline (Natural, B- and MARS) Regressive modeling processes [19] and their applicability for forecasting applications' usage of L_2-cache and L_3-cache memory. We list all the modelling that we consider in Table 13.2.

13.3.2 Evaluation Methodology

We use the Root Mean Square Prediction Error (RMSPE) to evaluate our regressive models' accuracy. The RMSPE metrics describe the difference between predicted values and actual observation values of the data set. This paper uses RMSPE as a model comparison metric; the lower RMSPE value

means the difference between the actual data and the forecast model is smaller and is preferable over a high RMSPE value. Equation 13.4 describes the RMSPE calculation, which is the root square value of the difference between all values in population $\hat{y_c}$ and y_c divided by the number of samples n.

$$RMSPE = \sqrt{\frac{\hat{y_c} - y_c}{n}} \tag{13.4}$$

Equation 13.4 gives the prediction error by squaring the sum of the averaged difference between predicted ($\hat{y_c}$) and actual (y_c) values, where n is the number of samples considered. In this paper, we exclusively use 400 as the number of samples, and therefore, n will always be equal to 400 in our experiments.

13.4 Experiments

We generate resource usage forecast models and evaluate the RSMPE value using the platform in Table 13.1.

Feature	Hardware Component
Processor	4xIntel® Core™ i5-8250U CPU (Kabylake) 1.6GHz
L_1-cache	32 KB 8-way set assoc. instruction caches/core + 32 KB 8-way set assoc. data cache/core
L_2-cache	256 KB 4-way set assoc. cache/core
LLC	6 MB 12-way set assoc. shared cache

Table 13.1: Hardware specifications Intel® Core™ i5-8250U

In addition to the hardware setup, we set the desired number of samples (s) to 400 for all experiments. We use 400 for all applications since it provides the best trade-off between over- and under-fitting of the curves for our test applications. In the following subsections, we discuss the applications put under test, the software execution environment and also the different regressive models that we use.

13.4.1 Execution scenario

We use three different applications including, a traditional bubblesort of an array, a conventional matrix multiplication of two randomly generated matrices,

and finally, an application containing the SIFT algorithm [20] for detecting features within an image. We use a matrix multiplication and bubblesort due to the simplicity in following their execution characteristics. Furthermore, we use SIFT to display resource forecasting usage in a more realistic non-synthetic scenario. In the following subsections, we discuss the basic mechanics and the resource usage of our applications.

13.4.1.1 BUBBLESORT

The BUBBLESORT algorithm compares two adjacent values within an array, the left-hand side value, and the right-hand side value. If the right-hand side value is lower than the left-hand side value, these values swap location within the array. The bubble sort application's main mechanic utilises comparisons mainly, which means it is a heavily branch-predictor dependent application.

13.4.1.2 Matrix multiplication

We use a standard *ijk* matrix multiplication, famous for loading the cache in a very suboptimal way. Our matrix multiplication multiplies the columns of one matrix A with the row of matrix B. The result value is stored in matrix C. The procedure of loading values from a matrix and storing new values into another matrix is very memory intensive, which means its a memory-bound application, including caches and DRAM.

13.4.1.3 SIFT

SIFT is a complex feature detection algorithm containing several mathematical operations such as the difference of Gaussian, nearest neighbor, hough transform voting, linear least squares, and more. The mathematical operations mean the SIFT application performs multiple steps and may depend on several different resources during the algorithm's different phases.

13.4.2 Environment

We collect data using the platform specified in Table.13.1) running the 64-bit desktop version of Ubuntu 18.04 LTS in an unmodified state with Linux kernel version 5.3.0-46-generic. As a measure to lessen stalls due to user-related interface interaction, we disable the graphical interface. We reboot our test

platform for each test run to clear cache levels and ensure each test runs with a cold cache and comparable circumstances. Our experiments run on an as-is Intel® Core™ i5 8250U(Kaby Lake architecture) with four homogeneous cores clocked at a base frequency of 1.60 GHz and a three levelled cache hierarchy. We list all the details on our test platform in Table 13.1.

13.4.3 Execution

For each test application, we collect L_2-cache and L_3-cache resource usage data at a sampling rate specified in Equation 13.1. The resulting three datasets are each individually split according to a 75/25% ratio which yields data subsets of 300(75%) and 100(25%) measurements for model training and testing. The values of hyperparameters is a crucial factor to consider when fitting regressive models. Regarding Auto-Regressive models(AR, ARMA, ARIMA), the adjustable hyperparameters are the Auto-Regressive(AR) and Moving Average(MA) orders. In our case, all Auto-Regressive-based models are of the first order. The Exponential Smoothing models(SES, DES and TES) are tunable by modifying the weights applied to previous observations(α), trends(β) and seasonality(γ). We optimize α in SES and α, β, γ in TES using maximum log-likelihood. α and β in DES are set to 0.8 and 0.2 respectively as this procured better results compared to maximum log-likelihood optimization. Finally, the Spline-based models are tunable by the number of knots, Degree of Freedom(DF), and the maximum polynomial degree of each spline. Both B-spline and N-spline DFs are set to 10, whilst the maximum polynomial degree is set to five. In MARS, only the maximum polynomial degree is adjustable by design and is set to 3.

A key aspect concerning regressive models is their data demands to avoid over-and underfitting. Thus we set the sampling frequency to capture enough measurements without compromising the significance of the observed usage. Adopting a higher sampling rate could provide unrepresentative usage data since the overhead of measuring the performance counters becomes overwhelming compared to the actual measurement samples.

We run the SIFT algorithm on an 8MB image. BUBBLESORT sorts a 6MB array of randomly generated values. The MATMULT workload multiplies two matrices summing up to a workload size of 1MB. We sample the L_2-cache, and L_3-cache accesses during each application's execution. Figures 13.5, 13.6 and 13.7 plots the execution profiles for the SIFT, MATMULT and BUBBLESORT respectively. Orange dots mark the quantity of L2 accesses and blue crosses, which marks the number of L3 cache accesses over 400 measurements.

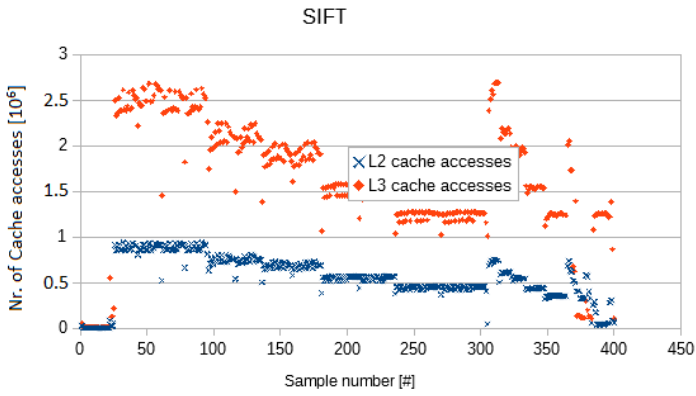

Figure 13.5: Memory usage illustration of SIFT using an 8MB image.

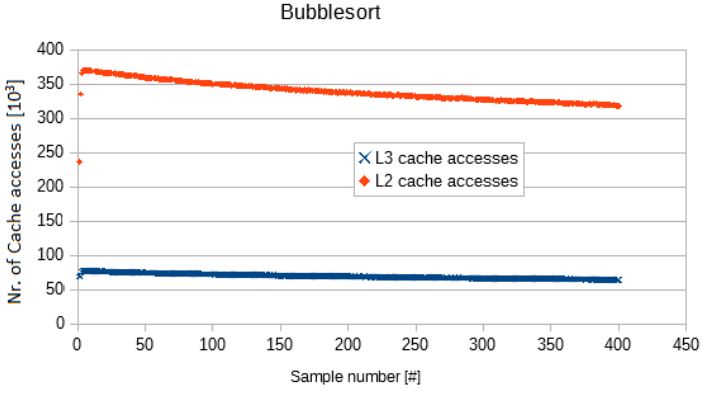

Figure 13.6: Memory usage illustration of a matrix multiplication using a 1MB dataset.

All three applications show very different execution profiles. The BUBBLESORT application shows a contiguous decrease in both L2 and L3 cache accesses. The matrix multiplication instead has a ramp phase, where the cache accesses rapidly increase in the beginning while saturating at measurement sample 80. SIFT shows stage-alike patterns in cache accesses, where there is first a dormant stage with almost no cache accesses. At sample 30, the cache accesses increase rapidly and remains high until sample 100, where the accesses starts to decrease gradually.

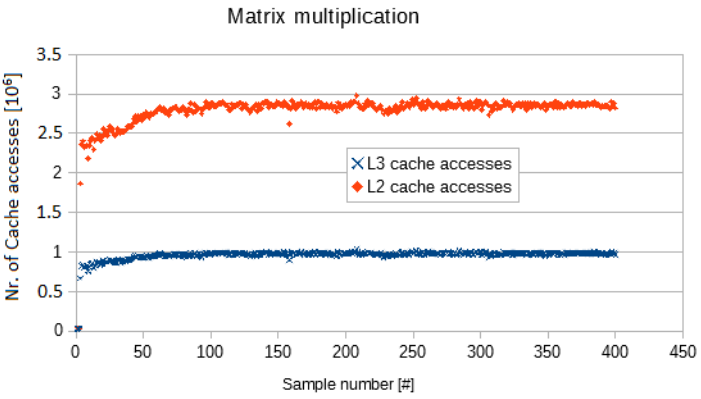

Figure 13.7: Illustration of SIFT using an 8MB image.

13.4.4 Model Comparison

Once the measurement phase finalizes, we create resource usage models using the regression processes listed in table 13.2 of each application, using the sample measurements.

Modelling process	Type
Auto Regressive (AR) [1]	Non-param
Auto Regressive Moving Average (ARMA) [28]	Non-param
Auto Regressive Integrated Moving Average (ARIMA) [3]	Non-param
Regressive B-spline [11]	Param
Regressive Natural Spline [5]	Param
Multivariate Adaptive Regressive Spline(MARS) [14]	Non-param
Simple Exponential Smoothing(SES) [4]	Non-param
Double Exponential Smoothing(DES) [16]	Non-param
Triple Exponential Smoothing(TES) [29]	Non-param

Table 13.2: Modelling processes evaluated in this work.

Our complete regression model suite generates a total of 54 forecasting models for our three example applications; 18 different models for each application, 9 different models for each cache level. For each of the 54 models, we calculate the RMSPE according to Equation 13.4. The RMSPE score describes how accurate forecasts made by a model are through calculating the the error size of the predictions. Thus, a lower RMSPE score is preferable over a higher one. Figures 13.8 and 13.9 shows the RMSPE scores for the SIFT application. Figures 13.10 and 13.11 the same for BUBBLESORT and Figures 13.12 and

13.13 the corresponding for MATMULT.

13.4.4.1 SIFT models

The first application we examine executes the SIFT on an 8MB image. The MARS models notably achieve the lowest RMSPE score for both cache levels. Figures 13.8 and 13.9 lists the RMSPE value of each different regression process on the left-hand side y-axis. Smaller RMSPE value means the prediction error is lower and is, therefore preferable to a high RMSPE value.

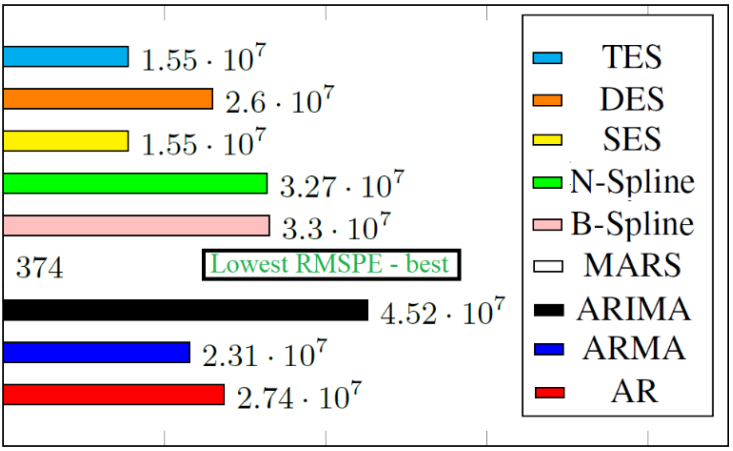

Figure 13.8: RMSPE score of L_2-cache usage models from data collected during execution of the SIFT algorithm with a 8MB image.

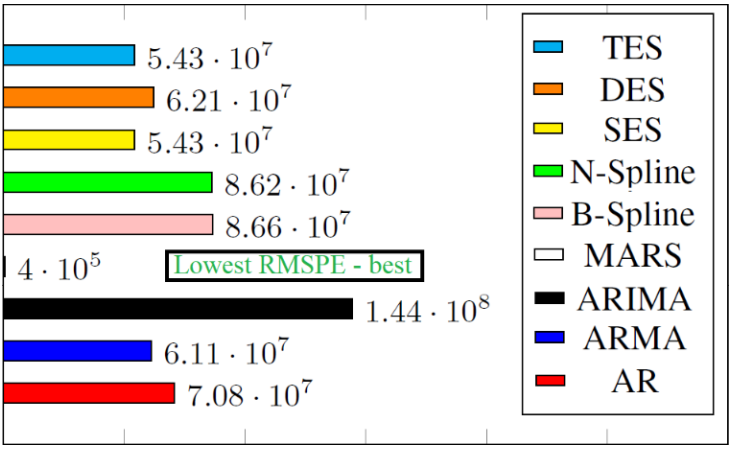

Figure 13.9: RMSPE score of L_3-cache usage models from data collected during execution of the SIFT algorithm with a 8MB image.

The SIFT workload identifies edge features in an 8MB image. On the other side of our SIFT RMSPE spectrum, the ARIMA models stand out with the highest RMSPE scores out of the calculated model scores. The remaining modeling processes achieve similar scores within the respective modeling process family. That is, B- and Natural splines models achieve similar RMSPE scores; the same applies to the Exponential Smoothing family of modeling processes (SES, DES and TES).

13.4.4.2 BUBBLESORT models

Our second application performs a traditional bubble sort on an unsorted integer array 6MB in size with random values. Figures 13.10 and 13.11 presents the RMSPE scores of each successfully constructed model.

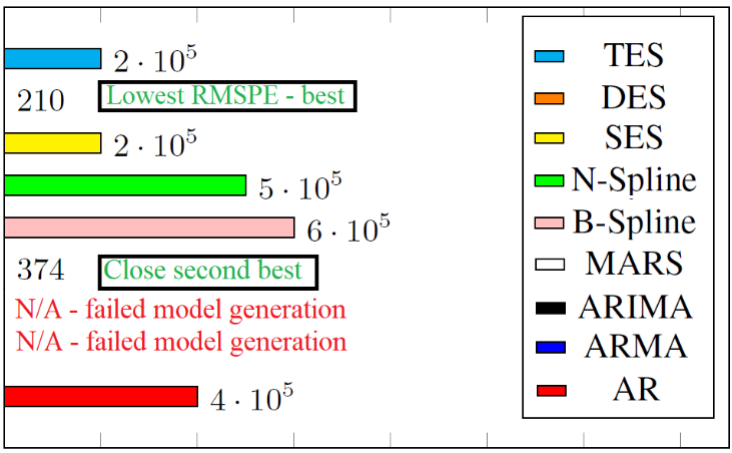

Figure 13.10: RMSPE score of L_2-cache usage models from data collected during execution of the BUBBLESORT algorithm with a 6MB array.

The BUBBLESORT RMSPE show MARS, SES, and TES outperform the other regression processes in prediction error. Furthermore, ARMA and ARIMA models failed to construct models from the dataset, due to lack of invertibility in the Moving Average(MA) component. We were, thus, unable to calculate RMSPE values for these modeling processes.

13.4.4.3 MATMULT models

Our third and final application performs a matrix multiplication between two square matrices with a working set size of 1MB for each matrix. Amongst

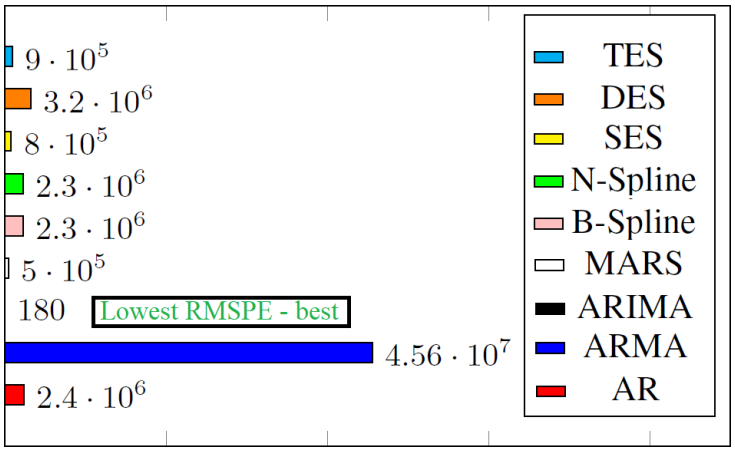

Figure 13.11: RMSPE score of L_3-cache usage models from data collected during execution of the BUBBLESORT algorithm with a 6MB array.

the L_2-cache models, the SES and TES models achieve, nearly identical, the lowest RMSPE scores, as seen in Figure 13.12. The B-spline L_2-cache model achieves the highest RMSPE score, and the ARIMA model fails to construct due to lacking data invertibility for the MA component. The RMSPE scores calculated for the L_3-cache usage models show a different outcome. Double Exponential Smoothing achieves a significantly higher RMSPE score, as visualized in Figure 13.13, while SES and TES models provide the smallest RMSPE scores.

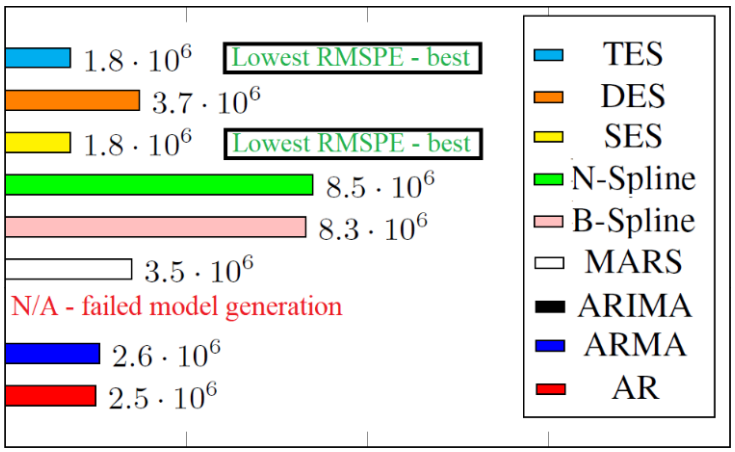

Figure 13.12: RMSPE score of L_2-cache usage models from data collected during execution of the MATMULT algorithm with a 1MB working set.

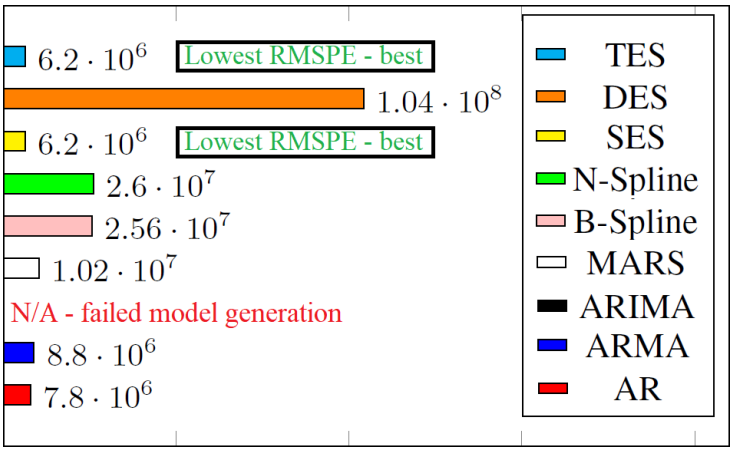

Figure 13.13: RMSPE score of L_3-cache usage models from data collected during execution of the MATMULT algorithm with a 1MB working set.

13.5 Dicussion of applicable methods

We examine several regressive modeling processes with the purpose of forecasting L_2-cache, and L_3-cache usage, expressed as the number accesses done per unit of time performed to a given cache level.

We examine both parametric and non-parametric modeling approaches. We state that parametric processes are too inflexible to model cache usage, as shown by the relatively high RMSPE scores of B- and Natural Spline models. Parametric models require set parameters before construction, requiring users to learn the resource usage pattern before setting optimal parameters. Setting parameters before-hand means additional testing time for finding the optimal parameters rather than actual testing, which contrasts our goal of reducing testing costs. As such, non-parametric models are preferable since they are more flexible compared to parametric alternatives.

Flexibility is a desired trait as cache usage patterns are not uniform across all applications. However, despite a higher degree of flexibility, non-parametric models are not fault-free since there is difficulty finding good fits when the measurement values are highly fluctuating [19]. Our work models three applications which do not have fluctuating resource usage characteristics to that extreme extent, and as such, we did not encounter the issue. A method of combating irregular usage patterns is adjusting the frequency to minimize the difference between each measurement. This solution is not without its prob-

lems, as some applications might have mixed and periodical resource boundness. Thus, measuring too frequently becomes an issue, as some applications might not finish a viable amount of work between each measurement.

In four out of six cases, MARS achieves the lowest RMSPE values of the non-parametric regression processes evaluated in this paper. TES and SES present a considerably lower RMSPE value, modeling the matrix multiplication application than MARS. These lower RMSPE values are visible in figures 13.12 and 13.13. The difference in low RMSPE values, depending on the different applications we use, suggests the best regression processes is dependent on the sample characteristics. Since MARS provides the best overall RMSPE value, it will be the best when forecasting resource usage of an application with completely unknown execution characteristics.

There is an inability to construct some of the regressive models with a moving average component, including ARMA and ARIMA, which happens as a consequence of wrongful tuning of the moving average component input parameter. Fine-tuning the input parameters of our models, however, means we need to add additional parameters into the model, which goes outside of the limitation of using only first-order models. As a final remark, referring back to this paper's original question: Can we predict a given application's resource usage using regression models? We argue that this is doable, as indicated by our results.

Since most CoTS hardware typically implements a broad set of performance counters, there are opportunities for complete resource usage forecasts. Our approach using frequency-based measurements on individual applications makes resource usage forecasts possible for any application as long as the hardware implements the performance counters, which are of interest.

13.6 Summary

In this paper, we evaluate methods for forecasting the resource usage of 3 different applications. We evaluate auto-regressive, spline regressive, and exponential smoothing as approaches for modeling applications and their usage of CPU L_2-cache L_3-cache. We compare the modeling fitness using RMSPE against each other to single out an ample modeling process. Our evaluation shows that MARS shows the most promise for forecasting application resource usage among the nine different evaluated regression processes.

13.6.1 Future work

We want to extend the MARS-work presented in this paper with resource scheduling forecasting using MARS processes making it possible to schedule processes in a cache-aware fashion. The processes do not interfere with each other, thus decreasing the risk for cache contention.

The three workloads we use in this paper are traditional Bubble Sort, Matrix Multiplication and Scale-invariant feature transform(SIFT). These commonly appear in larger applications and are thus representative for small chunks of code within a system. Future work includes examining our approach in conjunction with large-scale system solutions which include more complex applications with different resource usage profiles. Future work also includes conducting the resource forecasts in a scheduling environment where we test our hypothesis on resource contention. Ideally, two applications executing their most cache heavy phases should not run simultaneously since it builds a perfect environment for resource contention. Since we build resource usage profiles, the final goal is to create a new scheduling technique to mitigate cache contention through analysis of the cache access patterns.

The resource usage profiles will be different using different hardware and compiler flags. Our approach is agnostic since it only uses events produced from the performance counters. We would like to verify the RMPSE calculations against other hardware with different cache configurations.

Bibliography

[1] H. Akaike. Fitting autoregressive models for prediction. *Annals of the institute of Statistical Mathematics*, 21(1):243–247, 1969.

[2] R. Azimi, D. K. Tam, L. Soares, and M. Stumm. Enhancing operating system support for multicore processors by using hardware performance monitoring. *SIGOPS Oper. Syst. Rev.*, 43(2):56–65, Apr. 2009.

[3] G. E. Box and G. M. Jenkins. Time series analysis: Forecasting and control san francisco. *Calif: Holden-Day*, 1976.

[4] R. G. Brown. Exponential smoothing for predicting demand. In *Operations Research*, volume 5, pages 145–145, 1957.

[5] J. Cao, M. Valois, and M. S. Goldberg. An s-plus function to calculate relative risks and adjusted means for regression models using natural splines. *Computer methods and programs in biomedicine*, 84(1):58–62, 2006.

[6] X. Chen, Q. Quan, Y. Jia, and K. Cai. A threshold autoregressive model for software aging. In *2006 Second IEEE International Symposium on Service-Oriented System Engineering (SOSE'06)*, pages 34–40, 2006.

[7] Y. Cho, Y. Kim, S. Park, and N. Chang. System-level power estimation using an on-chip bus performance monitoring unit. In *2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 149–154, Nov 2008.

[8] M. Courtois and M. Woodside. Using regression splines for software performance analysis. In *Proceedings of the 2nd International Workshop on Software and Performance*, WOSP '00, page 105–114, New York, NY, USA, 2000. Association for Computing Machinery.

[9] J. Danielsson, J. Marcus, T. Seceleanu, M. Behnam, and M. Sjödin. Runtime cache-partition controller for multi-core systems. In *In 45th Annual Conference of the IEEE Industrial Electronics Society (IECON), 2019*, 2019.

[10] J. Danielsson, T. Seceleanu, M. Jägemar, M. Behnam, and M. Sjödin. Testing performance-isolation in multi-core systems. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 604–609. IEEE, 2019.

[11] C. De Boor. On calculating with b-splines. *Journal of Approximation theory*, 6(1):50–62, 1972.

[12] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using papi for hardware performance monitoring on linux systems. 08 2009.

[13] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache pirating: Measuring the curse of the shared cache. In *2011 International Conference on Parallel Processing*, pages 165–175. IEEE, 2011.

[14] J. H. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1):1–67, 1991.

[15] M. Hatzimihail, M. Psarakis, D. Gizopoulos, and A. Paschalis. A methodology for detecting performance faults in microprocessors via performance monitoring hardware. In *2007 IEEE International Test Conference*, pages 1–10, Oct 2007.

[16] C. C. Holt. Forecasting seasonals and trends by exponentially weighted moving averages. *International journal of forecasting*, 20(1):5–10, 2004.

[17] G. Lowe. Sift-the scale invariant feature transform. *Int. J*, 2:91–110, 2004.

[18] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[19] omitted for blind review. Master's thesis.

[20] Robertwgh. Ezsift. accessed: 2020-10-12.

[21] F. T. Schneider, M. Payer, and T. R. Gross. Online optimizations driven by hardware performance monitoring. *SIGPLAN Not.*, 42(6):373–382, June 2007.

[22] S. Seabold and J. Perktold. statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*, 2010.

[23] A. Sembrant, D. Eklov, and E. Hagersten. Efficient software-based online phase classification. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 104–115. IEEE, 2011.

[24] R. Sharma, A. Nori, and A. Aiken. Bias-variance tradeoffs in program analysis. volume 49, pages 127–137, 01 2014.

[25] S. Shimizu, R. Rangaswami, H. A. Duran-Limon, and M. Corona-Perez. Platform-independent modeling and prediction of application resource usage characteristics. *Journal of Systems and Software*, 82(12):2117 – 2127, 2009.

[26] L. Torvalds. Perf tools. accessed: 2020-07-07.

[27] S. Vogl and C. Eckert. Using hardware performance events for instruction-level monitoring on the x86 architecture. 01 2020.

[28] P. Whittle. *Hypothesis testing in time series analysis*, volume 4. Almqvist & Wiksells boktr., 1951.

[29] P. R. Winters. Forecasting sales by exponentially weighted moving averages. *Management science*, 6(3):324–342, 1960.

[30] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: a dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392. IEEE, 2014.