Characterization of Shared Resource Contention in Multi-core Systems

Jakob Danielsson

2019

Till minne av morfar

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 ownership of a shared resource. We investigate performance variations of parallel tasks in multi-core systems and present a method to pinpoint the source of the resource contention using existing hardware performance counters. Furthermore, we investigate methods to mitigate performance variations using resource isolation techniques. We present a methodology for verifying isolation and tested the achieved isolation using the Jailhouse hypervisor. We further investigate shared cache memory isolation techniques using a page coloring tool called PALLOC. Page-coloring is used for partitioning the cache, assigning specific cache lines to specific processes. Page coloring can however cause system performance degradation since it decreases the total amount of cache memory available for each process. Finally, we propose a dynamic partitioning assignment policy which assigns cache partitions to a process according to an adaptive model based on the process performance. The general conclusion from our investigations is that a large body of applications can suffer from shared resource contention and that techniques for mitigating resource contention are in dire need. Our methods measure and characterise applications, identifies resource contention and finally study isolation techniques.

Sammanfattning

Delad resursanvänding kan orsaka stora problem i dagens samhälle. Ett bra exempel skulle kunna vara en arbetande programmerare som åker bussen till jobbet varje morgon. Den delade resursen är i detta fall bussen som används av fler personer än bara programmeraren. Programmeraren skall vara på jobbet klockan 09:00 och varje bussresa tar cirka 25 minuter. Programmeraren har räknat att bussen som går klockan 08:30 ger tillräcklig marginal för att hinna fram till jobbet i tid. Ett problem som kan uppstå är om bussen är full när den når fram till programmerarens hållplats. Om bussen skulle vara full kan inte programmeraren gå på bussen och kommer då att komma för sent till jobbet. Detta problem uppstår på grund av att bussen delas av flera personer samt att bussen har en maxgräns på hur många personer som kan använda åka. Detta fenomen kallas vetenskapligt för *konkurens om delade resurser* och är vanligt förekommande även i datorer, speciellt i multi-core datorer där flera kärnor körs samtidigt och därmed konkurerar om samma delade resurser. Konkurensen om de delade resuserna har gjort multi-core datorer ökänt svåra att inom använda tidskritiska system på grund av exekveringstidsvariationer som uppstår, just på grund av just dessa konflikter. I denna avhandling undersöker vi problemet med konkurens om delade resurser i multi-core datorer genom att implementera program som medvetet framkallar konflikter. Vi har undersökt prestandavariationer i parallella applikationer som körs på multi-core datorer genom att mäta prestanda samt performance counters. Vi presenterar en metod som kan bestämma ursprunget av dessa prestandavariationer samt också identifiera konflikter i delade Cache minnen, CPU, minnesbus och lokala cachar. Vi har även undersökt metoder som helt eller delvis tar bort konflikten om delade resurser, s.k., isolationstekniker. Dessa isolationstekniker innefattar virtualiseringsverktyget Jailhouse samt cache partitioneringsverktyget PALLOC. Vi använder våran metod för att undersöka till vilken grad Jailhouse och PALLOC isolerar en specifik delad resurs. Fortsättningsvis undersöker vi även prestandaförluster och andra begränsande faktorer så som användningsområden när isolationstekniker används. Slutligen så föreslår vi en Pearson-

korrelationsbaserad policy som automatiskt anpassar cache partitionsstorleken till applikationer för att minimera slöseri av cacheresuser. Den slutliga sammanfattningen av våra undersökningar är att en stor mängd applikationer kan lida prestandaförluster som konsekvens av tävling om delade resurser. Våra metoder karakteriserar applikationer, identifierar konflikter om resurser och undersöker lämpliga isolationstekniker.

Acknowledgments

I would like to express my gratitude towards my supervisors and co-authors, Michael Sjödin, Moris Behnam, Tiberiu Seceleanu and Marcus Jägemar for their patience, help and valuable discussions throughtout this thesis. The work presented in this thesis has been funded by Mälardalens Högskola 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 taking his time and reading my thesis.

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 all these years and for all 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 through thick and thin during my master years and PhD student years.

Jakob Danielsson
Västerås, 2019

List of Publications

Papers included in thesis[1][2]

This paper is a collection of papers which include following publications:

Paper A: 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 22^{nd} Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2017.

Paper B: J. Danielsson, M. Jägemar, T. Seceleanu, M. Behnam and M. 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 C: J. Danielsson, T. Seceleanu, M. Jägemar, M. Behnam and M. Sjödin. Testing Performance-Isolation in Multi-Core Systems. In *43^{rd} Computer Society Signature Conference on Computers, Software and Applications (COMPSAC)*. IEEE, 2019.

Paper D: J. Danielsson, M. Jägemar, T. Seceleanu, M. Behnam and M. Sjödin. Run-time Cache-Partition Controller for Multi-core Systems In *45^{th} Annual Conference of the IEEE Industrial Electronics Society (IECON)* IEEE, 2019.

[1] A licentiate degree is a Swedish graduate degree halfway between M.Sc and Ph.D.

[2] The included papers have been reformatted to comply with the thesis layout, appendices have been moved to the end of the thesis, and some appendices added with respect to the papers as they were published.

Papers not included in thesis

Paper X: Jakob Danielsson, Mohammad Ashjaei, Moris Behnam, Thomas Sörensen, Mikael Sjödin, Thomas Nolte Performance Evaluation of Network Convergence Time Measurement Techniques. *22^{nd} Emerging Technologies and Factory Automation (ETFA)*, IEEE, 2017 [8].

Paper Y: J. Danielsson, N. Tsog and A. Kunnappilly A Systematic Mapping Study on Real-time Cloud Services *1^{st} workshop Quality Assurance in the Context of Cloud Computing (QA3C)*, IEEE, 2018 [9].

Contents

Part I

Thesis

Chapter 1

Introduction

The last few decades, chip manufacturers has established multi-core as the de-facto standard for CPU-chips. The major drivers for this trend is manly practical limitations in clock-frequency that favors architectures that allow parallel execution over singe-core and single-threaded execution, and the fact that mult-cores can performs more computations given a limited energy supply compared to single-cores. However, for timing critical and dependable systems, the use of multi-cores is not without problems due to *shared resource contention* - a state where multiple cores are requesting usage of one shared resource simultaneously. In this thesis, we specifically study bottlenecks and sources of unpredictable execution-times that occur in multi-cores due to shared resource contention. Figure 1.1 exemplifies a typical Commercial off the Shelf (CoTS) multi-core processor and contains several shared resources such as L_3-cache, I/O's, Memory bus and DRAM.

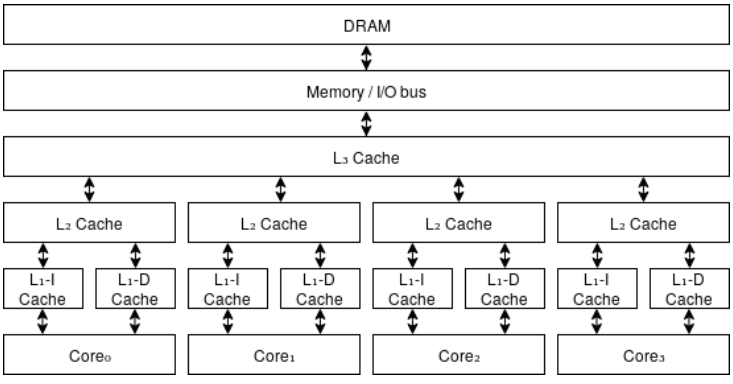


Figure 1.1: Example of a typical Intel® 4-core system

Intel® has, since the release of the first Core™ processors (Core™ solo and Core™ duo), followed the trend to add more cores to a chip. The latest Intel® coffee-lake releases have increased the number of cores to eight in total in the most powerful 9th generation i9 machines. The trend is similar for the powerful Intel® Xeon processors, which show an increase of available cores by a factor of three in just 6 years, see Figure 1.2.

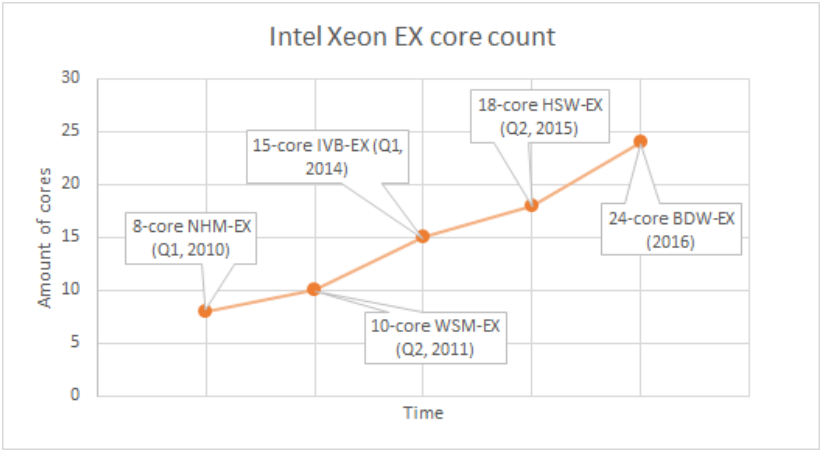


Figure 1.2: Intel® Xeon core development between 2010 and 2016

The increased number of cores in a processor means it is possible to utilize more parallelism but also present more problems due to simultaneous usage of shared hardware resources. Computers often follow a similar memory and I/O structure regardless of the number of cores. A processor has at least one core that is responsible for performing calculations. The core connects to a level one data cache (L_1D-cache) and a level one instruction cache (L_1I-cache). The caches are small but they connect directly to the core, which makes them very fast. The L_1D-cache and L_1I-cache are most commonly local and not shared across different cores. There is a special case when the local caches are still shared, due to the hyperthreading technique. The hyperthreading technique exists on high performance Intel®-processors such as the Intel® Core™ i7 and i9 versions where two threads to use the same physical resources simultaneously. This means one core can run two threads simultaneously, which in turn also share the local caches [23]. Modern Intel® processors use a second local cache, the level two cache (L_2-cache), which is a unified cache for both instructions and data, to increase the performance of a system. The L_2-cache is more sizeable than the L_1D-cache but is located further away from the processor, which results in higher access latencies. Recent Intel® Core™ processors also

add a third cache level (L_3-cache). The L_3-cache is commonly shared across all cores on the processor and is also known as the Last Level Cache (LLC). The LLC finally connects to the memory bus and is the last stop to fetch data from before the DRAM memory.

We aim to improve the knowledge of the resource contention in multi-core systems by investigating methods to verify the existence of resource contention in a shared resource. We also examine isolation techniques which can be applied to reduce the effects of resource contention.

1.1 Scope of the thesis

LLC is an encompassing term for all caches that are located last in the cache memory hierarchy and connects directly to the memory bus. The LLC does not necessarily have to be implemented as L_3-cache but can also be implemented as L_2-cache - which is common in ARM architectures, or even implemented as a L_4-cache (which implies a 4-level cache hiearchy) - which is common in powerful server processors.

Our research mainly targets shared resource contention, which happens as a consequence 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. The first part describes the shared resource contention problems and research results in the isolation of shared resources topic. The second part contains the included papers in the thesis, from Chapters 7 to 10.

Part 1 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 through isolation of shared resources. Chapter 3 lists the research challenges, formalizes the challenges into research questions, and describe the methodology we have used to solve these challenges. Chapter 3 describes the research contributions made in this thesis, pairs the research contributions with the research questions, and explains how the different papers are related to the research contributions. Chapter 4 provides relevant related work and Chapter 5 provides a summary of the papers included in this thesis. Chapter 6 finalizes the first part of the thesis with discussions, conclusions and directions for future work.

Chapter 2

Background

2.1 Internal memory subsystem of a computer

To understand the topic of internal memory contention, it is essential to understand how data requests travel through the memory hierarchy. A data request is always implemented with a load instruction in the CPU and must (if “unlucky") travel through the entire internal memory hierarchy, ending at the DRAM memory in search for the data. The internal memory subsystem of a processor is very complex and usually consists of several layers of caches and translation lookaside buffers (TLB’s). The different memory units can be visualized using a pyramid, where the smallest memory (the register memory) is placed at the top and the largest memory (the hard drive) is placed at the bottom.

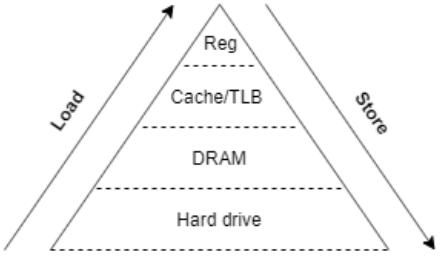


Figure 2.1: Model on the memory hierarchy

The chain of a computation always starts in the processor, where an operation is computed using one or more registers. The CPU holds a relatively reduced 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. Our thesis focuses mainly on the internal memory subsystem and we therefore use memory load and store operations for exemplifying the data pathing through the memory hierarchy of a computer. Figure 2.2 exemplifies the memory hierarchy at a high level. The hard drive is most spacious, the DRAM the second most spacious, caches are second least spacious and the register memory is least spacious.

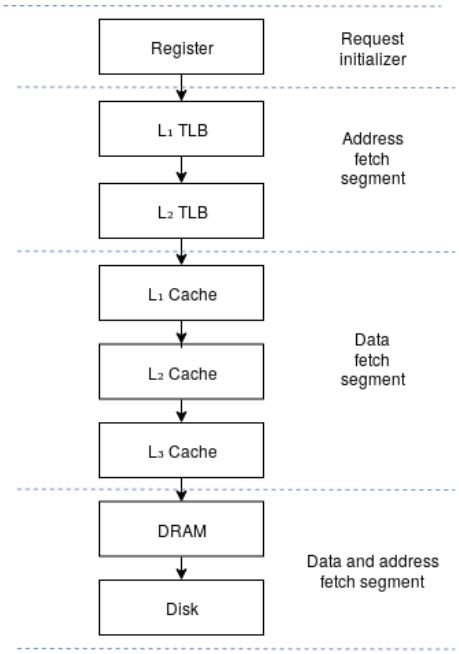


Figure 2.2: The entire memory call chain

Each memory operation contains a lookup procedure which checks if the requested data segment is available in the respective memory. 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), is divided into *words*, which is a unit of data that can be addressed. Each word is addressed by a 32-bit or 64-bit address depending on the computer architecture. The program address space of modern computers is often significantly larger than the actual DRAM address space. Modern computers implement *virtual memory*, which is a technique that uses the harddisk as a secondary memory address storing space. Using virtual memory thus reduces the possibility of system crashes due to insufficient physical memory. The virtual address space is significantly larger than the physical address space - a 64-bit system has a total of 2^{63} available virtual memory, while most physical memories in regular computers only host 2^{34} to 2^{35} available 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.

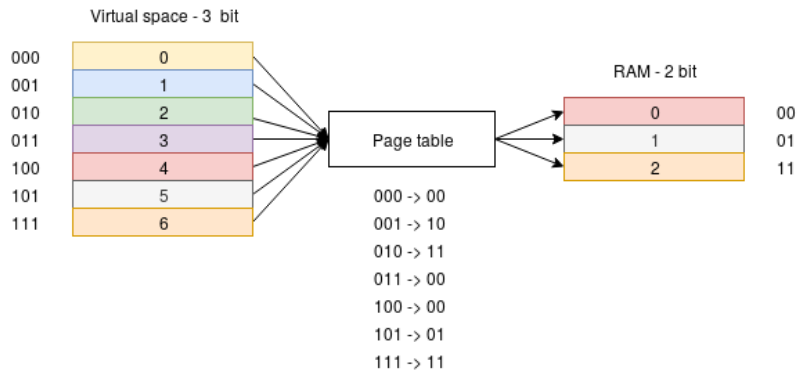


Figure 2.3: Illustration on page table mapping

The memory is divided into pages, typically 4 KB for most architectures, but the size may be configurable depending on the system requirement. It is also possible to use huge pages which typically resides in the megabyte or gigabyte range. 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, and the address has to be brought from disk. Page misses require new page table entries to be made for the requested address and causes substantial latency.

2.1.2 Translation lookaside buffer

The page table is relatively large, which means it is also relatively slow. Here is the point where the Translation Lookaside buffer (TLB) comes into play. The TLB is a small hardware-integrated buffer for storing page table entries. The TLB often contains entries within the double-digit range such as 32 or 64 entries. The TLB is thus significantly smaller than the page table, which contains millions to billions of entries.

Since the TLB is significantly smaller than the page table, it provides significantly faster page accesses. The TLB is essentially another cache used for address translations instead of data. It TLB is often further divided into 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.

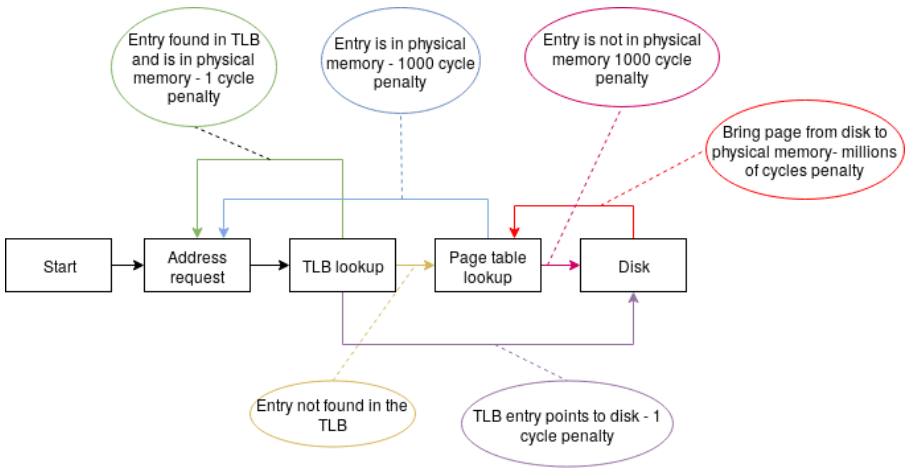


Figure 2.4: Virtual memory lookup procedure

2.1.3 Registers

The register memory is the closest memory to the processor and it is very small - usually, it consists of only a few registers. Registers 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, on the other hand, used to perform operations, also called instructions. Instructions include arithmetic-, logical-, comparison-, and memory operations. When executed, an instruction splits into smaller parts and sent to the

instruction pipeline. The smaller instruction parts are called *stages* and can be executed simultaneously in the pipeline. Figure 2.5 exemplifies a classic RISC pipeline.

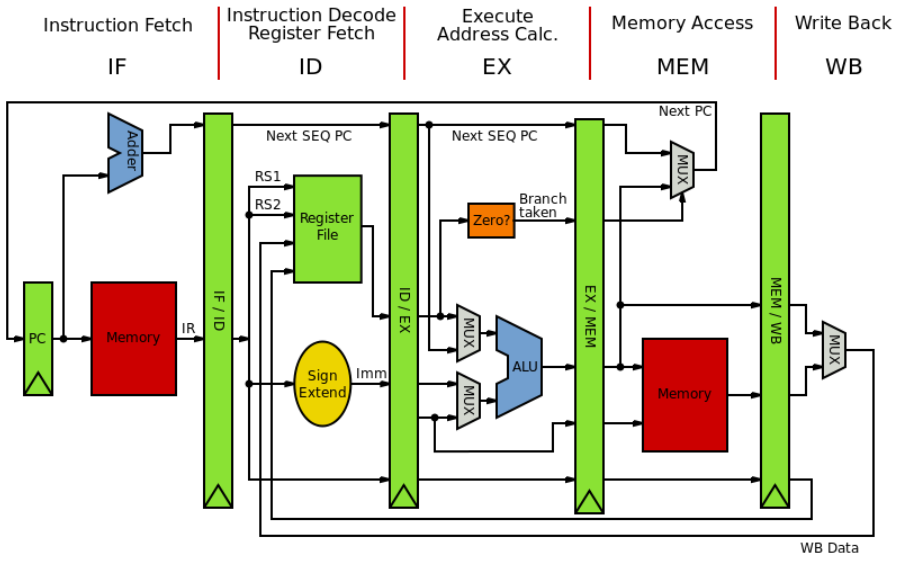


Figure 2.5: Classic 5-stage RISC pipeline [7]

Instructions in the RISC pipeline splits into 5 different stages.

1. Instruction fetch (IF) - fetches instructions from the instruction cache memory.
2. Instruction decode (ID) - fetches the registers which are about to be used.
3. Execute (EX) executes the instruction
4. Memory access (MEM) - data accesses are made, where data is accessed from the cache.
5. Write Back (WB) is the last stage writes back the result to a single register.

The classic RISC pipeline is very simple compared to modern piplines. 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] and Intel® Core™ Ivy-Bridge uses a pipeline length of 14-18 stages [17].

2.1.4 Cache memories

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

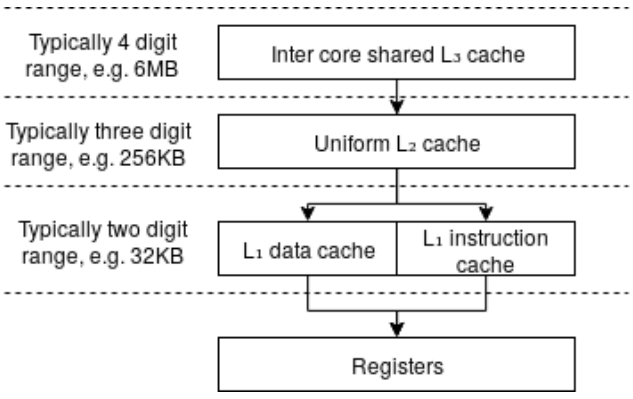


Figure 2.6: Intel® Core™ i5-3570k cache hierarchy

The cache is the second closest memory to the processor and can be used to increase the performance of applications significantly [10]. Modern Intel® chips often use several cache layers with different sizes. One of the most common designs includes one L_1-cache, one L_2-cache and one L_3-cache, where the L_1-cache is closest to the processor and therefore fastest, but also the least spacious. The L_2-cache is further away from the processor and therefore 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.

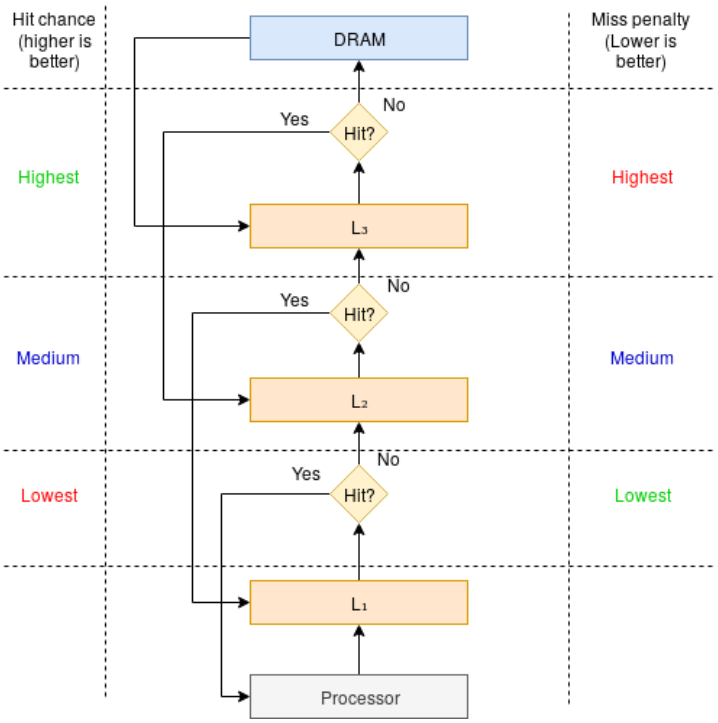


Figure 2.7: Three-level cache lookup procedure

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 forces a second lookup in the upper levels of the cache hierarchy to continue the search 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 in to 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 2.3.3.

All cache memories are however relatively small (usually ranging in the KB to single-digit MB sizes) compared to the DRAM. The small space of cache memories means that the likelihood of only cache hits during execution is small. The memory footprint of most applications is usually significantly more sizeable than the LLC. Since caches have only a limited memory space, there are memory replacement strategies such as least recently used (LRU) and first in first out (FIFO). These replacement strategies are used to determine which memory is evicted from the cache to make room for new incoming data. These strategies are called cache eviction policies. Eviction policies are, however, one of the prime forces behind cache contention - a hazardous situation where different applications continuously steal cache memory from each other. Cache contention is further described in detail in section 2.3.3 while cache mechanisms for dealing with incoming data are described in section 2.1.4.1. Misses in the L_3-cache lead to another trip in the hierarchy, forcing data fetches from the DRAM.

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:

- 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.

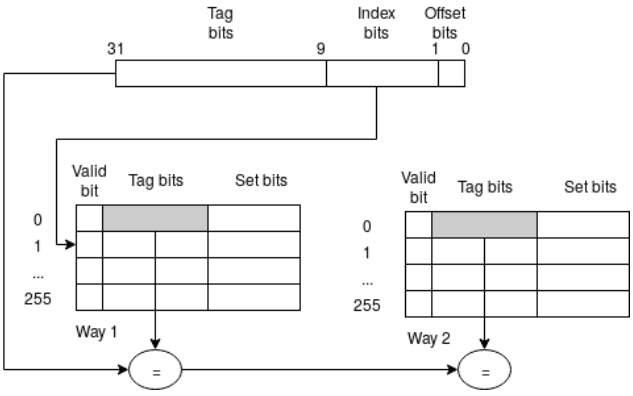


Figure 2.8: 2-way set-associative cache lookup example

The set-associativity of a cache creates a clear border between two ways. Incoming data will end up in either the first or second way of the cache. The placement decision is made using the offset bits of each cache line. If the offset bit is 0, the choice will fall on the first cache way; if the offset bit is 1, the decision will fall on the second cache way. The index bits of the cache line are used to address to which row within the way the data should be placed. The final tag bits are used during cache lookups, to match existing cache lines with new incoming data. If the tag bit of a newly 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. The set-associativity comes with the great benefit of flexibility when choosing cache replacement strategies such as LRU or FIFO. 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 2.12, which can be accomplished by techniques such as page coloring.

2.2 Parallel Computation

Parallelization of systems, to fully utilize multi-core capabilities, can significantly improve the execution time of entire systems. We consider parallel computation to be split into two subsets, application-level parallelism, and system-level parallelism. Application-level parallelism further splits a program into multiple smaller parts which can be run independently of each other. Application-level parallelism can achieved using the fork-join model [26], depicted in Figure 2.9 where array operations are assigned to individual cores, $c_0, \ldots c_3$, based on the array index.

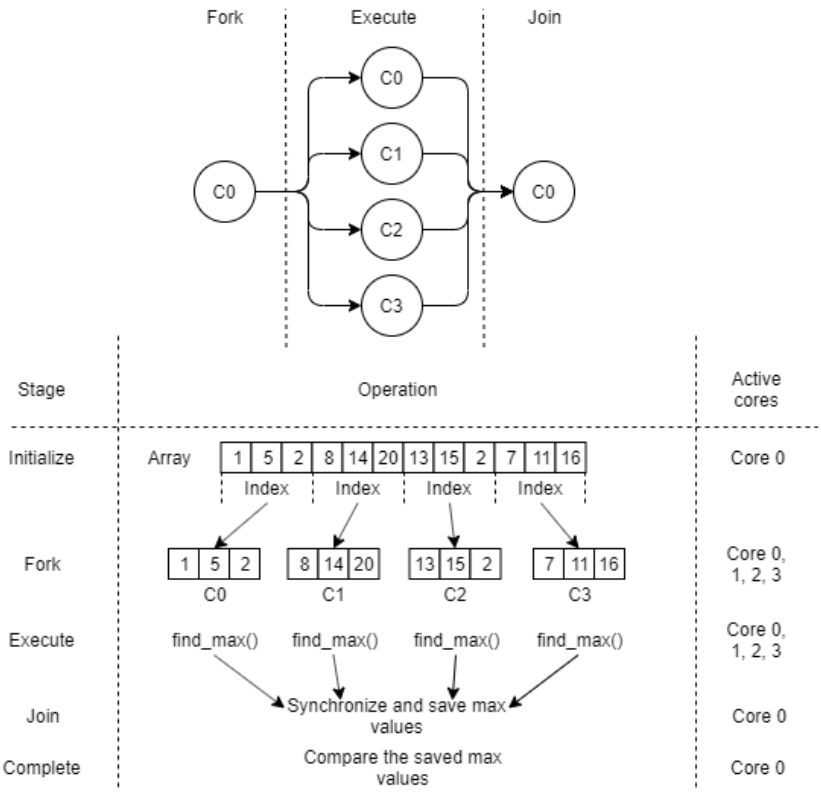


Figure 2.9: Fork join model applied on on finding max value in an array

The fork-join model is a parallelization method built on a simple concept, listed as follows:

1. Initialize - Identify parallelizeable parts of the program

2. Fork - Split the parallelizeable parts to individual threads

3. Execute - Execute the individual threads

4. Join - Synchronize thread execution and go back to the original core

It is also possible to parallelize entire systems, where applications are seen as runnables and are assigned to different cores. System-level parallelism is the most common approach to parallelize software and is enforced by the general Linux scheduler per default.

2.3 Resource sharing

Resource sharing splits into three categories: CPU-, memory-, and I/O-sharing [40], all of which are potentially hazardous in time-critical systems if left unhandled. We shortly describe the known main problems for these categories in the following subsections.

2.3.1 CPU sharing

The CPU is an essential part of a computer since it is responsible for executing the instructions a program is built upon. The problem with CPU contention arises with the multi-thread technology, where multiple tasks are allowed to execute within one execution environment. Thread execution is controlled by a process scheduler, which consists of several queues controlling the execution order of tasks. Linux runs applications in kernel-space or user-space. Kernel-space is most commonly reserved for kernel service tasks, while user-space is used for executing applications.

The most commonly used general Linux scheduler, also the scheduler used in this thesis, is called the Completely Fair Scheduler (CFS) [28]. There also exist more time-predictable, real-time oriented schedulers such as Rate Monotonic (RM) [21] and Earliest Deadline First (EDF) [13], most process threads are, however, executed using the CFS scheduler. Experiments presented in this thesis were created using a general Linux environment, the remainder of this thesis therefore discusses CPU contention under the regular Linux CFS scheduler.

Linux operates using several scheduling classes within the scheduler complex, a kernel-level scheduler, responsible for handling I/O's and kernel tasks, and a process scheduler - the CFS. The kernel-level scheduler runs at a higher priority than the process scheduler, which means kernel tasks will always get execution priority. CFS aims at maximizing CPU utilization and also supports process priority scheduling, which means a process with the highest priority will always run first. Maximizing the CPU utilization in multi-core systems can, however, become problematic since the scheduler now gets the opportunity to choose a core which potentially is executing time-critical loads. Consider the system depicted in Figure 2.10. There are 4 different applications with equal priority, App_0, App_1, App_2 and App_3, in the system (realistically, a default Linux system has hundreds of active processes), with different CPU utilizations executing on the cores C_0, C_1, C_2 and C_3 respectively.

Once App_4 with equal priority to the other applications is started, the scheduler has to choose which core App_4 should execute on. Since the goal

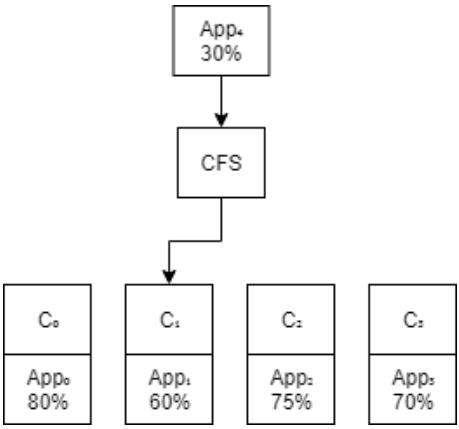


Figure 2.10: Example of thrashing

of the scheduler is always to maximize the CPU utilization, C_1 will be the chosen execution environment. App_1 will now share execution environment with App_4, and there is a high probably that App_4 preempts App_1 and therefore may stop the execution of App_1. The preemption can be problematic if App_1 is doing important time-critical activities, and is expecting not to be preempted. The problem, therefore, lies in App_1 being unexpectedly preempted by another application due to the scheduler choosing the wrong core.

2.3.2 I/O sharing

Sharing I/O's can be problematic since I/O often control the decision making in many systems. Problems with I/O sharing arises when I/O requests on different cores are served on a first in first out (FIFO) basis. A race condition appears when the I/O controller consumes the data. Consider the following scenario, where one application instantiates two processes, p_1 and p_2, for performance reasons. The processes execute on different cores and use the same I/O. There is a high risk that p_1 reads the I/O and thus consumes it, just about before p_2 was going to read it. Naturally, this can cause memory contention behavior since the value read by p_2 now has a different value than before p_1 read it. This can, therefore, cause severe miscalculations in a system.

2.3.3 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 2.11 exemplifies a system suffering from cache contention due to usage of multiple cores.

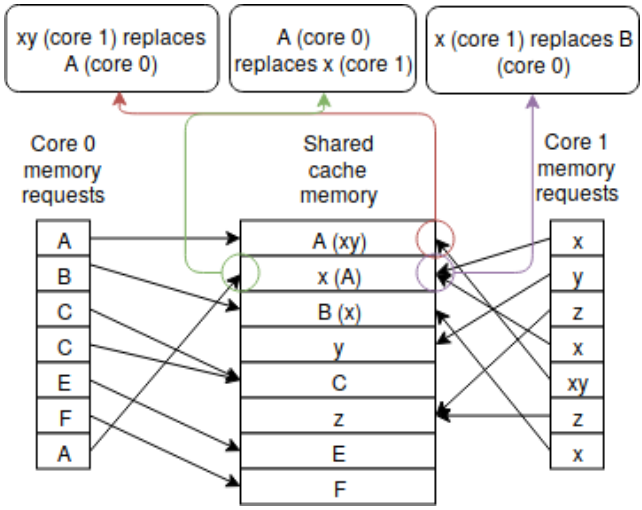


Figure 2.11: Example of thrashing

The figure shows the memory requests of two applications app_0 and app_1, executing on core 0 and core 1, during the period 0-5. The applications are completely synchronized, which means app_0 is running just about before app_1. In the case of completely synchronized applications, the shared cache memory starts storing the memory requests from each application in a sequential way. The cache memory requests starts with request A from core 0, stored in the first cache line, continues with request x from Core 1 and so on. 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, and was recently replaced by the xy memory request from app_1. The A memory request will thus result in a cache miss and suffer from 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 replaces 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.

2.4 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, which is 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 which are used for different purposes such as profiling for system optimization. To give a brief idea of what the PMU measures, Table 2.1 exemplifies a set of some internal PMU events.

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

These performance counters are originally per-core bound, which means that 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 measurement of process ID (PID) specific events rather than core-specific events. Tools such as perf [38], 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 setup the PMU events. Other tools such as the Performance API (PAPI) [25] re-use the PMU mappings created by perf and enable an even easier way for initialization and tagging tracking the counters of PIDs. We have used the PAPI framework to measure the PMUs on Intel® hardware and assembly instructions for measuring PMU events on ARM hardware.

2.5 Resource isolation

Isolation is a concept based on removing the natural resource sharing of a multi-core system without having to alter the hardware architecture, thus creating an isolated environment. Therefore, executing applications within an isolated environment will not affect applications in another environment.

Full isolation of entire systems can, however, become immensely complex due to a large number of hardware units within a computer, see Figure 2.12 (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 containers in order to provide full isolation including different techniques such as TLB coloring [29], DRAM bank partitioning [44], memory bus bandwidth scheduling [45], I/O virtualization [36].

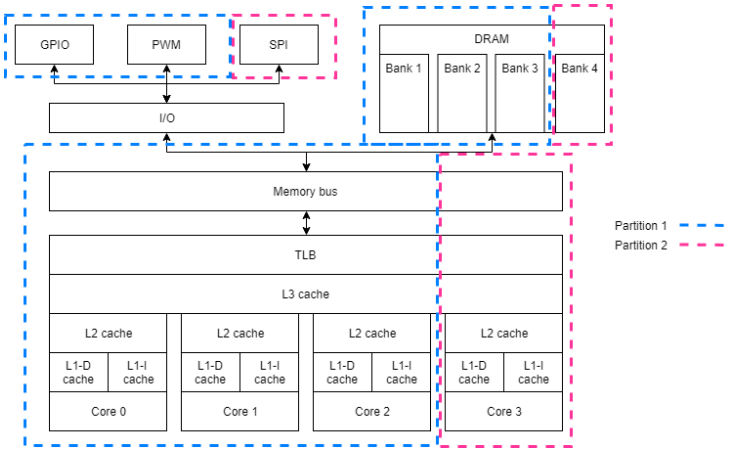


Figure 2.12: Example of a completely partitioned system

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 as a consequence of resource contention in one of the higher memory hierarchies. Therefore, there is a risk that indications of resource contention in a certain hardware unit can be falsely reported. Our main challenges do not lay in how to isolate specific hardware resources, but instead in identifying contention and using isolation techniques in an appropriate way.

2.5.1 Cache coloring - an example of an isolation technique

We exemplify cache isolation in Figure 2.13, through cache coloring [30] suffering from previously mentioned cache thrashing problem.

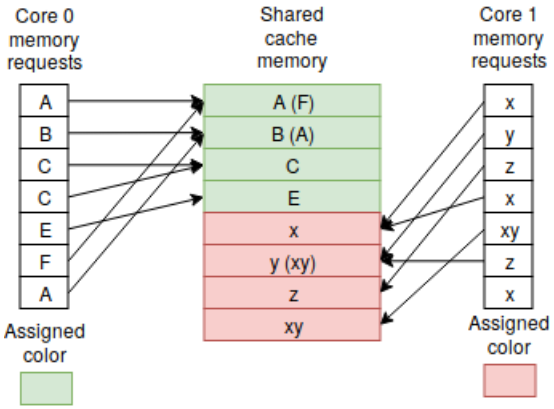


Figure 2.13: Thrashing avoided by cache coloring

The idea behind cache coloring is to remove the "shared" aspects of the cache through software techniques which alter the way on how memory is mapped from the DRAM to the cache. Memory requests from different applications are assigned specific colors, which correspond to specific memory regions within the cache. Enforcing the cache coloring methodology disables processes from different cores from using each other's cache lines. The only cache evictions which occur now will happen due the application itself, 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*. The cache is however not the only shared resource, and other approaches exist for isolating other units.

2.5.2 Feature detection algorithms

Feature detection algorithms are mathematical and logical expressions for defining objects and other image characteristics to computers. Autonomous systems such as self-driving cars [6] and assisted aircraft landings [14], therefore, often use such algorithms to detect obstacles and mission relevant key-points. We describe the basic feature detection mechanic as a three-step process:

1. Decode the input image - There exists a variety of different image formats, such as bmp, jpg, img, png and many more. The purpose of this

stage is to decode the image and move the data from the hard disk into a matrix in the RAM memory. Data points in the image are called pixels and are represented by 3 values; red, green and blue (RGB).

2. Process image data - Once the image data is moved to the DRAM, it is possible to process the data using mathematical equations. Simpler algorithms such as FAST [32] only compare the luminosity of adjacent pixels to a center pixel, while more complex algorithms such as SIFT involve Difference of Gaussian, nearest neighbour and hough transform [11] calculations.

3. Decision making - The image processing step will generate a set of key-points which the computer can use to make decisions. Feature detection algorithms are not 100% accurate, and, it may not be feasible to make decisions purely on processing output. Therefore, additional processing steps such as machine learning may become relevant to investigate connections between key-points and thus creating better feature selection.

We have selected feature detection algorithms as workload to many papers of this thesis since they provide an excellent example of industrial workloads and typically involve large-scale data-sets. Figure 2.14 exemplifies what to expect from a corner detection algorithm.

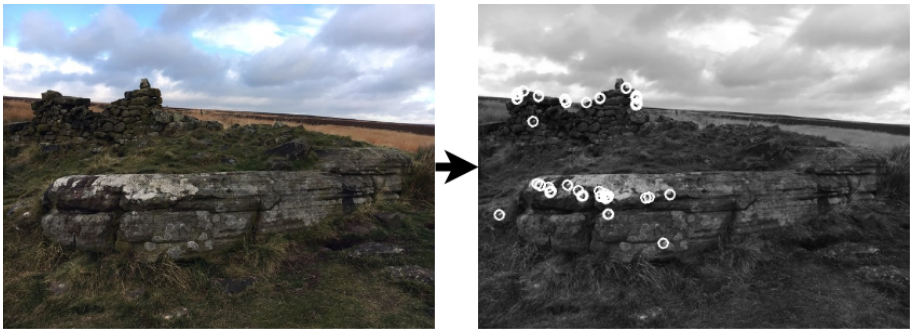


Figure 2.14: Example of the Harris algorithm executed on a stone picture

Feature detection algorithms are very interesting from a shared resource contention point of view due to the data size. The subsequent text will discuss the data usage of the Harris corner detection algorithm assuming a 6 MB large LLC.

Figure 2.14 exemplifies a Harris algorithm executed on an 841x271 image, which in the vision detection library OpenCV [27] corresponds to an

841x271x3 large matrix in program code, due to the RGB nature of images. Each RGB value is represented by a 8-bit character, which means the total size of the image once in RAM is $841 * 271 * 3 * 8/4 = 1.3$ MB, which is a substantial part of the cache memory.

If we dissect the Harris algorithm processing stages into smaller parts, the cache contention scenario becomes more evident. We split the Harris algorithm into three data intensive stages:

1. Grayscale conversion of image.

2. Apply Sobel operator [34] to the all pixels within the grayscale image and calculate the Harris Response.

3. Perform a non-maxima suppression, which filters out local maxima points within a radius.

The first step converts all pixels of the RGB matrix into a grayscale, using luminosity calculations according to: $Lum_{xy} = R_{xy} * 0.2126 + G_{xy} * 0.7152 + B_{xy} * 0.0722$. The end result of the grayscale conversion is a new matrix 841x271 sized matrix, consisting of unsigned characters and has a total size of $841 * 271 * 8/4 = 445$ KB, thus using almost one third of the available cache space.

The second step takes the converted grayscale image as input parameter and applies a Sobel filter to each individual pixel of the matrix. A Sobel filter is a 3x3 matrix which brings forth the edges in an image in both y- and x-directions. The Sobel filter results in an x-value and a y-value of each pixel of the grayscale matrix, and is used to calculate the Harris response for the individual pixel. Each Harris response is stored in another 841x271 matrix. The Harris response matrix is then sorted, the highest Harris response is inserted first in the list and the lowest is inserted last. Once the sorting is done, a non-maxima suppression calculation is made, which creates a radius and compares all pixel within that radius to each other in order to detect the maximum Harris response pixel among all pixels with a response above a Harris threshold value. Once the non-maxima suppression has been made, the Harris corner points have been determined. Figure 2.15 summarizes the Harris data usage.

In our scenario with an 841x271 image, each iteration of the Harris algorithm would use 2.2 MB of cache data. Most of this data can be re-used one time due to the relatively small image size, which significantly improves the execution time of the Harris algorithm, since the same data does not have to be written to the cache twice and assuming that the Harris algorithm is running as the only application in the system. It is, however, not realistic to assume

that the Harris algorithm would run as the single task in a system since an operating system typically hosts multiple kernel tasks. Furthermore, the actual point of using multi-core systems is to able to run multiple tasks in parallel, and therefore, a scenario with a Harris algorithm running as single task in the system is also not relevant.

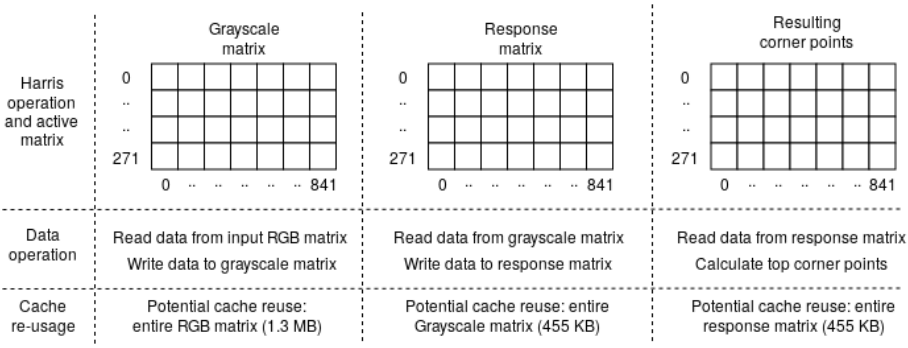


Figure 2.15: Illustration of the Harris algorithm data-usage

We have established that the Harris algorithm will use 2.2 MB of cache memory, which is 36% of the total available cache memory. The Harris algorithm execution time will become jittery due to cache contention if other applications on other cores utilizes a similar amount of cache memory. There also exists several other feature detection algorithms such as SURF [2] and SIFT [22] which are more complex and require an increased amount of memory. An increased amount of memory usage in turn increases the risk of memory contention.

Chapter 3

Research Overview

3.1 Problem formulation

We categorize the problems of this thesis into three areas: identification of shared resource contention, performance impacts of the proposed isolation techniques and finally reasonable setup and dynamic adjustment of isolation techniques.

- **Identification** - Identifying resource contention can be difficult due to the complexity of computer systems. Identifying the source of contention and assessing the impact on a specific resource is essential, since it is the only way of knowing which resource should be isolated.

- **Performance impacts** - Enforcing isolation techniques hinders processes from using the entirety of a shared resource. Partially removing the usage of the shared resource can mean performance penalties for a particular process. Isolation techniques are also often complex, which means run-time overhead may raise other issues.

- **Reasonable setup & dynamic adjustment** - Once a certain resource has been identified as the source of contention, it is possible to isolate this resource.

  For example, resources such as the internal memory cannot be fully isolated and discarded from usage by another core, since they are integral parts of the memory hierarchy. Instead, they can be partitioned, which means, assigning one part of internal memory to one application and another part of memory to another application.

Partition size may have an important performance impact on executing processes (see above). We see as a solution here, the dynamic adjustment of partition parameters during process run-time.

3.2 Research problem

The overall goal of the work leading up to this thesis was to investigate bottlenecks to execution time-predictability in multi-core systems. More specifically, what are the limiting factors in achieving full time-predictability of multi-core systems? Our initial research pointed us towards two areas of interest - identifying resource contention and efficiently solve the resource contention using isolation techniques. Our research goals are listed as follows:

RG1 - Finding methods to determine the resource contention caused by multiple users of a single resource.

RG2 - Using isolation techniques to partition shared resources for processes and using them efficiently such that the system maintains the performance of critical tasks.

3.3 Research methodology

This thesis addresses the mentioned research goals using empirical studies. The empirical studies are concentrated on executing industrial workloads. We have used a set of feature detection algorithms which are common in obstacle localization and avoidance domains such as avionics and autonomous vehicles. We also used matrix multiplications which are an integral part of feature detection algorithms.

- Our methodology towards addressing RG1 uses performance counters to monitor individual applications and characterize memory contention, see papers A and B.

- To reach our second research goal, we studied partitioning to find applicable techniques, see papers C and D. The partition techniques studied includes static virtualization using the Jailhouse hypervisor [36] and page coloring using PALLOC [44].

Table 3.1 shows how the publications contribute to the goals of the thesis.

Papers	RG1	RG2
Paper A	✓	
Paper B	✓	✓
Paper C		✓
Paper D		✓

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

Our main research focus is investigating the effects on adapting single-core programs to multi-core systems. All our papers use a baseline versus parallel style. We first implement application tests which are run independently on one core, without any parallelization technique or deliberately disturbing loads on other cores. The independent execution of an application is our reference point for comparison when moving towards a parallel environment and we call this *baseline* execution. Once we have established a baseline execution of an application, we make measurements of the application running in a parallel environment. The parallel implementations we use in this thesis have two variations; fork-join adaptation (called forked execution, see paper A and B) and contention implementation (called loaded execution, see paper C and D). Our goal of the forked-join adaptation is implementing parallel versions of feature detection algorithms, while investigating limiting factors such as cache contention. Our goal of the contention implementation is simulating real systems where many processes run on different cores and, thus, causing resource contention through over-commitment of resources such as the cache.

We continue with our baseline execution versus parallel execution approach in our tests of partitioned systems - see paper C and D. Here, we use two baseline measurements; B_L, which is our baseline measurement value for a regular Linux system running without extensions and B_P, which is our baseline measurement value for a regular Linux system running with isolation extensions. Equation 3.1 shows the usage baseline metric values B_L and B_P used to calculate the overhead (O) of an isolation technique. We compare the baseline measurements to two parallel measurements which run with leeches (intentionally memory heavy and CPU heavy applications) on either the same core (CPU and cache leech) or other cores (LLC and memory bus leech) in the system in order to force shared resource contention. C_L denotes a leech loaded regular Linux system and C_P denotes a leech loaded Linux system running with partition extensions. We use C_L and C_P to calculate the isolation coefficient (I), a metric which quantifies the isolation achieved from a partition solution, see Equation 3.2.

$$O = B_L - B_P \tag{3.1}$$

$$I = \frac{C_L - O}{C_P} \tag{3.2}$$

We compare the parallel execution to the independent execution of an application using two metrics - execution time and performance counter. Execution time measurement remains the same for all tests. We place a timestamp directly before application start and directly at end of application start to measure execution time. We also measure performance counters, which are hardware triggered events that count events such as cache misses. We have used two approaches of performance counter measurement. The first approach measures the total performance counter amount during the entire application execution. This measurement is done in a similar style as the execution time measurement, where the performance counters are initialized and started right before application execution and collected at application end. The second approach measures the run-time characteristics of an application, where the performance counters are monitored at a fixed time interval. We present all papers as empirical studies and execute each test at least ten times for increased validity. The test results of each paper are presented as either average or median values of the entire data collection.

3.4 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 goals.
3. Categorize the research problems into thesis contributions which more clearly defines what problem is solved.
4. Define hypothesis for an application in a multi-core setting.
5. Monitor application run-time characteristics in that setting.
6. Use deductive reasoning on application behavior in the current setting.
7. Evaluate possible isolation options for application.

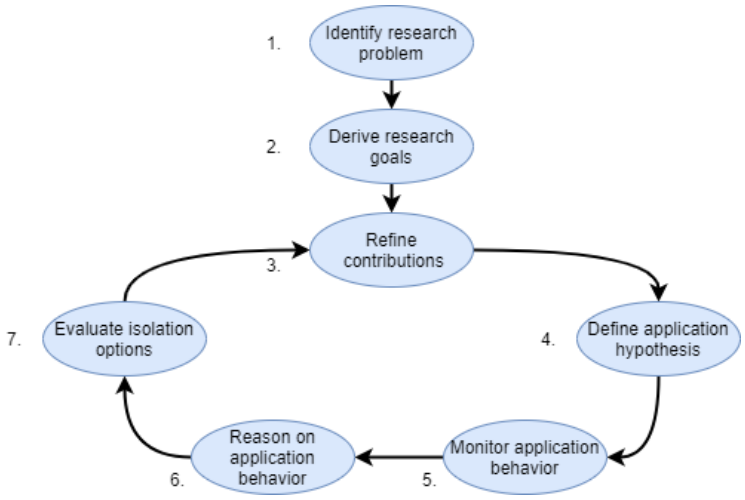


Figure 3.1: Research methodology

The first step is to establish an overall goal of the thesis. We define two research goals based on the output of the related work.

Step 1 provides an overall goal for the thesis while step 2 and 3 dissects the overall goal into smaller problems. Our iterative investigation process starts from step 3, where a new application hypothesis is presented for each thesis contribution. We create tests and monitor the application behavior to verify the hypothesis and finally use deductive reasoning to formulate why the application behaves a certain way. The output from the deductive reasoning results in an evaluation on what isolation options there are for an application given a certain behavior. Once completed, we refine our research contribution and continue the iterative process.

Chapter 4

Related work

Many different studies exist, which tries to isolate shared hardware resources that affect the execution time predictability of applications. We divide a multi-core system into three critical parts which are in risk of resource contention, including the CPU, the internal memory components, and the I/O units. In this thesis we have put our focus mostly on investigating internal memory. The following subsections discuss related work which are in line with our research goals.

4.1 Resource monitoring

Performance counters are very handy tools for profiling system performance. Works such as Cache Pirating [12] introduce methods which steal cache to determine the application cache demand. Jägemar et al. [18] show how it is possible to correlate the hardware resource counters with application performance using a Pearson coefficient and furthermore deny a process from over-committing to hardware resources such as the shared cache. Application execution often splits into different execution phases [33], bound to different resources. Such phases include cache dominant, bandwidth dominant and GPU dominant phases which is also important to realize when applying partitioning techniques. The previously mentioned phases can also be re-occurring, e.g., the characteristics of the application may shift from memory dominant to computing dominant and then go back to memory dominant again [35]. The previously mentioned papers show how it is possible to build an understanding of application behavior using the internal performance counters of the computer. Our methodology continues the work on using performance counters, however we take a different approach on what we can do with the knowledge of

application behavior. We use performance counters to derive shared resource contention and give suggestions on which isolation methods are appropriate.

4.2 Software isolation techniques

Isolating the internal memory hierarchy includes cache partitioning (also called cache coloring and page coloring). The cache partitioning technique prevents certain processes from using certain cache lines (also known as colors). Cache partitioning can be enforced to increase energy efficiency [41], [15], execution time of processes [30], improving deadline miss rates [16] and also determinism of hypervisor partitions [20]. Another interesting approach is the "partition-sharing" concept [4], which allows for multiple processes to share the same partition. The introduction of the Linux perf tools [38] has made it significantly easier to read the performance counters of Linux based computer systems. Tools such as Coloris [43] use the performance counters to create cache partitions according to the cache usage of SPEC2006 benchmarks.

Resource isolation can be difficult to verify due to the large amount of hardware resources in a computer. *Application performance isolation* [24] is used as a term for when the performance of an application remain unchanged when running in a resource contentious environment [19, 37, 45, 31]. Matthews et al. [24] performed a study on how to quantify the performance isolation given by a hypervisor using six different tests including memory, disk, scheduling, CPU, network send and network receive.

The previous works can be seen grouped on two branches. The first branch is isolating hardware from simultaneous use of other processes. The second branch is verifying that isolation has actually been achieved. The main differences between our works and the previous works are the combination between performance counters, execution time and isolation. We use performance counters combined with execution time as means for verifying isolation. The isolation measurements are used for implementing solutions to increase performance and predictability of system where it is needed.

4.3 Evaluating performance

Measuring performance degradation's and performance isolation is important to our thesis, since it enables us to quantify the amount of isolation given by an isolation technique and also measure the effectiveness of the isolation technique. The perhaps most common approach is measuring the slowdown

ratio given between a baseline environment system and its extended counterpart [39, 24, 42, 5]. 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 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, the methodology of measuring the performance of an application will differ depending on the application purpose. It is not sufficient to use execution time as a performance metric of a TCP/IP stack algorithm, 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's in applications, rather than looking at the execution as the prime measurement for performance. There are countless studies evaluating how to increase the performance through parallelisms, measuring system performance effects from to a newly implemented algorithm, measuring performance degradation's of virtual machines and many more. In facts, the entire field of computer science is imbued with performance measurements, whenever a new application is implemented, the programmer will want to know the performance of the application and how it plays with the system. We try to add a methodology for measuring performance, while maintaining an understanding on why the performance behaves like it does.

Chapter 5

Thesis contributions

5.1 Thesis contributions

We present three contributions of this thesis, which are tied to our two research goals, see Figure 5.1. The first contribution targets identification of shared resource contention. The second contribution targets testing of available isolation techniques. The third and final contribution targets dynamic adjustment and efficient usage of isolation techniques. The research contributions are distributed in three conference papers and one work in progress paper. Section 5.1.1 describes the research challenges and pairs them with the relevant research question, and Section 5.3 provides a short overview of the papers.

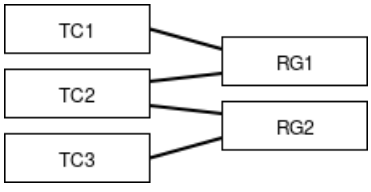


Figure 5.1: Mapping between thesis contributions and research goals

5.1.1 Contribution 1 - Identification of shared resource contention

The first thesis contribution focuses on the identification of shared resource contention, mainly in the LLC, but it also touches on memory bus contention and on CPU contention. The main idea behind this contribution is to develop a method for the identification of shared resource contention using the performance counters of a CPU. The associated papers discuss about the hardware performance counters of the LLC to create a model for describing shared

resource contention. We use the terminology of *baseline performance* and *loaded performance* to describe resource contention, defined as follows.

If an application is executing as the single application on one core, without any deliberately disturbing loads on the same core or other cores, it runs at *baseline performance*. The execution of an application further produces two relevant, measurable metrics: the *execution time* and the *PMU events* which we use to identify resource contention. If we deliberately place disturbing loads on either the same core or on other cores, the application executes at *loaded performance*.

In paper A, we investigate the parallelization of the Features from Accelerated Segment Test (FAST) feature detection algorithm using a fork-join approach. Paper B extends the study of paper A, presenting an investigation on forked versions of all available OpenCV algorithms. In paper C, we insert *leeches* - small memory-intensive workload programs which are intended to disturb the execution of App_0 on the other cores. The resource contended environment is called *loaded* execution. Figure 5.2 depicts the program core setup from paper B and paper C, respectively.

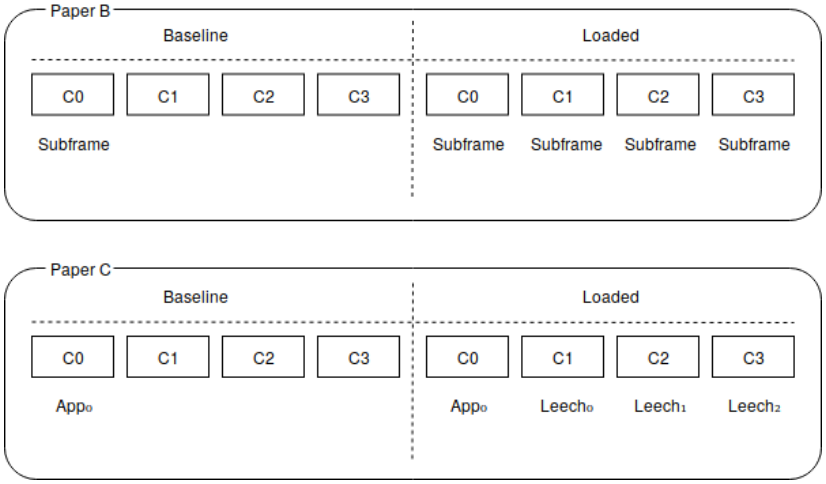


Figure 5.2: Identification of resource contention

We assume that the baseline performance will always be better than the loaded performance, if the application is a subject of resource contention. We can, therefore, draw the assumption that resource contention is present if the two following statements are true:

1. The loaded performance of an application is worse than the baseline performance of the application.

2. The loaded performance counter of an application is significantly higher (which means higher cache usage), than the baseline performance counter of the application.

This contribution is partially presented in Paper A, using a PowerPC architecture. The contribution is further investigated in Paper B, using an Intel architecture, and finalized in Paper C, based on an ARM-64 architecture. All these partially answer RG1. Our main goal of this contribution was to create a generalizeable method for verifying resource contention which spans over different instruction set architectures.

5.1.2 Contribution 2 - Apply isolation techniques and understand the performance trade-off

This contribution enforces available isolation techniques and investigates the performance impact caused by the isolation technique. Isolation techniques are often complex, since they, in some way, have to override mechanisms of the operating system. Adding extra isolation mechanisms to an operating system can thus decrease the overall performance of tasks. Application of isolation techniques, therefore, presents two important questions. Firstly, does the proposed technique provide the promised isolation? Secondly, how much impact does the isolation technique have on the performance of applications executing in the system? To answer these questions, we created a methodology to verify the isolation gained, that is, increase I (Equation 3.2), employing the proposed methodology, using the setup depicted in Figure 5.3.

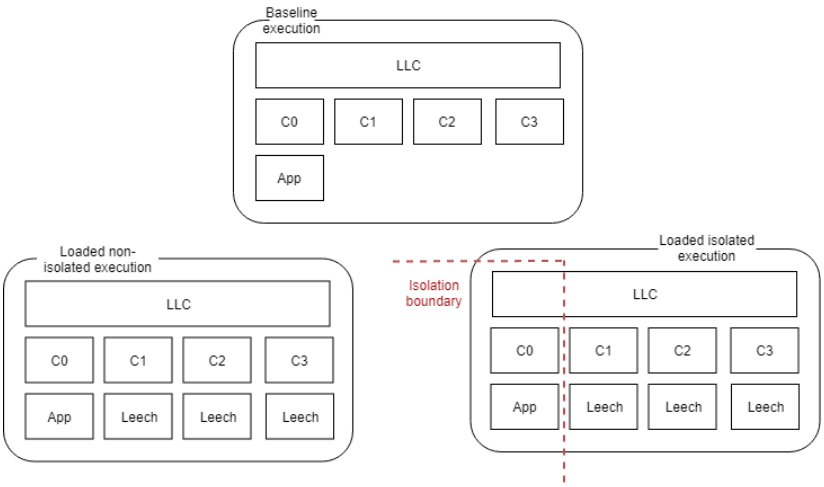


Figure 5.3: Verification of isolation

We divide our methodology into two steps: the first one identifies the baseline execution time of an application; the second one measures the loaded execution time of the same application. 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.

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

As previously stated (Section 5.1.1), we expect that a loaded non-isolated environment suffers significant performance degradation due to shared resources contention. However, an isolated environment is expected have an execution time close to the baseline execution time, with variations given by overhead (see Equation 3.1) of the isolation technique. We have tested two isolation techniques, the static Jailhouse partitioning hypervisor [36] and the combined LLC and DRAM partitioning kernel module PALLOC [44]. We measured the degree of isolation obtained using the Jailhouse hypervisor, and also performed similar experiments on 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 C mainly addresses this contribution and paper D also touches the topic. This contribution partially answers RG1 and RG2.

5.1.3 Contribution 3 - Setup and adjustment of isolation techniques

This contribution explores isolation techniques and investigates ways to efficiently allocate an adequate amount of resources to a specific core. We present a generic method for allocating hardware resources during run-time, using the combined DRAM/Cache partitioning tool PALLOC [44], as proof of concept. The Jailhouse hypervisor is currently a static approach, which that means that once deployed, the parameters of the isolation will remain the same until the reboot of the environment. PALLOC, is, on the other hand, dynamic, which means that it is possible to change the parameters of the isolation during run-time and adapt the parameters to the current needs of an application. Allocating too few resources to an application may cause the application to run slower than expected. Allocating too many resources to an application may steal resources from other applications which also needs them. Figure 5.4 exemplifies the resource allocation problem, where a 512x512 matrix multiplication, run for 100 times, is granted different amounts of cache partitions. The y-axis

shows the execution time of the matrix multiplication versus the sample number of the matrix multiplication on the x-axis.

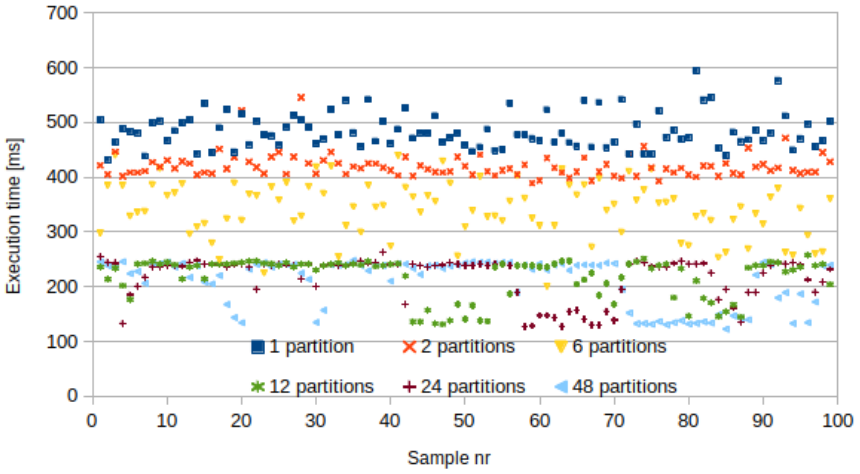
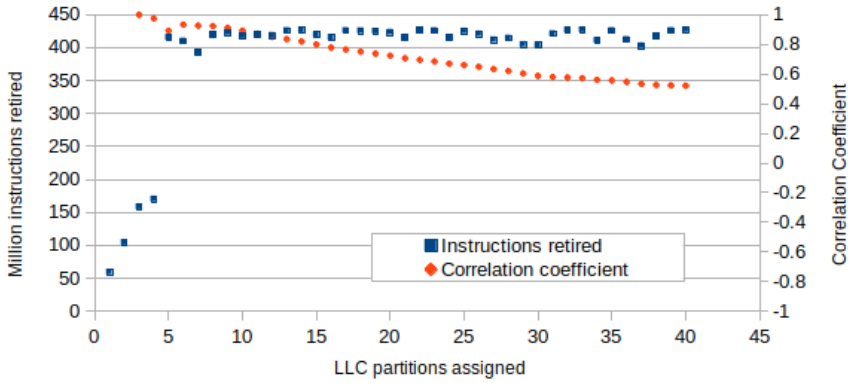


Figure 5.4: Matrix multiplication using different cache partition sizes

It is clear that allocating a very small cache partition space to the matrix multiplication severely downgrades the execution time, compared to the more sizeable cache partition allocations. One may also notice, that the the increase of execution time levels out at 12 partitions assigned to the matrix multiplication. This plateau is a consequence of cache re-usage being completely saturated, where the data access patterns of an algorithm has reached the point where all cache lines that can be re-used - are being re-used. We call this occurrence *point of saturation*. Adding more cache partitions to the matrix multiplication beyond the 12 partitions mark will not improve the execution time any further, it would occupy an unnecessarily big amount of cache space. Therefore, it is essential to understand how much cache space an application actually needs.

We propose a correlation-based methodology for finding a feasible cache memory allocations for executing applications. We use the Pearson correlation coefficient [3] to localize trends in execution time when increasing the amount of cache given to a process. The basic idea is to slowly increase the cache partition size for applications until the point of saturation has been found. We use the Pearson correlation coefficient to determine when the point of saturation has been reached, through correlating the number of *instructions retired* and the history of cache sizes. Figure 5.5 depicts an example of a correlation run on the Scale Invariant Feature Transformation (SIFT) [22] algorithm. The red line shows the correlation between the instructions retired and the increased

Figure 5.5: 4 MB SIFT execution



cache space. The blue line shows the instructions retired, i.e., the performance of the SIFT application.

We monitor the performance of an application during run-time using the instructions retired as performance metric. It is beneficial to use instructions retired over of the execution time of an application since the performance counters can be monitored ad-hoc and therefore does not require complex communication schemes between a controller and an application. We continuously measure the performance counters and store cache partition history-data throughout the execution of an application. Once we reach the correlation threshold, we have found an adequate cache partition size for the application. This contribution partially answers RG2.

5.2 Summary of papers

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

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

Papers	TC1	TC2	TC3	RG1	RG2
Paper A	✓			✓	
Paper B	✓			✓	
Paper C	✓	✓		✓	✓
Paper D		✓	✓		✓

5.3 Overview of included papers

5.3.1 Paper A - Investigating Execution-Characteristics of Feature-Detection Algorithms

Jakob Danielsson, Marcus Jägemar, Moris Behnam, Mikael Sjödin

Summary We evaluate a fork-join model applicable to feature detection algorithms and present a method for measuring how well the algorithm performance correlates with hardware resource usage. We have applied our method to the Featured from Accelerated Segment Test (FAST) algorithm. Our characterization of FAST reveals that it is an algorithm with excellent parallelism opportunities, resulting in an almost linear speed-up per core. Our measurements also show that the performance of FAST correlates very little with the number number of misses in the L1 data cache. Further measurements also show low correlation with L1 instruction cache, data translation lookaside buffer, and L2 cache. Thus, the FAST algorithm will not harm the execution time when the input data fits in the L2 cache.

Thesis contribution TC1

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 22^{nd} Emerging Technologies and Factory Automation (ETFA), 2017, IEEE

5.3.2 Paper B - 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 TC1

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

5.3.3 Paper C - Testing Performance-Isolation in Multi-Core Systems

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

Summary In this paper, we increase the isolation scope, where we previously only looked at the LLC, to also investigate contention in the CPU, LLC, and also the memory bus. We use this 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 require 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 TC1 and TC2

Research goal RG1 and RG2

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

5.3.4 Paper D - 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 TC2 and TC3

Research goal RG2

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

Chapter 6

Conclusions & future work

The main goal of this thesis work is to understand the origins of shared resource contention, investigate the reasons for it, propose means to solve the resource contention, and finally evaluate the suitability of the solutions. We have investigated the optimization of computationally heavy tasks such as feature detection algorithms using a fork-join method to form a resource-contentious environment. The feature detection algorithms shows a variety of speedups using a fork-join model. Some algorithms such SIFT and FAST show a substantial execution time improvement. Other algorithms such as SURF and ORB show almost very small execution time improvements on on higher resolution frames. Our methodology in paper B uses performance counters and execution time to investigate the reasons for the low performing algorithms. We conclude that the low performance improvements of the feature detection algorithms executing on high resolution frames are a consequence of LLC contention.

We expand our methodology to investigate resource-contention on other shared resources including memory bus, CPU, L1 cache and also the LLC using additional programs - the so-called leeches. The purpose of a leech is to create resource contention within one targeted resource, such as the LLC or the memory bus. We test the Jailhouse hypervisor with two kinds of leeches; memory leeches and CPU leeches. We use memory leeches to create contention in the caches and on the memory bus, through intensive memory reads and writes into different buffers. We use CPU leeches to create contention in the CPU, through forced execution swaps. Jailhouse showed to be effective when isolating local resources such as CPU and local caches, causing almost no overhead. PALLOC, on the other hand, also provides excellent resource partitioning but leads to a substantial slowdown of the task executions.

The main problem of using partitioning techniques - apart from potential overhead - are their difficult setups. Partitioning techniques reserve a specified

amount of resources to its partitioned environment, and it can be hard to determine precisely how much resources a partitioned environment needs. It is especially hard to know the resource requirements of an application. Allocating too few resources will risk that the execution suffers performance degradation as a consequence. Allocating too many resources will risk that the allocated resources bottleneck other partitioned environments. We show, in our final contribution, how to use our correlation-based controller - called LLC-PC - is used to find the saturation point of cache partitioned systems.

Future work includes extending the resource partition controller to cover more isolation techniques other than PALLOC. Techniques such as bandwidth partitioning and I/O partitioning exists and they are needed to enforce full isolation of a system. Employing such techniques will, however, cause a large system complexity since it means re-routing the data pathing of every single instruction. Therefore it is also very important to study the performance consequences of enforcing full isolation. It may become necessary to implement a decision-making solution which only partitions the most time-critical tasks and leaves the non-critical un-isolated. Such a solution can also implement a requirement specification for tasks, where user-defined, which inserts inputs such as minimum acceptable performance.

Another future work comes from the inspiration of Paper A and paper B, where the FAST algorithm is as a non-cache bound algorithm. Applications often tie to several different resources, such as caches, TLBs, CPU, and the memory bus. There is a chance that an application has a performance tightly bound with the CPU and FPU, such as the FAST algorithm. The FAST algorithm is however not bound to the caches, memory bus or I/O's. Inserting the FAST application to LLC-PC will only increase the overhead and not the performance since FAST is not cache-bound. It is, therefore, necessary to extend LLC-PC to include more resources such as TLB and CPU.

Extending LLC-PC to include more resources, however, requires careful planning - to what resources does an application bind? An in-depth Classification of applications will, therefore, become very important when extending the cache partition controller to span over other shared resources. Such a classification scheme enables us to determine which partition type could be useful for an application, and thus, control the partition size.

Bibliography

[1] ARM. Cortex-a53. `https://developer.arm.com/ip-products/processors/cortex-a/cortex-a53`. Accessed: 2019-11-04.

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

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

[4] 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.

[5] 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.

[6] A. Cherubini, F. Spindler, and F. Chaumette. Autonomous visual navigation and laser-based moving obstacle avoidance. *IEEE Transactions on Intelligent Transportation Systems*, 15(5):2101–2110, 2014.

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

[8] J. Danielsson, M. Ashjaei, M. Behnam, T. Sorensen, M. Sjodin, and T. Nolte. Performance evaluation of network convergence time measurement techniques. In *2017* 22^{nd} *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–7. IEEE, 2017.

[9] J. Danielsson, N. Tsog, and A. Kunnappilly. A systematic mapping study on real-time cloud services. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 245–251. IEEE, 2018.

[10] 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.

[11] R. O. Duda and P. E. Hart. Use of the hough transformation to detect lines and curves in pictures. Technical report, Sri International Menlo Park Ca Artificial Intelligence Center, 1971.

[12] 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.

[13] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino. An edf scheduling class for the linux kernel. In *Proceedings of the 11th Real-Time Linux Workshop*, pages 1–8. Citeseer, 2009.

[14] Z. Fleischman and C. Sullivan. Optically assisted landing of autonomous unmanned aircraft, May 5 2016. US Patent App. 14/631,520.

[15] X. Fu, K. Kabir, and X. Wang. Cache-aware utilization control for energy efficiency in multi-core real-time systems. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 102–111. IEEE, 2011.

[16] G. Gracioli and A. A. Fröhlich. An experimental evaluation of the cache partitioning impact on multicore real-time schedulers. In *2013 IEEE* 19^{th} *International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 72–81. IEEE, 2013.

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

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

[19] K. Jian, Z. X. Dong, N. Wen-wu, Z. Jun-wei, H. Xiao-ming, Z. Jian-gang, and X. Lu. A performance isolation algorithm for shared virtualization storage system. In *2009 IEEE International Conference on Networking, Architecture, and Storage*, pages 35–42. IEEE, 2009.

[20] 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.

[21] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *[1989] Proceedings. Real-Time Systems Symposium*, pages 166–171. IEEE, 1989.

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

[23] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), 2002.

[24] 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.

[25] 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.

[26] L. Nyman and M. Laakso. Notes on the history of fork and join. *IEEE Annals of the History of Computing*, 38(3):84–87, 2016.

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

[28] C. S. Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

[29] 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.

[30] 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.

[31] H. Raj, R. Nathuji, A. Singh, and P. England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 77–84. ACM, 2009.

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

[33] 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.

[34] V. Sanduja and R. Patial. Sobel edge detection using parallel architecture based on fpga. *International Journal of Applied Information Systems*, 3(4):20–24, 2012.

[35] 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.

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

[37] G. Somani and S. Chaudhary. Application performance isolation in virtualization. In *2009 IEEE International Conference on Cloud Computing*, pages 41–48. IEEE, 2009.

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

[39] 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.

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

[41] W. Wang, P. Mishra, and S. Ranka. Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems. In *2011* 48^{th} *ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 948–953. IEEE, 2011.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

Part II

Included Papers

Chapter 7

Paper A
Investigating Execution-Characteristics of Feature-Detection Algorithms

J. Danielsson, M. Jägemar, M. Behnam, and M. Sjödin. Investigating Execution-Characteristics of Feature-Detection Algorithms. In 22^{nd} *Emerging Technologies and Factory Automation (ETFA)*, IEEE, 2017.

Abstract

We discuss how to obtain information of execution characteristics, such as parallelizability and memory utilization, with the final aim to improve the performance and predictability of feature and corner detection algorithms for use in e.g. robotics and autonomous machines. Our aim is to obtain a better understanding of how computer vision algorithms use hardware resources and how to improve the time predictability and execution time of such algorithms when executing on multi-core CPUs. We evaluate a fork-join model applicable to feature detection algorithms and present a method for measuring how well the algorithm performance correlates with hardware resource usage. We have applied our method to the Featured from Accelerated Segment Test (FAST) algorithm. Our characterization of FAST reveals that it is an algorithm with excellent parallelism opportunities, resulting in an almost linear speed-up per core. Our measurements also reveal that the performance of FAST correlates very little with the number number of misses in the L1 data cache, L1 instruction cache, data translation lookaside buffer and L2 cache. Thus, the FAST algorithm will not have a negative effect on the execution time when the input data fits in the L2 cache.

7.1 Introduction

Robots are becoming more autonomous which means they are getting more dependent upon perception algorithms for object detection. Object recognition algorithms often use preprocessing algorithms to extract changes in color or features in an image, called feature detection algorithms. Since the robot is getting a continuous video stream, it is important that the response of the feature detection algorithm is fast enough so that the robot can maintain uninterrupted perception of the environment.

Many different feature detection algorithms exist, which use many different execution patterns. Some algorithms, themselves, are depending upon other feature detection algorithms to do preprocessing before doing the actual work, while others execute directly on the raw image data. Different feature detection algorithms also work in finding different targets in an image such as lines, corners and edges. Perception often make use of many different, and well established, feature and corner detection algorithms such as FAST, SIFT, SURF, and Harris. Often they are combined to achieve better object recognition. However, the algorithms which are combined have very different execution patterns which put stress on how to use the computer hardware efficiency to meet the timing. Multi-core architectures offer great opportunities for executing multiple algorithms on different cores and also enables the workload of a single algorithm to be spread out across multiple cores. However, since feature detection algorithms use different execution patterns, they exhibit very different memory-access patterns which impact on both the predictability and execution time of the algorithms. Characteristics which affect the predictability and execution time of an algorithm can be, but are not limited to, resource utilization, execution time, possibility to distribute the workload to different cores as well as how to schedule different algorithms together to achieve maximum quality of service. To investigate how these characteristics affect the system performance, a firm understanding of the algorithms properties and how it use hardware resources is needed. In this paper, we discuss aspects which are important to consider when using a feature detection algorithm as a real-time application executing on a parallel platform. As a case study, we have evaluated the Features from Accelerated Segment Test (FAST) [12] algorithm with respect to these above mentioned categories by investigating the algorithm behavior.

The rest of the paper is organized as follows. Section 7.2 presents a brief study of related work. Section 7.3 presents the requirement to build the testing tool. Section 7.4 show our methodology and measurements on the characteristics parallelism and memory utilization, Section 7.4 presents challenges occurring when executing FAST on multiple cores, while Finally, Section 7.5 concludes the paper and directs to future work.

7.1.1 The FAST algorithm

The FAST algorithm is used for detecting features and corners in images. The main mechanism of FAST is based upon using a Bresenham circle of radius 3 (depicted in figure 7.1) which is compared to all pixels in an image matrix. The execution of FAST is divided into two steps: determining if a selected pixel is an **Interest Point** and determining if the interest point is a **Corner**.

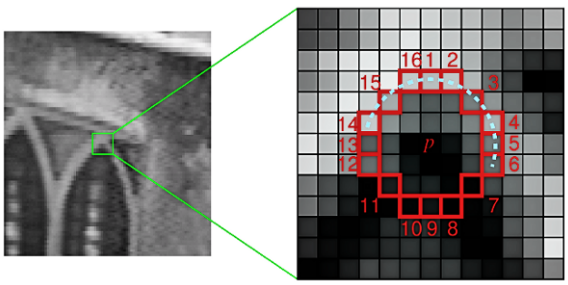


Figure 7.1: FAST algorithm corner example where p is the currently selected pixel for analysis

Two threshold values are used for making this decision. The first threshold is the intensity threshold (I_t) which is a percentage value that is applied to the pixel intensity (I) of all pixels within the Bresenham circle. If a pixel in the Bresenham circle is (I_t)% darker or brighter than the currently selected pixel, it is considered as a feature. The second threshold value N is used for deciding how many pixels in the Bresenham circle should be features in order for the currently selected pixel to be considered as a corner. To determine if the selected pixel is an interest point, FAST compares the pixel intensity value of the four utmost pixels - marked in Figure 7.1 as pixel 1, 5, 9 and 13 with the intensity of the currently selected pixel. If at least 3 of the utmost pixels are considered features, the current pixel is marked as an interest point and the algorithm continues to execute, else break. In the second step, the FAST algorithm compare the currently selected pixel to the rest of the pixels in the Bresenham circle. If at least N contiguous pixels are considered as features,

the current pixel is considered as a corner [12]. Since different thresholds will affect execution time, we have executed the tests using 10%, 20%, 30% and 40% as threshold values for I_t and a threshold value of 12 for N.

7.1.2 Hardware Resource Monitoring

The computational performance of the CPU is not the only limiting factor when evaluating the performance of an algorithm. Often, an algorithm process data that read from main memory putting a severe strain on memory buses and various cache levels. An algorithm may also suffer from other congestion-related side-effects; For example, if the branch prediction unit fails to predict the execution flow of the program. One of our goals is to monitor and evaluate the hardware resource usage of several edge detection algorithms. Having a thorough understanding of the system-level performance together with the low-level shared resource utilization makes it possible to understand better how to implement and deploy different algorithms [2] efficiently. Understanding the hardware usage of the algorithms will also make it easier to select the optimal hardware without expensive resources overprovisioning providing a greater throughput [3]. We estimate that it is possible to substantially improve the overall system performance of co-located algorithms by optimizing the core allocation of algorithms [4]. We will use a performance monitoring application that utilize the Performance Monitor Unit (PMU) to continuously monitor the performance of selected processes. Our application utilizes the Perf API for convenient PMU configuration.

7.2 Related work

Different works for comparing Edge detection techniques have been done, whereas the comparisons often include correctness of the algorithms [7] [15] [14] [6] and FPS comparison [11]. Furthermore, many works try to optimize feature detection algorithms using specialized hardware environments such as FPGA [8] [13] [10]. Other studies also compare image processing algorithms using different hardware units such as CPU, GPU and FPGA [1]. Furthermore, Johny et al. [9] presented a method which investigates the resource usage in the Harris Corner detection algorithm.

In this study, we instead try to focus on a broad scope by identifying characteristics which are important to investigate when using feature detection algorithms on limited hardware. Furthermore the ultimate goal of this work is to be able to determine how the quality of service can be affected of feature

algorithm using a system with specific characteristics. In this paper we investigate the mechanics of the FAST algorithm by evaluating the code as well as analyzing the algorithms effects on the memory of the system in which it is running.

7.3 Method

In this work, we evaluate three characteristics including memory resources, opportunities for parallelism and resource usage with respect to the FAST algorithm. For a richer proof of concept, we used the 8-core Freescale P4080 with a 2017-03 NXP fsl-core linux distribution for evaluating the parallel issues due to the high multi-core capabilities while we used an Intel Core i-3570k using Ubuntu 16.04, kernel version 4.4 for investigating PMU related topics. We used a 512x512 bitmap version of the Lena image as test data, due to its frequent usage in other corner detection research papers. We executed all FAST tests on a 512x512 pixel bitmap, depicted in figure 7.2[1].

Figure 7.2: Input data for FAST

7.3.1 Opportunities for Parallelism

A traditional way of executing algorithms in parallel is using the fork-join model, which partitions a workload into smaller workloads and executes these smaller workloads on different cores. When working with image processing, this is a simple and intuitive way of increasing the performance, since a pixel matrix often can be divided into sub-matrices by the amount of cores used. There is no need for synchronization within the algorithm steps because the FAST algorithm does not use global shared variables. The FAST algorithm with a fork-join using an 8 core machine is depicted in Figure 7.3. The optimal performance of an algorithm running in parallel is defined by Execution time

[1]The figure was selected because it is one of the standard test images in the image processing community

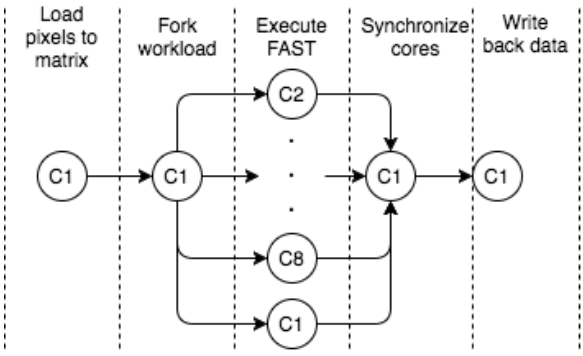


Figure 7.3: FAST 8 core execution model

using one core divided by amount of cores being used. The execution model in Figure 7.3 depicts a straight-forward fork-join execution model which is possible for the fast algorithm. This leads to a near-optimal execution speed-up when using multiple cores. For proof of concept, we took measurements of FAST running on 1 to 7 cores using an 8 core Freescale P4080 machine and executing a test 100 times on the same picture. Table 7.1 shows the percentage deviation from the optimal performance when executing on multiple cores using I_t thresholds of 10%, 20%, 30% and 40%.

Cores	$I_t = 10\%$	$I_t = 20\%$	$I_t = 30\%$	$I_t = 40\%$
2 cores	4,41%	3,05%	1,98%	1,26%
3 cores	4,69%	3,21%	1,79%	0,98%
4 cores	5,10%	3,61%	2,21%	1,06%
5 cores	5,09%	2,93%	1,59%	0,92%
6 cores	5,86%	4,34%	2,80%	1,69%
7 cores	4,51%	2,79%	2,58%	1,95%

Table 7.1: Fork-join measurements of FAST

As shown in Table 7.1, we see only a small percentage deviation to the optimal execution time even when the threshold is set to 10% which can be considered a very sensitive threshold. Thus we can conclude that FAST is an algorithm very well suited for parallelism. Executing tests using 5 cores managed to decrease the deviation percentage compared, even though the trend was an increasing percentage. This can be an indication of uneven workload between the different cores.

7.3.2 Resource Utilization

In our investigation, we have focused on two memory-related issues. The first issue relates to the code size of the algorithm itself. A long and complex execution flow causes several performance-related side-effects, such as instruction cache misses and branch mispredictions. The second memory-related issue relates to the data processed by the algorithm. A memory-bound algorithm has a large working set and triggers a high degree of data cache misses, Data-TLB reloads and memory bus contention which can cause a decrease in performance. If two algorithms are memory-bound, it may not be sufficient to distribute them on different CPU cores because they often share HW resources. We have used a system-level metric (SLM) as performance indicator that describes the number of pixels traversed per time unit. Simultaneously, we monitor Low-Level Metrics (LLM) describing the memory subsystem usage. We use the Pearson coefficient [5] to denote how well SLM correlates with each LLM whereas 0 mean no correlation at all and 1 full correlation. We tested the correlation by executing a test-suite that fetches SLM and LLM at 100Hz. We ran this test using a fork-join model with four cores, where core 0 was set up as a synchronization core and core 1-3 as workload cores.

Core : I threshold	L1D miss	L1I miss	L2 miss	DTLB miss
Core 1 : I_t=10%	0.195	0.114	0.191	0.233
Core 2 : I_t=10%	0.103	0.067	0.084	0.21
Core 3 : I_t=10%	0.056	0.395	0.029	0.133
Core 1 : I_t=20%	0.004	0.365	0.145	0.024
Core 2 : I_t=20%	0.206	0.172	0.197	0.24
Core 3 : I_t=20%	0.076	0.395	0.427	0.4
Core 1 : I_t=30%	0.073	0.013	0.07	0.234
Core 2 : I_t=30%	0.198	0.187	0.078	0.169
Core 3 : I_t=30%	0.208	0.083	0.204	0.131
Core 1 : I_t=40%	0.012	0.035	0.119	0.083
Core 2 : I_t=40%	0.21	0.187	0.246	0.172
Core 3 : I_t=40%	0.431	0.395	0.427	0.4

Table 7.2: Fork-join measurements of FAST

Table 7.2 show Pearson coefficient obtained from the workload cores during the tests of correlation between SLM and the LLM. Our measurements indicate that the performance correlates very little with the memories measured in this test. This occurrence may be due to the fact that FAST has few memory operations and instead uses many branch operations. The current pattern however suggest that core 3 correlates best with the LLM we chose. The

correlation may be an effect of how the picture is divided. With the current division of the picture, core 3 would detect the least corners, and would use the least branches and would therefore correlate better with the memory.

7.4 Resource usage challenges

Many resources affect the performance of an algorithm apart from the earlier mentioned ones. Different parallel paradigms are useful depending on the algorithm, for example, the fork-join model. Due to the many if statements of FAST, it is very unlikely that a forked algorithm will execute with at the same speed on different cores. If one core is overwhelmed with corner detections it can lead to one core executing the algorithm slower than the other cores, leaving the other cores underutilized, Figure 7.4 illustrates such behavior.

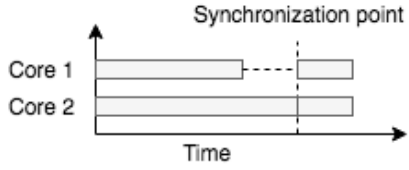


Figure 7.4: Core idle issue due to synchronization

Altering the threshold values of FAST can dramatically change the result of found corners in an image. To test the resource utilization of FAST, we executed 1000 tests on a single picture, measuring the execution time of each individual core, whereas core 1-7 were used as work-set cores and core 0 as housekeeping/synchronization thread. Figure 7.5 depicts the mean execution time of the 1000 tests for each individual work-set core using I_t values of 10%, 20%, 30% and 40%. Each core also had different amount of corner detection, Table 7.3 shows the amount of corner points detected in each individual core.

Core	10%	20 %	30 %	40 %
1	2018	761	373	206
2	2460	873	432	232
3	2301	842	377	203
4	2391	796	390	178
5	1938	588	432	102
6	1284	253	99	39
7	546	99	42	11

Table 7.3: Corners detected per core

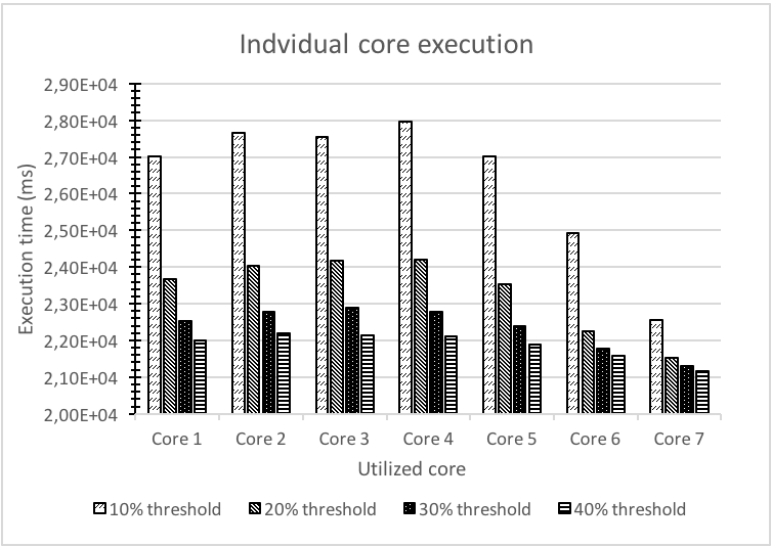


Figure 7.5: Execution time per core with different I_t values using FAST

From the measurements conducted in this section, we can conclude that the inter-core synchronization time correlates with the amount of corners detected. This mean executing FAST on images which does not have corners evenly distributed, may lead to a utilization loss when executing FAST on multiple cores.

7.5 Conclusion

In this study, we have evaluated an implementation of the FAST algorithm regarding aspects of resource utilization, opportunities for parallelism and memory consumption using different thresholds. Our results show that FAST is a relatively simple algorithm with strong opportunities for parallelism. We see a challenge with choosing threshold values which has to be investigated further. If choosing a high threshold, there is a risk of not detecting important corners. If however choosing a low threshold, there is a possibility of loading the system inefficiently. It could however be possible to schedule FAST more efficiently by programming already finished cores to help the non finished cores finish. This approach could reduce the synchronization performance loss. By monitoring the PMU counters, we also conclude that FAST does not suffer much from misses in the memory.

Future work includes conducting a deeper study of the execution characteristics, investigating both execution behavior as well as memory behavior of more well-known feature and corner detection algorithms such as Harris, SURF and SIFT. By carrying out such a study, it is possible to understand how different feature detection algorithms should be partitioned and scheduled together. Ultimately, this knowledge can lead to a more time-predictable and dependable corner detection suite.

Bibliography

[1] S. Asano, T. Maruyama, and Y. Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 126–131. IEEE, 2009.

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

[3] S. Eyerman and P. Michaud. Defining metrics for multicore throughput on multiprogrammed workloads. Technical report, Ghent University - Team ALF, 2013.

[4] M. Jägemar, A. Ermedahl, and S. Eldh. Decision support for OS process scheduling based on HW-, OS- and system-level performance counters, 2016.

[5] 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.

[6] L. Juan and O. Gwun. A comparison of sift, pca-sift and surf. *International Journal of Image Processing (IJIP)*, 3(4):143–152, 2009.

[7] R. Maini and H. Aggarwal. Study and comparison of various image edge detection techniques. *International journal of image processing (IJIP)*, 3(1):1–11, 2009.

[8] R. Mehra and R. Verma. Area efficient fpga implementation of sobel edge detector for image processing applications. *International Journal of Computer Applications*, 56(16), 2012.

[9] J. Paul, W. Stechele, M. Kröhnert, T. Asfour, B. Oechslein, C. Erhardt, J. Schedel, D. Lohmann, and W. Schröder-Preikschat. Resource-aware harris corner detection based on adaptive pruning. In *International Conference on Architecture of Computing Systems*, pages 1–12. Springer, 2014.

[10] P. R. Possa, S. A. Mahmoudi, N. Harb, C. Valderrama, and P. Manneback. A multi-resolution fpga-based architecture for real-time edge and corner detection. *IEEE Transactions on Computers*, 63(10):2376–2388, 2014.

[11] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[12] E. Rosten and T. Drummond. Fusing points and lines for high performance tracking. In *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, volume 2, pages 1508–1515. IEEE, 2005.

[13] V. Sanduja and R. Patial. Sobel edge detection using parallel architecture based on fpga. *International Journal of Applied Information Systems*, 3(4):20–24, 2012.

[14] G. Shrivakshan, C. Chandrasekar, et al. A comparison of various edge detection techniques used in image processing. *IJCSI International Journal of Computer Science Issues*, 9(5):272–276, 2012.

[15] K. Zeng, N. Wu, L. Wang, and K. K. Yen. Local visual feature detection and description for non-rigid 3d objects. *Advances in Image and Video Processing*, 4(2):01, 2016.

Chapter 8

Paper B
Measurement-based evaluation of data-parallelism for OpenCV feature-detection algorithms

J. Danielsson, M. Jägemar, M. Behnam, M. Sjödin, and T. Seceleanu. In 42^{nd} *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.

8.1 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 8.2 give background information related to feature detection algorithms and their resource usage. Detailed information on our implementation is given in Section 8.3 and the experiments in Section 8.4. We conclude the paper by summarizing our conclusions in Section 8.5.

8.2 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.

8.2.1 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 8.1.

Table 8.1: 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

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.

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. 8.1. 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.

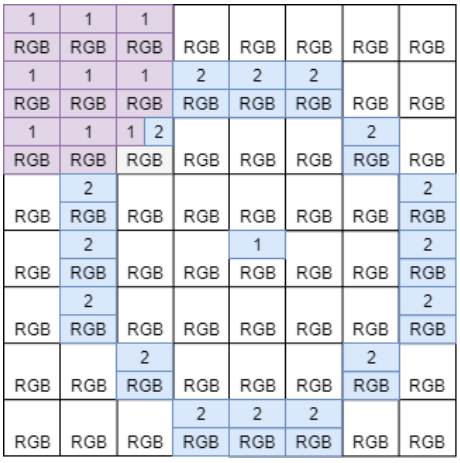


Figure 8.1: Example of FAST and Harris.

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.

8.2.2 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. 8.2 illustrates an example of the fork-join model utilizing 4 cores.

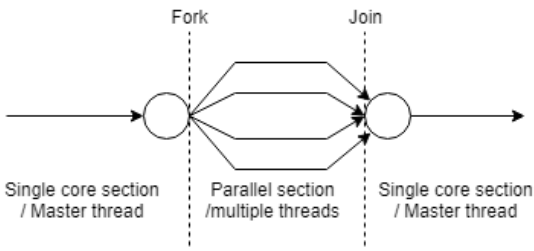


Figure 8.2: The fork-join model for parallelization of algorithms.

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.

8.2.3 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.

8.3 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.

8.3.1 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. 8.3.

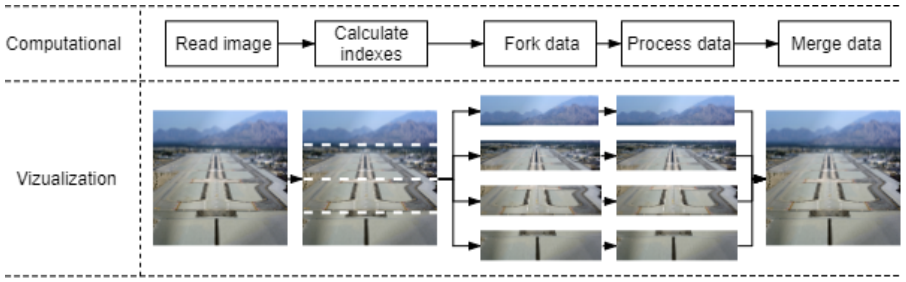


Figure 8.3: The image data is partitioned to support the fork-join model.

Fig. 8.3 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 (8.1) 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 8.3 and 8.4, 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.

$$Work_x = ImageSize_x \tag{8.1}$$

$$if(aff) = 0, \quad UpperBound = 0 \tag{8.2}$$

$$if(aff) > 0, \quad UpperBound = \frac{ImageSize_y}{aff + 1} \tag{8.3}$$

$$LowerBound = \frac{ImageSize_y}{aff + 2} \tag{8.4}$$

8.3.2 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.

8.3.2.1 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. 8.4.

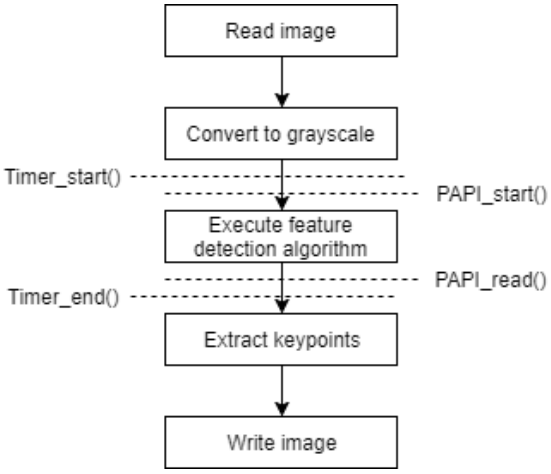


Figure 8.4: The algorithm performance measurement sequence.

8.3.2.2 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. 8.4.

8.4 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 8.2. 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 8.2: 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 8.3.

Table 8.3: 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. 8.5 and follow the specifications presented listed in Table 8.3.

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

Figure 8.5: 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.

8.4.1 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 (8.5), 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.

$$S = \frac{t_0}{\{max(t_i) : 0 \leq i < n\}} \tag{8.5}$$

Fig. 8.6 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. 8.5 a, the second set to Fig. 8.5 b, and the third set to Fig. 8.5 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.

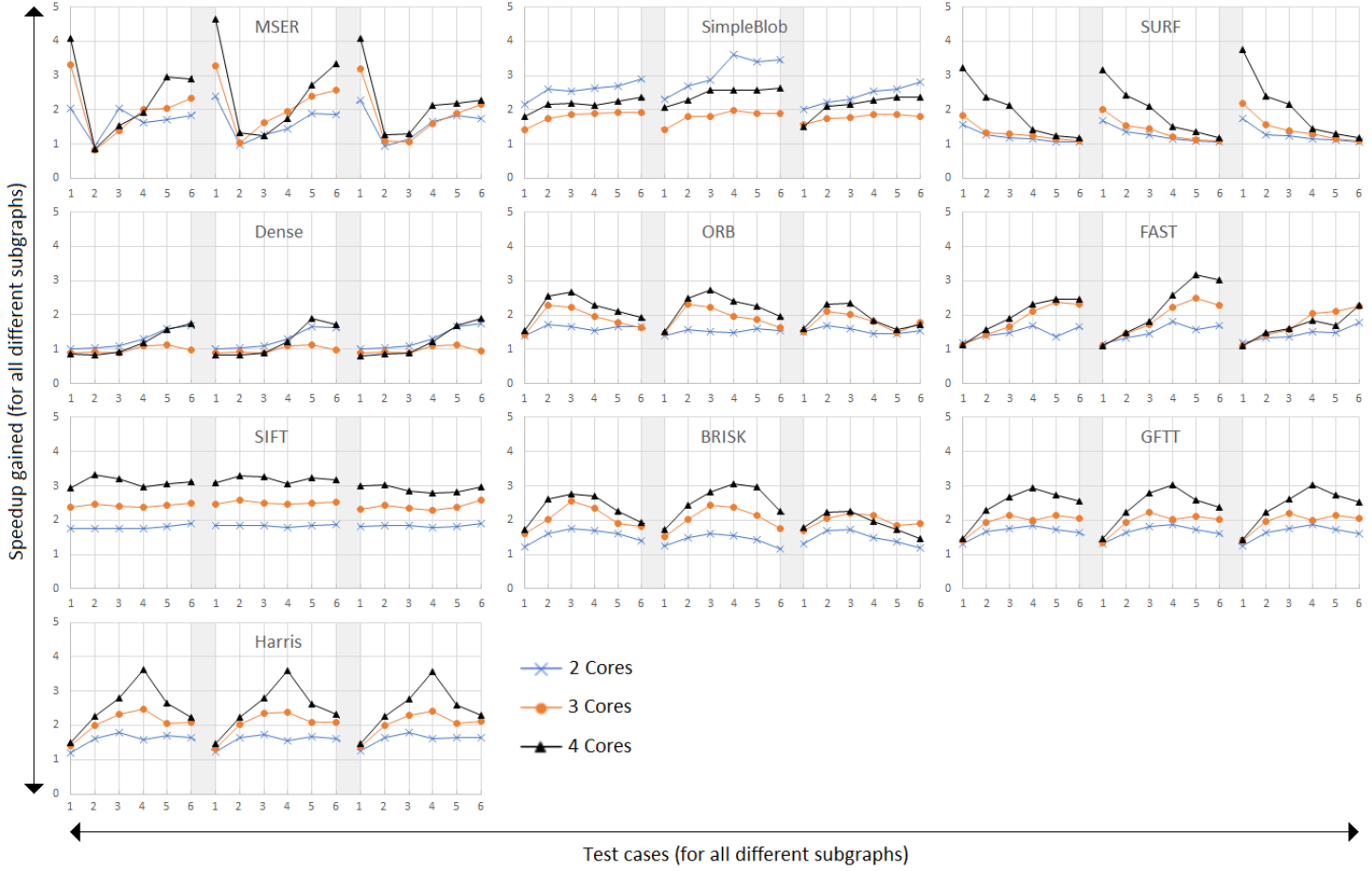


Figure 8.6: Feature detection algorithm speed-up factors for various test-cases when running a multi-core test system.

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.

8.4.2 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 8.4 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 8.4: Detected key points.

8.4.3 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. 8.7. 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.

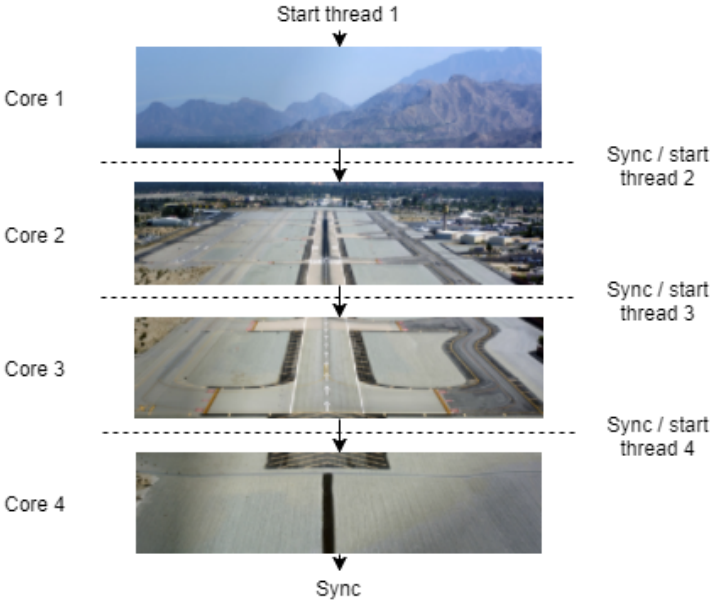


Figure 8.7: Sequential version of the test program.

We call the difference between our sequential execution and our parallel execution ΔT, which is calculated according to equation (8.6) 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.

$$\Delta T = \{max(tp_i) : 0 \leq i < n\} - \{max(ts_i) : 0 \leq i < n\} \quad (8.6)$$

ΔT allows us to quantify how much of the program execution time is affected by utilizing a multi-core architecture. Fig. 8.8 illustrates the ΔT per core per image.

Fig. 8.8 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.

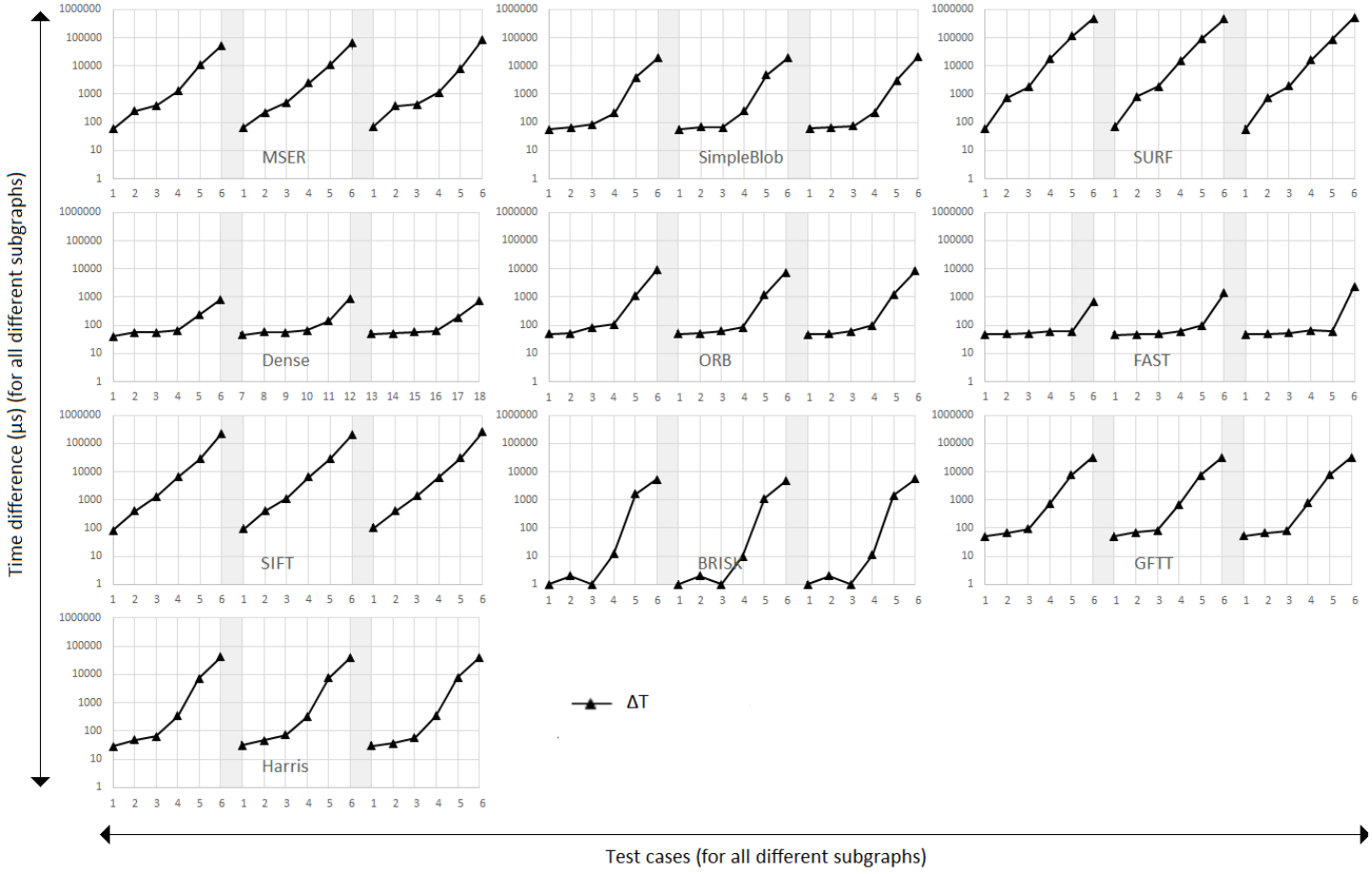


Figure 8.8: Differences in execution time using parallel and sequential approach.

8.4.4 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. 8.9 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 (8.7).

$$\Delta C = \frac{ParallelMisses(C_{1..4})}{SequentialMisses(C_{1..4})} \tag{8.7}$$

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.

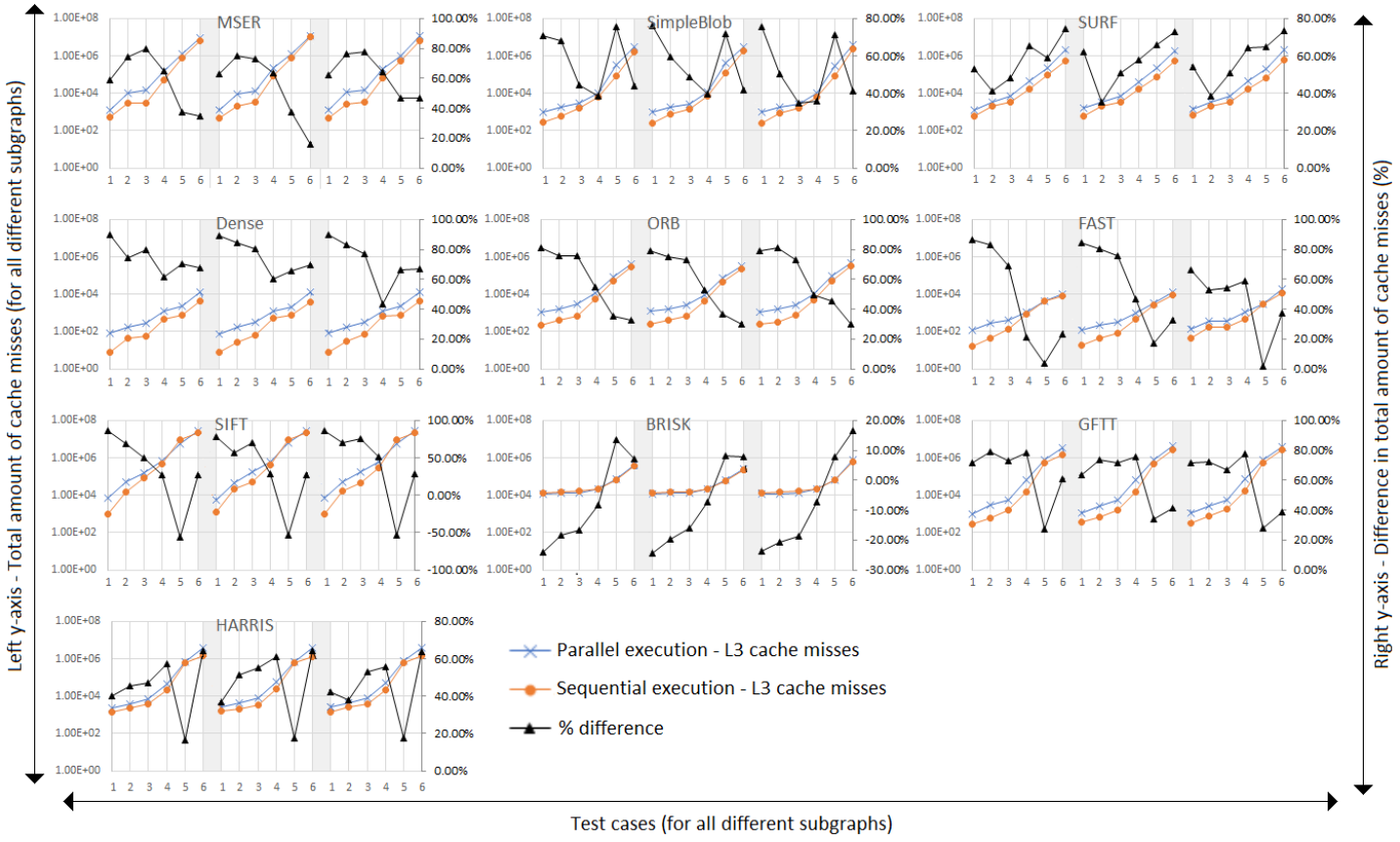


Figure 8.9: L3 misses using parallel and sequential version.

Compared to the other algorithms, FAST has a low L_3-cache usage, see Fig. 8.9, 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. 8.5, 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. 8.8 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. 8.6, 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. 8.9 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.

8.5 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.

8.5.1 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 9

Paper C
Testing Performance-Isolation in Multi-Core Systems

J. Danielsson, T. Seceleanu, M. Jägemar, M. Behnam and M. Sjödin. In 43^{rd} *Computer Society Signature Conference on Computers, Software and Applications (COMPSAC)*. 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.

9.1 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 Jailhouse 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.

9.2 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.

9.2.1 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. 9.1 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).

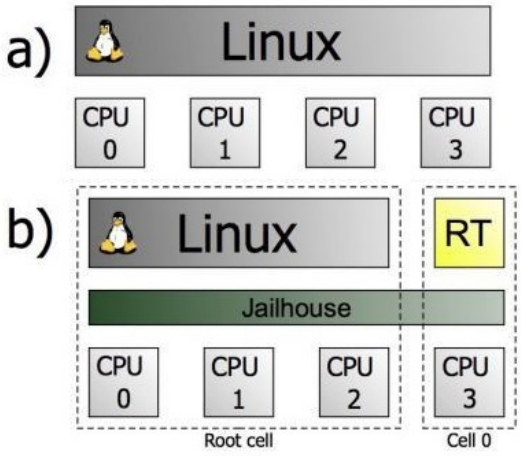


Figure 9.1: a) Usual Linux deployment. b) Linux with Jailhouse configuration [15]

9.3 Shared resource contention

We describe the performance degradation of a process in Equation 9.1, where performance is equal to the execution time of an application.

$$I = \frac{P}{C} - 1 \tag{9.1}$$

We denote I as the *isolation coefficient*, representing the resulting slowdown 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 9.1.

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 9.1: Hardware specifications Xilinx Zynq UltraScale+ MPSoC

9.3.1 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.

9.3.2 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.

9.3.3 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.

9.4 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 9.1) 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.

9.4.1 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. 9.2 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 9.2.

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 9.2: *I* coefficient in CPU contention test (percentage)

Fig. 9.2 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 9.2 and using Equation 9.1, 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 asso-

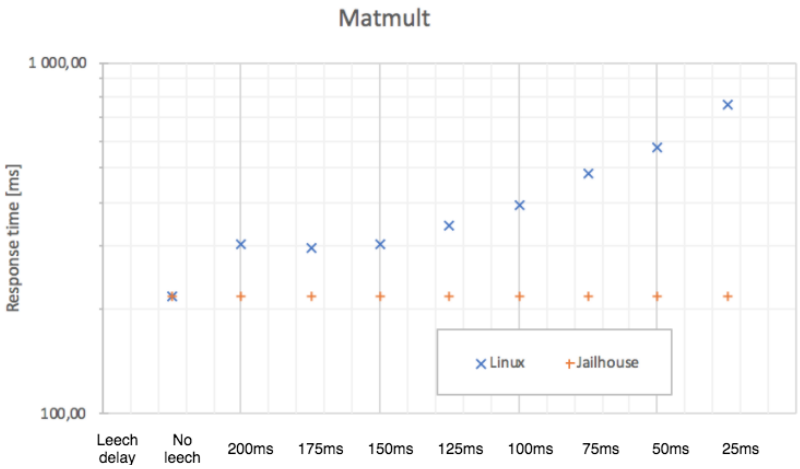


Figure 9.2: CPU isolation test

ciated 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.

9.4.2 L_2-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. 9.3, and the results of running it within a Jailhouse Linux cell are shown in Fig. 9.4. 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 9.3 lists the calculated isolation for the matrix multiplication when co-run with the Tinymembench load.

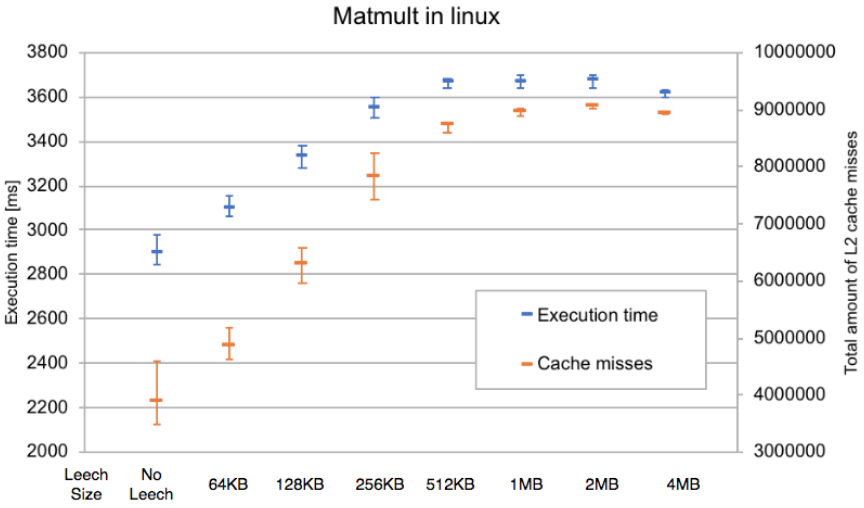


Figure 9.3: Linux L_2-cache isolation test

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

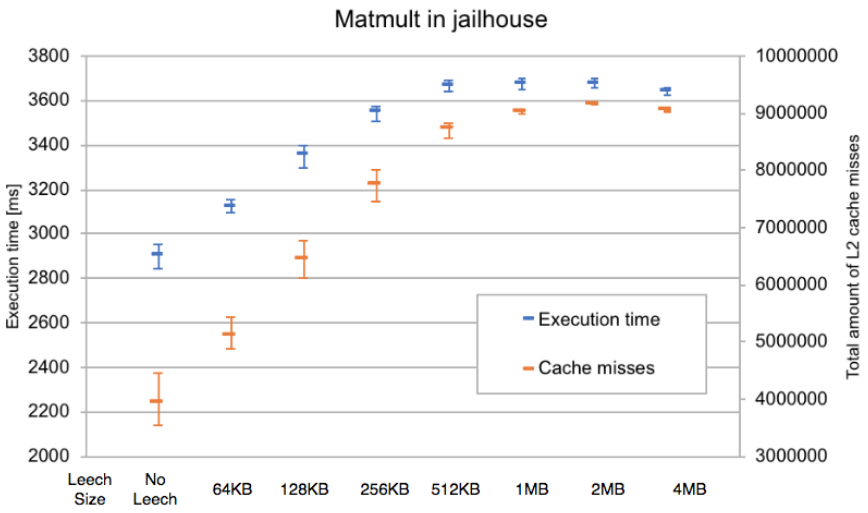


Figure 9.4: Jailhouse L_2-cache isolation test

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 9.3, 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 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.

9.4.3 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 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. 9.5 depicts the results of the regular Linux matrix multiplication execution, and Fig. 9.6 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. We list the calculated isolation coefficient for the matrix multiplication using regular Linux and Jailhouse in Table 9.4.

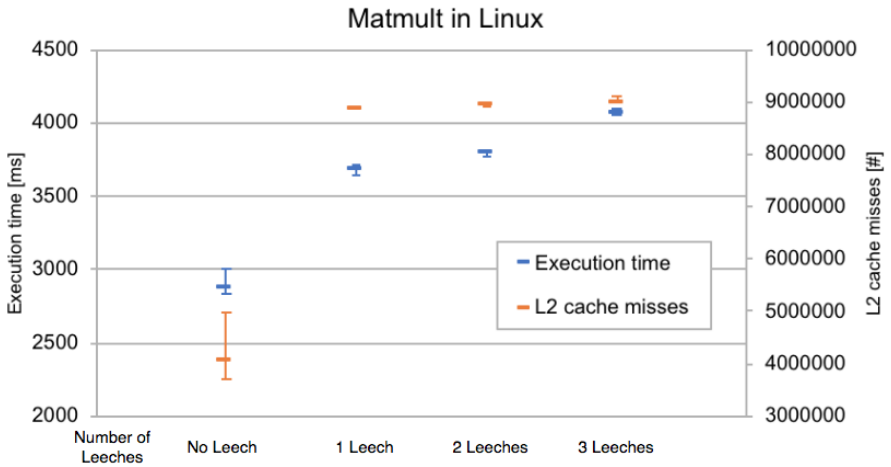


Figure 9.5: Linux memory bus isolation test

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

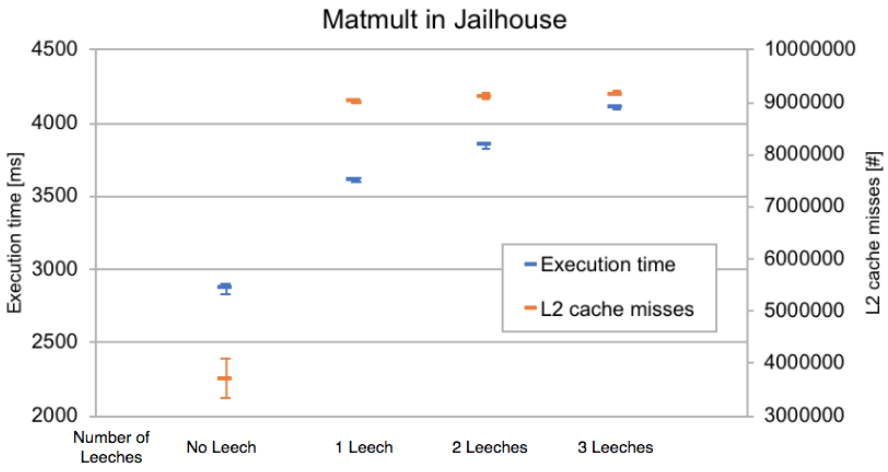


Figure 9.6: Jailhouse memory bus isolation test

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 9.4: *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%

9.5 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 page coloring mechanisms using our methodology is also relevant future work in the Jailhouse case.

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 10

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

J. Danielsson, M. Jägemar, M. Behnam, T. Seceleanu and M. 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.

10.1 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 run-time. 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 10.2 and describe the LLC partition controller we have implemented in Section 10.3. 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 10.4. Section 10.5 describe work related to ours and we conclude the paper in Section 10.6.

10.2 Background

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

10.2.1 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 10.1.

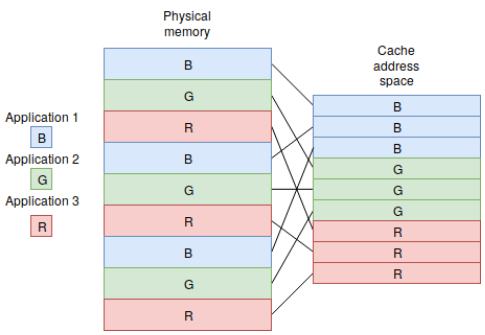


Figure 10.1: Cache coloring

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 10.1 [13].

$$Nr.\ of\ Colors = \frac{Cache\_size}{Cache\_ways * page\_size} \tag{10.1}$$

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.

10.2.2 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. 10.2 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.

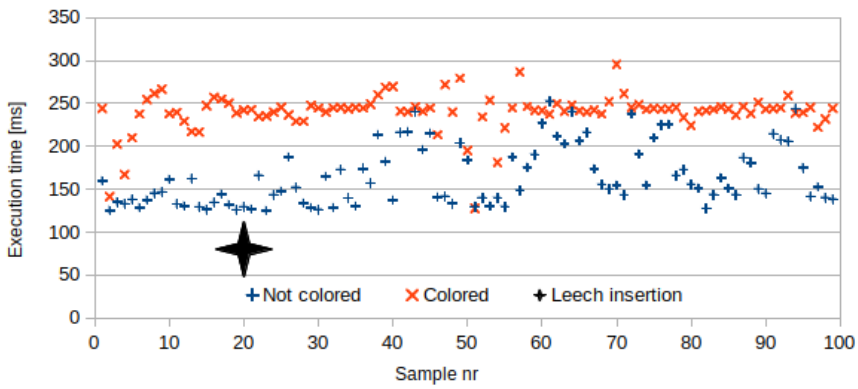


Figure 10.2: Matrix multiplication - isolation example

Fig. 10.2 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. 10.3 shows the performance difference of the same matrix multiplication using various amount of LLC partition size.

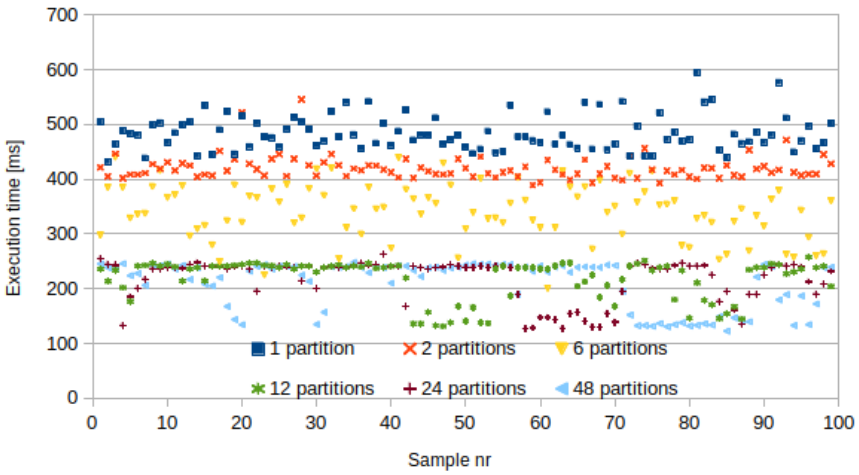


Figure 10.3: Matrix multiplication using different cache partition sizes

Assigning only 1 LLC partition to the matrix multiplication significantly reduces the performance, compared to the execution in Fig. 10.2, 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. 10.3 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.

10.3 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. 10.4.

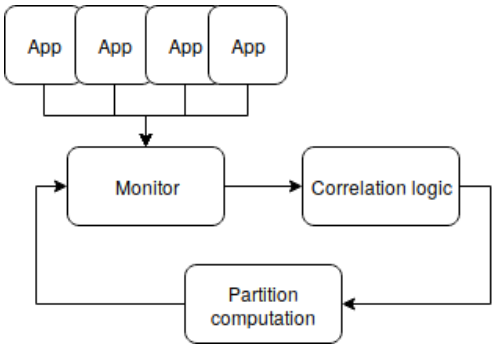


Figure 10.4: LLC-PC

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 10.2, 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.

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2(y_i - \overline{y})^2}} \tag{10.2}$$

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]. The correlation significance may, however, vary depending on the data set.

10.3.1 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. 10.5.

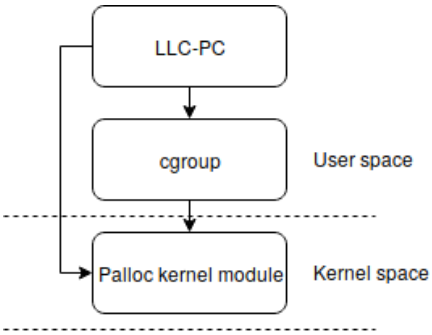


Figure 10.5: System connections

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. 10.4, consists of three parts. The monitor part, the correlation part and the partition computation part. The controller implementation is described in Algorithm 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 1: LLC-PC pseudocode

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.

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.

10.4 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 10.1.

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 10.1: Hardware specifications Intel® Core™ i578850H

10.4.1 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. 10.6 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.

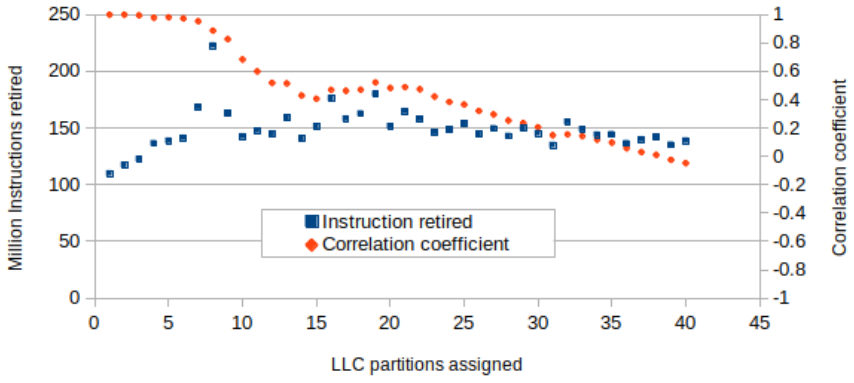


Figure 10.6: 512x512 matrix multiplication execution

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. 10.7 show as an execution of the SIFT algorithm run on a 4MB image with different cache partition sizes from 1 to 40.

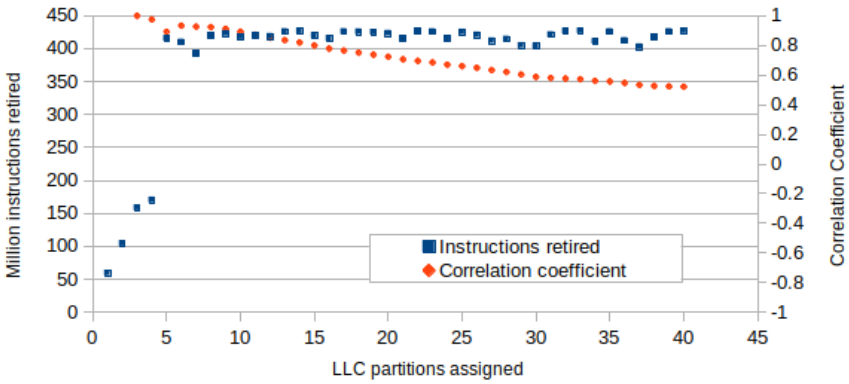


Figure 10.7: 4 MB SIFT execution

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. 10.8 depicts the correlation coefficients from the random calculation test.

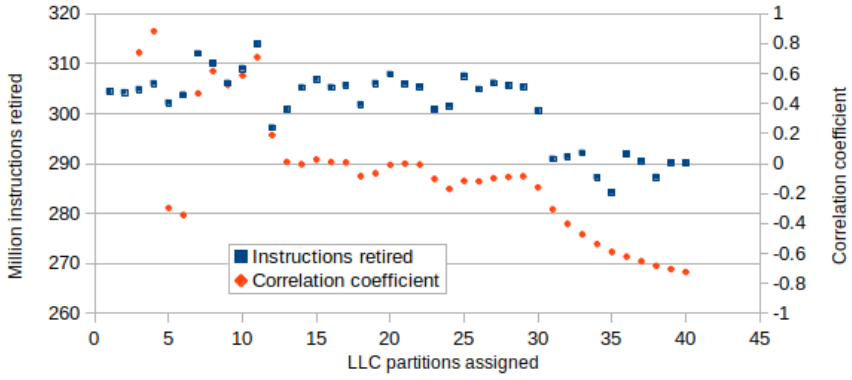


Figure 10.8: Random calculation execution

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.

10.4.2 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.

10.4.3 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 10.2.

Property	Value
Available LLC partitions	128
Correlation window size	5
Correlation threshold	0.8
Control loop sleep	50ms

Table 10.2: 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. 10.9 and Fig. 10.10 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.

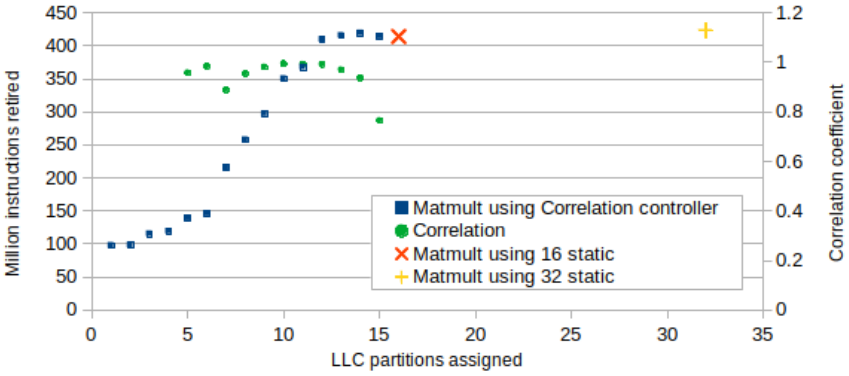


Figure 10.9: Comparison of 756x756 matrix multiplication executions

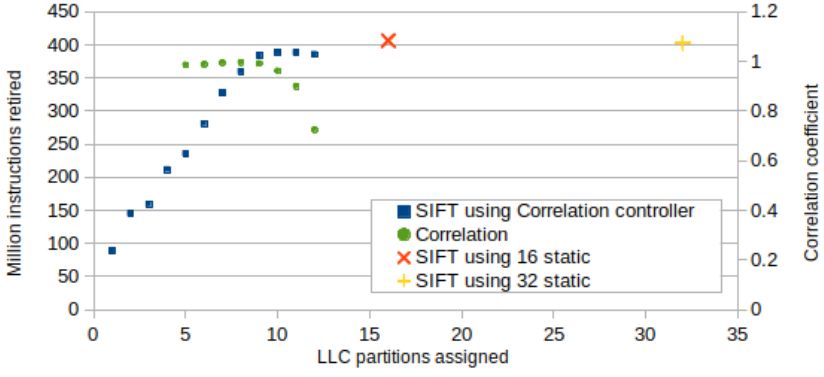


Figure 10.10: Comparison of 8MB SIFT executions.

Fig. 10.9 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. 10.10 show SIFT operating within the LLC-PC, with a final assignment of 13 LLC partitions at which point the correlation value has dropped from 0.89 to 0.72. The correlation-based methodology almost reaches the same performance achieved by the static LLC partition allocations.

Table 10.3 and Table 10.4 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 10.3: 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 10.4: SIFT tests

Table 10.3 shows the benefit of LLC-PC, especially using the smallest matrix multiplication size of 256x256, which saturates at a partition size of 7. Increasing 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 10.4 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.

10.5 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.

10.6 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.

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] ARM. Cortex-a53. `https://developer.arm.com/ip-products/processors/cortex-a/cortex-a53`. Accessed: 2019-11-04.

[4] S. Asano, T. Maruyama, and Y. Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 126–131. IEEE, 2009.

[5] 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.

[6] 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.

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

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] A. Cherubini, F. Spindler, and F. Chaumette. Autonomous visual navigation and laser-based moving obstacle avoidance. *IEEE Transactions on Intelligent Transportation Systems*, 15(5):2101–2110, 2014.

[15] J. Cohen. *Statistical power analysis for the behavioral sciences*. Routledge, 2013.

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

[17] J. Danielsson, M. Ashjaei, M. Behnam, T. Sorensen, M. Sjodin, and T. Nolte. Performance evaluation of network convergence time measurement techniques. In *2017* 22^{nd} *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–7. IEEE, 2017.

[18] J. Danielsson, N. Tsog, and A. Kunnappilly. A systematic mapping study on real-time cloud services. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 245–251. IEEE, 2018.

[19] 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.

[20] 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.

[21] 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.

[22] R. O. Duda and P. E. Hart. Use of the hough transformation to detect lines and curves in pictures. Technical report, Sri International Menlo Park Ca Artificial Intelligence Center, 1971.

[23] 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.

[24] 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.

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

[26] 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.

[27] S. Eyerman and P. Michaud. Defining metrics for multicore throughput on multiprogrammed workloads. Technical report, Ghent University - Team ALF, 2013.

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

[29] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino. An edf scheduling class for the linux kernel. In *Proceedings of the 11th Real-Time Linux Workshop*, pages 1–8. Citeseer, 2009.

[30] 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.

[31] Z. Fleischman and C. Sullivan. Optically assisted landing of autonomous unmanned aircraft, May 5 2016. US Patent App. 14/631,520.

[32] X. Fu, K. Kabir, and X. Wang. Cache-aware utilization control for energy efficiency in multi-core real-time systems. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 102–111. IEEE, 2011.

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

[34] G. Gracioli and A. A. Fröhlich. An experimental evaluation of the cache partitioning impact on multicore real-time schedulers. In *2013 IEEE* 19^{th} *International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 72–81. IEEE, 2013.

[35] 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.

[36] 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.

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

[38] 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.

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

[40] M. Jägemar, A. Ermedahl, and S. Eldh. Decision support for OS process scheduling based on HW-, OS- and system-level performance counters, 2016.

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

[42] 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.

[43] K. Jian, Z. X. Dong, N. Wen-wu, Z. Jun-wei, H. Xiao-ming, Z. Jiangang, and X. Lu. A performance isolation algorithm for shared virtualization storage system. In *2009 IEEE International Conference on Networking, Architecture, and Storage*, pages 35–42. IEEE, 2009.

[44] L. Juan and O. Gwun. A comparison of sift, pca-sift and surf. *International Journal of Image Processing (IJIP)*, 3(4):143–152, 2009.

[45] 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.

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

[47] 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.

[48] 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.

[49] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *[1989] Proceedings. Real-Time Systems Symposium*, pages 166–171. IEEE, 1989.

[50] 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.

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

[52] 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.

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

[54] R. Maini and H. Aggarwal. Study and comparison of various image edge detection techniques. *International journal of image processing (IJIP)*, 3(1):1–11, 2009.

[55] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), 2002.

[56] 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.

[57] 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.

[58] 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.

[59] R. Mehra and R. Verma. Area efficient fpga implementation of sobel edge detector for image processing applications. *International Journal of Computer Applications*, 56(16), 2012.

[60] 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.

[61] 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.

[62] L. Nyman and M. Laakso. Notes on the history of fork and join. *IEEE Annals of the History of Computing*, 38(3):84–87, 2016.

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

[64] C. S. Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

[65] 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.

[66] 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.

[67] J. Paul, W. Stechele, M. Kröhnert, T. Asfour, B. Oechslein, C. Erhardt, J. Schedel, D. Lohmann, and W. Schröder-Preikschat. Resource-aware harris corner detection based on adaptive pruning. In *International Conference on Architecture of Computing Systems*, pages 1–12. Springer, 2014.

[68] 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.

[69] P. R. Possa, S. A. Mahmoudi, N. Harb, C. Valderrama, and P. Manneback. A multi-resolution fpga-based architecture for real-time edge and corner detection. *IEEE Transactions on Computers*, 63(10):2376–2388, 2014.

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

[71] H. Raj, R. Nathuji, A. Singh, and P. England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 77–84. ACM, 2009.

[72] 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.

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

[74] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[75] E. Rosten and T. Drummond. Fusing points and lines for high performance tracking. In *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, volume 2, pages 1508–1515. IEEE, 2005.

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

[77] 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.

[78] 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.

[79] V. Sanduja and R. Patial. Sobel edge detection using parallel architecture based on fpga. *International Journal of Applied Information Systems*, 3(4):20–24, 2012.

[80] 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.

[81] 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.

[82] 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.

[83] G. Shrivakshan, C. Chandrasekar, et al. A comparison of various edge detection techniques used in image processing. *IJCSI International Journal of Computer Science Issues*, 9(5):272–276, 2012.

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

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

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

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

[88] G. Somani and S. Chaudhary. Application performance isolation in virtualization. In *2009 IEEE International Conference on Cloud Computing*, pages 41–48. IEEE, 2009.

[89] 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.

[90] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

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

[92] 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.

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

[94] 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.

[95] W. Wang, P. Mishra, and S. Ranka. Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems. In *2011* 48^{th} *ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 948–953. IEEE, 2011.

[96] 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.

[97] 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.

[98] 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.

[99] 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.

[100] 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.

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

[102] K. Zeng, N. Wu, L. Wang, and K. K. Yen. Local visual feature detection and description for non-rigid 3d objects. *Advances in Image and Video Processing*, 4(2):01, 2016.

[103] 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.

[104] 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.