INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

Evaluating Sector Caches in High-Performance Computing

Sergej-Alexander Breiter

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

Evaluating Sector Caches in High-Performance Computing

Sergej-Alexander Breiter

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Dr. Karl Fürlinger
	Dr. Josef Weidendorfer
Abgabetermin:	31. Januar 2022

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 31. Januar 2022

. . . . . .
(Unterschrift des Kandidaten)

Abstract

The sector cache is a hardware cache partitioning mechanism of the A64FX processor. The A64FX is used in the Fugaku system – currently the fastest supercomputer on the TOP500 list (as of November 2021). It allows application software to dynamically partition a cache and can reduce the occurrence of cache misses by protecting data with high temporal locality from eviction. Many cache partitioning techniques focus on optimizing the cache behavior of shared caches when multiple co-scheduled processes run on the same processor by assigning them to partitions. In contrast, the sector cache aims to improve the cache behavior of a single application by assigning its data to partitions. However, even the hardware manufacturer of the A64FX states that it is difficult to use the sector cache in a meaningful way. Therefore, a profiling tool based on the reuse distance metric is being developed using Intel's PIN binary instrumentation framework. The profiling tool tries to provide programmers with opportunities where the sector cache can be usefully applied without requiring the programmer to have detailed knowledge of a program's data locality. Using the parallel NAS benchmarks as an example, it is shown that the tool can indeed help programmers to find code regions where the sector cache can improve cache behaviour. In addition, it is shown that sector cache can significantly improve performance in certain typical situations and these as well as the sector cache behavior of the A64FX are explored and analyzed.

Contents

1 Introduction

Computing is getting increasingly important in many fields of science such as machine learning, climate modeling, weather forecasting or astrophysics [YTY21]. Today, also other fields like biology profit from high-performance computing capabilities. A recent example is the *fight against COVID-19* [LDDB+21, HP20], where compute resources of supercomputers are dedicated to simulations that help understanding the structures and mechanisms of the virus in order to develop vaccines and therapeutic treatments. In the times of big data, ever faster and bigger machines have to be constructed to handle the vast amounts of data and huge computational tasks of scientific workloads in acceptable time [OFK12]. Supercomputing is on the verge of achieving exascale floating point performance with the Fugaku [SIT+20] system being currently the leader of the fastest supercomputers on the TOP500 list [DMS+97] as of November 2021.

Huge amounts of data in real-world applications often result in memory-intensive applications that put high demands on the memory system, and the memory transfer rate can become the application performance bottleneck. This can be tackled either by increasing the memory system performance or by reducing the data volume transferred. Caches – fast memory with low capacity – are placed between the main memory and the Arithmetic Logical Units (ALUs) of the Central Processing Unit (CPU) to hold data and make it accessible to the CPU orders of magnitude faster compared to the access time of the main memory. But because of the smaller size, typically not all of the data required by an application can be stored in the cache at the same time. Loading data from memory is expensive, thus data that is frequently reused should be kept in cache and not replaced with data that is only used once. Optimizing the use of the cache system is at the center of many performance optimization techniques. The goal of these techniques is either to reduce the memory access penalty or the transferred data volume and can be achieved, for example, by exploiting localities of the data access pattern in applications.

The CPU used in the Fugaku system, the A64FX CPU, comes with a hardware feature called *sector cache*. It enables an adaptive partitioning of the cache area to make better use of temporal localities within an application. Reusable data can be placed in a cache partition separate from other data and thus be protected from being driven out of the cache in order to avoid reloading reusable data from memory. However, it is not always clear when the sector cache can reduce the amount of data traffic and how much performance can be gained. In fact, even the manufacturer of the A64FX CPU states that it is hard to make good use of the sector cache and that it can significantly degrade performance if not used with caution [Fuj21b].

A central question of this thesis is how application designers can spot opportunities for the use of the sector cache feature in their software. The developed approach is based on a binary instrumentation tool that analyzes the data access pattern of an application to detect parts within that application where the sector cache can improve data locality. Another question is how much performance potentially can be gained by the use of the sector cache on the A64FX CPU under idealized conditions.

The next chapter of this thesis provides the background knowledge on the topic. The general approach is explained in the third chapter. The implementation is described in the fourth chapter and the fifth chapter contains the evaluation of the sector cache performance gains and the chosen approach on the A64FX CPU. Finally, in the last chapter, the results of the thesis are summarized and possibilities for future work are pointed out.

2 Background

The background chapter provides the necessary background knowledge on caches and cache performance. Cache optimizations related to the sector cache are discussed before the employed hardware of the A64FX CPU is commented. The sector cache is explained next. Reuse distance analysis, which is the theoretical basis of the developed tool, is discussed before the related work is presented at the end of this chapter.

2.1 Caches

Computer programs consist of two parts: instructions and data. Ideally, hardware would provide unlimited amounts of fast memory to feed the instructions and data to the ALUs of a CPU in order to keep the CPU busy all the time. But fast memory is expensive and the increase in CPU speeds as well as the addition of multiple cores in modern CPUs leads to higher average memory request rates that can not be matched by the technical progress in the increase of memory speeds (*Dynamic Random-Access Memory (DRAM) gap*). Thus, the memory subsystem becomes the performance bottleneck.

2.1.1 Memory Hierarchy

A well-known solution to this dilemma is to organize the memory in multiple hierarchical levels – each level smaller and faster than the next lower level. Figure 2.1 shows an example of the memory hierarchy of a multicore CPU. On top of the hierarchy are the CPU registers – the fastest memory – on the bottom is the main memory and in between lie the *caches*. The main memory is typically DRAM and located outside of the CPU (off-chip). A cache is a high-speed memory with low capacity, usually Static Random-Access Memory (SRAM), and usually integrated on the CPU (on-chip). There are multiple levels of caches: the first level is the *L1 cache* and it is private to each core of a CPU. The L1 cache is further divided into two parts: the L1I cache which stores the instructions and the L1D cache which stores the data. The next lower level(s) are often shared between multiple cores. The lowest cache level in the memory subsystem is called Last Level Cache (LLC). The LLC is connected to the main memory via a memory controller that manages the accesses of the processor to main memory. The memory becomes slower and larger as we move farther away from the processor and also the energy cost per access increases [JWN10, HW10].

Data in one cache level is also included in all the lower levels of the memory hierarchy in an *inclusive* cache system (*inclusion property*). On multicore systems, *cache coherence* must be maintained. Multiple cores may hold a copy of data referring to the same location in memory. The write to a memory location from one core must be propagated to the local caches of other cores before the data update becomes visible. Cache coherency is managed by hardware using a *cache coherency protocol*.

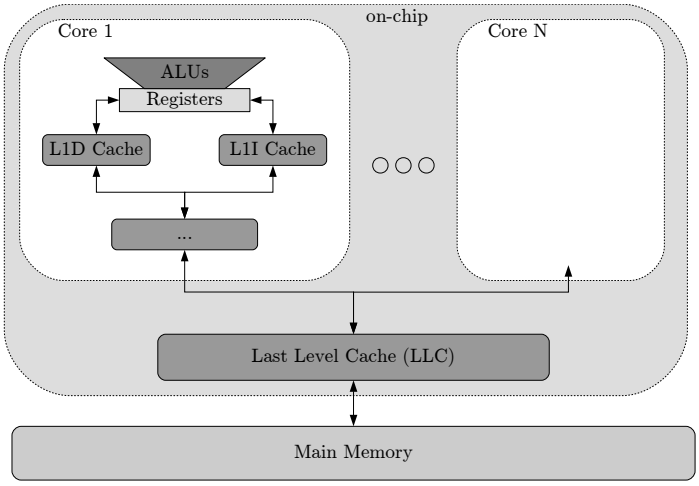


Figure 2.1 – Typical memory hierarchy on a multicore system. The L1 cache is divided into the L1D (data) and L1I (instruction) caches. L1 caches are local to the cores and the LLC is shared by many cores. Multiple cache levels may be between the L1 cache and the LLC. The LLC is connected to the off-chip main memory. Latency to the ALUs increases with the distance.

Based on [HW10, Fig 1.3.].

2.1.2 Data Transfer

When data is demanded by the CPU, it must be fetched from a lower level in the hierarchy. If the requested data is found in a cache level it is a *cache hit*, otherwise a *cache miss*. The time penalty for a cache miss depends on the latency, the bandwidth and on the level in the memory hierarchy in which the cache miss occurs. Earlier cache hits have lower latency and reduce the bandwidth requirements to the next lower level because less data must be transferred. The reduction in bandwidth requirements becomes even more important if the data path is shared by multiple cores.

For efficiency reasons, data is stored and transferred in cache lines (blocks) of fixed size containing multiple neighbouring words. All words in a cache line have the same *block address*, determined by the high bits of the word's address. The low bits of the address are called *block offset* and determine the location of the word in the cache line (see Figure 2.2). Where cache lines can be stored and thus must be searched for, depends on the mapping of block addresses to cache locations [HP17].

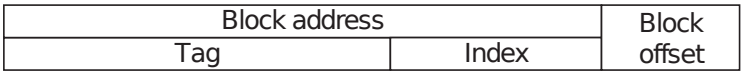


Figure 2.2 – The two parts of a word's address: block address and block offset. The block address can be further divided into the tag and index.

Based on [HP17, Fig B.3.]

2.1.3 Cache Mapping

The cache mapping, or cache placement policy, determines where a cache line can be placed and found in the cache. There are three major design choices of cache mappings – each having its downsides – of which the *set-associative* mapping is the most popular and discussed in more detail below [HP17].

- **Fully associative** caches can map every cache line to every location in the cache. They are expensive and can not be large.
- In **direct-mapped** caches, every cache line is mapped to one specific cache location. The location is fully determined, usually by the *index* (low bits) of the block address. They are cheap, but cache thrashing occurs due to conflict misses (explained in subsection 2.1.5).
- **Set-associative** caches map cache lines direct to a set and fully associative within the set. They form a trade-off between fully associative and direct mapped caches.

Set-associative Cache

Set-associative caches can be thought of as being composed of a number of m equally sized fully associative caches (sets). If there is space for n blocks in each set, the *associativity* is n and the cache placement is said to be *n-way set-associative*. Cache locations within the sets are called *ways*. Figure 2.3 shows an example of the placement of a data word in a 16-way set-associative cache.

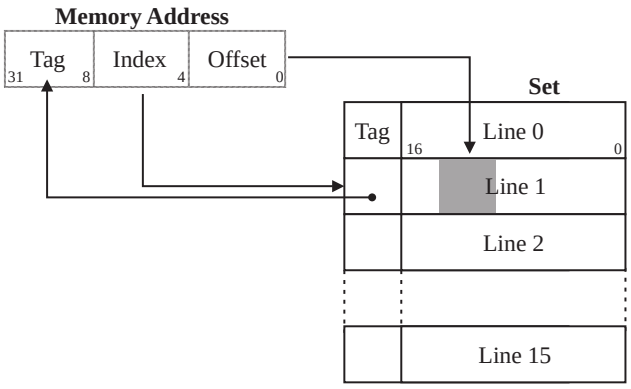


Figure 2.3 – Cache placement of a data word in a set of a 16-way set-associative cache. The set is selected by the index, the position in the cache line is determined by the offset. The tag is stored together with the cache line data to indicate the cache line content

Based on [PLM09, Fig 1.]

In set-associative caches, the block address is divided into two parts: the *tag* (high bits) and the *index* (low bits). The tag is required to indicate the corresponding location within main memory and is stored together with the data in the *tag array*. The index determines the set of a block usually by bit selection:

$$set\# = (block\ address)\ MOD\ m$$

However, there are also other ways to calculate the block's set from its address. Finding the location of a block in a set-associative cache consists of two steps: first the block address is mapped to one of the sets in the cache, and then all tags stored in the set are compared, typically in parallel. If no matching tag is found, a cache miss occurs.

Storing a block in an n-way set-associative cache is similar: first the block's set is determined, and then one of the blocks within the set is selected for removal (evicted). The evicted block is decided by the *replacement policy* (explained in subsection 2.1.4) and replaced with the incoming block [HP17, HW10]. The relationship between cache size, block size, associativity and number of sets is:

$$\textit{cache size} = (\textit{block size}) \cdot n \cdot m$$

Address Translation

Today, applications execute within an Operating System (OS)-managed virtual address space. Virtual addresses simplify programs and help increase security. The address specified within the program is the *virtual address*. The address sent to the memory system is the *physical address*. Addresses are mapped at the granularity of *pages* from *virtual pages* to *page frames* in main memory. The Memory Management Unit (MMU) is the translation unit of the CPU that performs the mapping from virtual to physical addresses. Recently used address mappings are stored in the Translation Lookaside Buffer (TLB), a cache that allows to speed up the translation process.

The tag and index parts of the block address can either be taken from the physical address, the virtual address, or both. For example, a cache using the virtual index and the physical tag is called *VIPT* cache. The advantage of using the virtual address parts is the lower latency, because address translation must finish before the physical index or tag may be used. But multiple virtual addresses may translate to the same physical address (aliasing) and the implementation becomes more difficult because of cache coherence [JWN10].

2.1.4 Replacement Policies

The *replacement strategy*, or *replacement policy*, decides which block is evicted in case of a cache miss. In a direct-mapped cache there is only one possible location for each block and thus there is only one choice for the replaced block. With fully associative mapping, every block can be chosen on a miss. In n-way set-associative caches, any of the n blocks within the indexed set can be chosen. The best choice would be to replace the block that will not be used for the longest period of time [Bel66]. However, with hardware it is not possible to know in advance what data will be needed again soon and therefore the replacement strategy is based on heuristics. Replacement strategies are one of the most heavily researched areas in cache design [JWN10].

LRU Replacement Policy

The Least Recently Used (LRU) policy is based on the heuristic that data that has not been required for a long time will most likely not be needed in the near future. To decrease the chance of throwing out data that will be required soon, LRU replaces the block that was unused for the longest time (LRU-block). LRU requires keeping track of the access order

and is often approximated, because there is a significant overhead in timing and energy cost involved by maintaining the required information to determine the LRU-block [Han98].

Pseudo-LRU Replacement Policies

LRU is also called *true-LRU* in order to distinguish from Pseudo-LRU (PLRU) policies that approximate LRU. These policies are cheaper to implement and today, commercial processors use only PLRU for caches with high associativity [KMCV10]. There are two major types of PLRU: *tree-PLRU* and *bit-PLRU* [XS20].

- **Tree-PLRU** is based on a bit-tree indicating the path to the PLRU-block and requires n-1 additional bits per set in an n-way set-associative cache. The tree-based PLRU is also simply referred to as PLRU in the literature [Han98].
- **Bit-PLRU**, also referred to as *not recently used* (NRU) [KMCV10], requires n additional bits per set (one per way), indicating the *most recently used* cache lines (MRU-bits). Every access to a cache line sets its MRU-bit to 1. When a cache miss occurs, one of the ways whose MRU-bit is 0 is selected for eviction. As soon as all MRU-bits are set to 1, they are set to 0, except in the replaced way. The way selected for eviction can e.g. be chosen at random or using the uppermost way whose MRU-bit is 0. It approximates LRU because the most recently used cache line is never evicted and the last evicted cache line before the MRU-bits are reset is always the (true-)LRU-block [HP17].

2.1.5 Cache Performance and Optimization

Cache optimization techniques aim at improving performance by reducing the number of cache misses or the miss penalty of accesses and many techniques have been developed. They are either hardware-based, such as increasing the cache size and hardware prefetching, or software-based such as access pattern and data layout transformation [HP17, KW03].

Cache misses can be the cause of performance-degrading CPU *stalls*. Stalls refer to the situation when the CPU cannot make progress because it has to wait until required data is available. Modern CPUs reduce stalls by resolving or relaxing data dependencies through the reordering of instructions in the instruction stream (out-of-order execution) and the ability to have multiple pending memory operations, but this does not fully avoid stalls due to cache misses [HW10].

Loading data ahead of time before it is demanded by the CPU can hide the access latency and is called *prefetching*. Prefetching makes use of otherwise unused bandwidth of the memory system, but may decrease performance when it interferes with the bandwidth requirement of demand misses. Demand misses are more likely to cause a stall. Determining what data will be requested next can be done either in hardware by detecting the access pattern of a program, or in software with prefetch instructions. The *miss rate* (misses per memory reference) metric is commonly used to measure the effectiveness of the cache system. It can be divided into demand miss rate and prefetch miss rate and depends on the considered cache level. An access responsible for a miss in one level might be a hit in the next lower level of the cache-hierarchy [HP17].

The causes of cache misses can be categorized in three categories (3C's) [HP17]:

- A **compulsory** miss, or **cold** miss, occurs when a cache block is referenced for the first time. Compulsory misses cannot be avoided, but the penalty can be reduced with prefetching.
- **Capacity** misses occur when the program's currently frequently used data (*working set*) is larger than the cache capacity and can only be avoided by increasing the cache size.
- A **conflict** miss appears in set-associative caches when the requested data was previously present, but got evicted by other data that was mapped to the same location in cache. Conflict misses can be reduced by increasing the associativity or cache size.

Cache-behavior of Applications

The memory access pattern of a programs is typically not random. Accesses often repeat themselves in time and tend to be near each other in the memory address space (*locality of reference*). These localities can be exploited to improve the cache utilization and it is often in the programmers responsibility to increase locality. A program has *spacial locality* when it has high probability that a data word next to the previously accessed word is demanded soon. This reduces cache misses, because the data will likely be located on an already fetched cache line. A program has *temporal locality* when the probability is high, that previously accessed data is demanded in the near future [JZTS10]. Based on their cache behavior, applications can be classified in multiple categories [Mit17]:

- **Insensitive** applications do not benefit from cache.
- **Friendly** applications benefit from cache.
- **Streaming** applications have a very large working set and inadequate cache reuse.
- A **trashing** application's working set is larger than the cache capacity.

Cache Content Management Optimizations

While many cache optimization techniques focus on reducing latency or cache misses by altering the cache design, the sector cache (described in section 2.3) aims at improving the cache utilization by optimizing the data placement. *Scratch-pad memory* and *cache bypassing* are similar in that sense and are described below. The sector cache is a type of *cache partitioning* which is explained in the next section.

Scratch-Pad Memory

Caches are usually transparent using built-in heuristics that determine what data to retain. Scratch-pad memory is managed explicitly and the application software has direct control over the cached data. Data allocation relies on manual code annotations by the programmer or can be automated by the compiler. The code annotations are usually not portable as they have to be manually adjusted for different memory sizes, which makes software development more expensive [JWN10]. Scratch-pad memory is often used as a complement to transparent caches and is part of many modern hardware

accelerators such as Field-programmable Gate Arrays (FPGAs) or Graphics Processing Units (GPUs) [SXS+15].

Cache Bypassing

Cache bypassing is a technique to reduce the bandwidth utilization or to increase the effective cache capacity by skipping placement of specific data in the cache. Mittal et al. describe different methods in [Mit16]. A simple example of cache bypassing is *non-temporal stores* (Intel's "*streaming stores*"), an instruction allowing to write an entire cache line to memory without going through the cache system. No prior read for the write-allocate is required and no cache space is consumed. Its utility is to store data without temporal locality [HW10].

2.1.6 Cache Partitioning

Cache partitioning is a cache optimization technique that divides the cache into multiple partitions, allowing to place data with high temporal locality in a separate partition to protect it from eviction. Typical goals are performance, fairness and quality-of-service improvements. The cache placement and replacement policy can be altered by cache partitioning and it obviously has no use if the application working sets fit in cache [BJM11].

On multicore processors, cache partitioning can resolve contention on shared resources between the partitioned cache and the next lower level in the memory hierarchy [Mit17]. For example, if a cache-friendly process and a streaming application run simultaneously on a machine with shared LLC cache, and the bandwidth is the bottleneck, allocating a separate partition to each process can prevent the streaming application from replacing reusable data of the cache-friendly process. This may improve the performance of both processes, because the increase in cache hits of the cache-friendly process reduces its bandwidth requirement and additional bandwidth is available to the streaming process.

Generally, one can differentiate between software-based and hardware-based cache partitioning. The granularity of cache partitioning techniques can be categorized into way-based, set-based and block-based [Mit17]. The sector cache of the A64FX is a way-based hardware cache partitioning. Mittal et al. provide a survey of cache partitioning techniques in [Mit17] and Balasubramonian et al. discuss a variety of cache partitioning research papers in [BJM11]. Most techniques focus on partitioning schemes in which different co-scheduled applications or threads are assigned to partitions of a shared cache on a multicore processor. The sector cache, on the other hand, is intended to assign data within a single application, or even a single thread, to partitions.

Intel CAT

Intel's Cache Allocation Technology (CAT) [Int15] is an example for a dynamic way-based hardware cache partitioning mechanism for shared LLCs. CAT is designed for improving performance, fairness and quality-of-service in shared computing platforms such as cloud computing e.g. by prioritizing LLC resource usage of important real-time processes or by enforcing fair sharing of cache resources between processes. Besides that, CAT can also increase security in shared systems by preventing cache side-channel attacks [HVA+16, LGY+16].

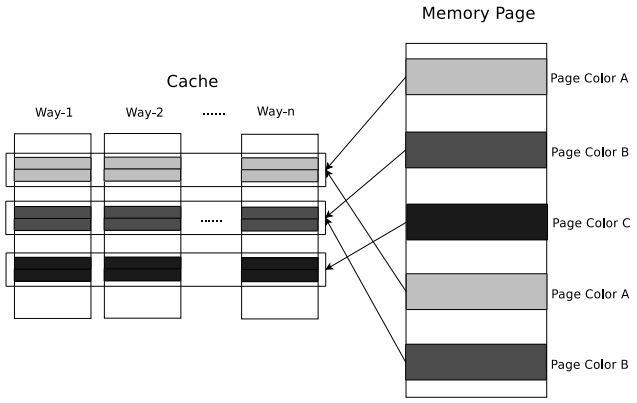


Figure 2.4 – Illustration of the page coloring technique for a set-associative cache. Memory pages mapping to the same cache lines are assigned the same color.

Taken from [ZDS09, Fig 1.]

OS Page Coloring

Page coloring is a software technique that controls the mapping of memory pages to cache lines. Memory pages mapping to the same sets are assigned the same color, as illustrated in Figure 2.4. The OS can control the page frames available to an application by restricting the virtual address space to those addresses matching certain colors and thus effectively partition the cache [ZDS09]. The OS page coloring partitioning by Lin et al. [LLD+08] has been integrated in the Linux operating system.

2.2 A64FX CPU

The A64FX CPU is an out-of-order superscalar processor designed for High-Performance Computing (HPC). The CPU has 52 cores in total and is organized in four Non-Uniform Memory Access (NUMA) domains called Core Memory Groups (CMGs). Each CMG consists of 13 cores, but only 12 out of the 13 cores are used for computation. A single compute core is also called Processing Element (PE) in the ARMv8 terminology. The A64FX has a peak DP performance of 3.1 TFLOPs and is used in the Fugaku supercomputer which is currently listed as #1 in the TOP500 list as of November 2021. Table 2.1 summarizes the most important hardware details [Fuj21a, Arm21].

2.2.1 Memory System

The memory hierarchy of the A64FX has two on-chip cache levels. Each core has a private L1 cache which is divided into L1D cache and L1I cache. The L1 caches of a CMG are connected to a shared L2 cache via a shared bus. A Move In Buffer (MIB) and Move Out Buffer (MOB) asynchronously manage move-in and move-out requests to the L2 cache (see Figure 2.5). Each of the four CMGs is connected to a 8GB High Bandwidth Memory (HBM) unit via its Memory Access Controller (MAC). The physical address space is divided between the CMGs. The read and write requests from the L2 cache are managed by the MAC. The L2 caches of the CMGs are interconnected by a two-way ring bus (see Figure 2.6). The

Table 2.1 – A64FX CPU and memory system facts.

Instruction set architecture	ARMv8-A, ARMv8.1, ARMv8.2, ARMv8.3, SVE
SVE-implemented vector length	128 / 256 / 512 bits
Number of compute cores	48 (12 per CMG)
Number of CMGs	4 (1 MAC per CMG)
Frequency	1.8GHz, 2.0GHz, 2.2GHz
Peak DP FLOPs	3.1TFLOPs (@ CPU 2GHz 2x FMA)
L1D cache size	64 KiB / 4-way set-associative VIPT (per Core)
L2 cache size	8 MiB / 16-way set-associative PIPT (per CMG)
Cache line size	256 bytes
Write method	Writeback
Memory capacity	8 GiB HBM Gen2 (per CMG)
L1D <-> L2 R/W	64/32 bytes / cycle (per Core)
L1D <-> L2 (shared) R/W	512/256 bytes / cycle (per CMG)
L2 <-> Memory R/W	128/64 bytes / cycle (per CMG)

peak memory bandwidth of one MAC is 256GB/s and 1024GB/s for the whole system (read-only). A hardware prefetcher detects streaming access patterns and can be refined using the *hardware prefetch assist mechanism* [Fuj21a].

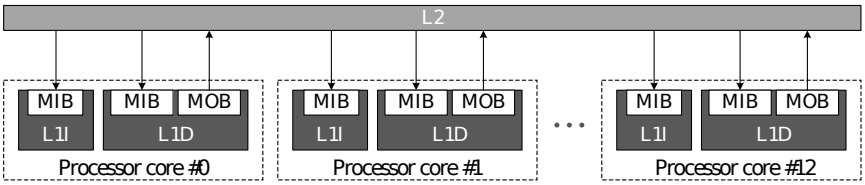


Figure 2.5 – A64FX memory hierarchy: the private L1 caches of one CMG are connected to a shared L2 cache via a shared bus.

Taken from [Fuj21a, Fig. 9-2]

2.2.2 Instruction Set Architecture

The A64FX is the first processor of the ARMv8-A Scalable Vector Extension (SVE) architecture and supports a maximum vector length of 512 bits. SVE is a vector extension of the ARM Instruction-Set Architecture (ISA) that allows the programmer to write vectorized code without specifying the actual vector length. The ARMv8-A ISA is extended by the *Fujitsu HPC extensions* on the A64FX [Fuj21a].

ARMv8-A Exception Levels

The ARMv8-A architecture [Arm21] defines a set of Exception Levels (ELs) from EL0 to EL3 that indicate software execution privileges in ascending order. Execution at EL0 is called *unprivileged* execution and user applications typically run in EL0. EL1 is commonly

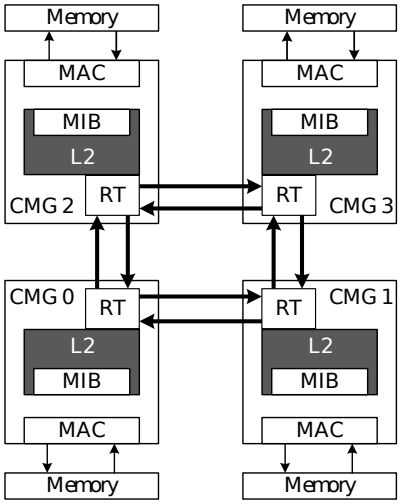


Figure 2.6 – A64FX memory hierarchy: the A64FX is organized in four CMGs. Each CMG has a L2 caches and is connected to its HBM unit via its MAC. The CMGs are interconnected by a ring bus.

Taken from [Fuj21a, Fig. 9-1]

used for the OS kernel and associated functions that are typically described as *privileged*. User applications typically run in EL0. The EL becomes important for the sector cache configuration, because some of the required system registers are only accessible in EL1. Accessing these registers from a user application requires switching to EL1.

Memory Tagging

Today, there is no need yet for a 64-bit address space. Memory tagging means that additional information – a tag – is stored in the otherwise unused high bits of the 64-bit virtual addresses and it is part of the ARMv8 memory model. Although ARM requires that the top 16 bits of addresses must be either 0x0000 or 0xFFFF, a hardware feature introduced with ARMv8, called *top-byte ignore*, allows software to use the 8 most significant bits of pointers as a tag. When the top-byte ignore feature is activated, the 8 high bits of the virtual address are ignored during address translation. Setting the top eight bits of the virtual address is called *top-byte override* [Arm21].

One of the use cases is the *memory tagging extension* [Fra19] introduced with ARMv8.5-A that aims to increase memory safety. A lock value is associated with the memory locations of a memory allocation. Access to those memory locations is only granted when the key of a reference, stored in the tag of the virtual address to that location, matches the lock value. Memory tagging is also required in the Fujitsu HPC extensions of the A64FX.

Fujitsu HPC Extensions

The Fujitsu HPC extensions consist of four hardware features that derive from previous Fujitsu processors (e.g. SPARC64 VIIIfx) and can potentially improve the performance of applications. The *hardware barrier* supports the synchronization between software threads with hardware. The *HPC tag address override* function is equivalent to the top-byte ignore

and top-byte override functionality of the ARMv8 ISA and is required for the control of the *hardware prefetch assistance* and the sector cache features. Software can hint the A64FX's hardware prefetch mechanism through the hardware prefetch assistance in order to reduce the penalty of memory accesses by providing an access pattern in advance. The sector cache feature is also part of the Fujitsu HPC extensions [Fuj20b, Fuj21a].

2.3 Sector Cache

Sector cache is a cache partitioning function which can virtually split a cache in multiple configurable partitions ("sectors"). Cache partition sizes are chosen by allocating a number of cache ways to each sector in the related system registers (see subsection 2.3.3) and may be changed dynamically during program execution. The partitioning effectively changes the replacement policy and can improve performance especially in memory-intensive programs by protecting data with high temporal locality from eviction.

The technical name for the hardware implementation is *instruction-based way partitioning* [PS12] and it was first employed in the SPARC64VIIIfx processor [YHKS12] for the K computer (at that time #2 TOP500). The SPARC64VIIIfx is a predecessor of the A64FX.

There are sector cache controls for each L1D cache of each core and each L2 cache of each CMG. The L1D and L2 caches each have four sectors, but the L2 cache sectors are structured hierarchically. The four sectors of the L2 cache are grouped into two sector groups as shown in Figure 2.7. The four sectors of each core's L1D cache are mapped to two sectors of one sector group within the L2 cache of their CMG. Each core can use two sectors of one selected group of the L2 cache [Fuj21a].

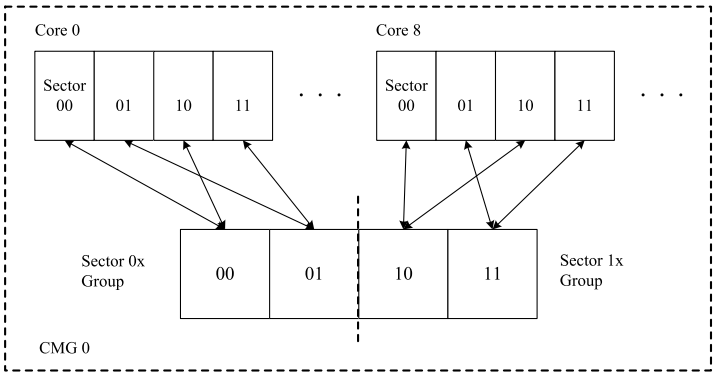


Figure 2.7 – Sector cache partitioning on the A64FX CPU. Within each CMG, the four sectors of each core's L1D cache are mapped to the two sectors of one sector group of the L2 cache.

Taken from [Fuj21a, Fig. 12-1]

2.3.1 Sector Cache Behaviour

Tagged addresses on load, store and prefetch instructions specify the sector for the data on each memory instruction and the data is stored together with the sector id in cache. If no sector id is specified, the default sector is assumed. When the sector id in a tag differs from the stored sector id of an accessed block, the sector cache *operating mode* decides whether

or not the sector id is updated on access. The sector information is stored and propagated on write-back from L1D to L2, even if the L1D sector cache is disabled [Fuj21a, Fuj20b].

Data in a sector is not evicted from cache as long as the consumed area of the sector is lower than its maximum capacity. When an entry has to be evicted, a block is selected such that each sector does not exceed its specified sector size. Software can always access data in all cache ways [Fuj21a].

Dynamic Capacity Adjustment

The maximum capacity of a sector may change during program execution and the hardware will gradually bring the capacity of the sectors to the newly specified maximum capacity on data loads [Fuj21a, Fuj20b]. The behaviour in case the capacity is changed is explained using the next two examples. The current situation is shown on top, whereas the new situation is shown on the bottom of each figure. Both examples show one set of an n-way set-associative cache with two sectors. The numbers show the LRU numbers: the block with number 1 is the most recently used and the block with number n is the LRU-block. The LRU numbers are increased by one in the step from current to new.

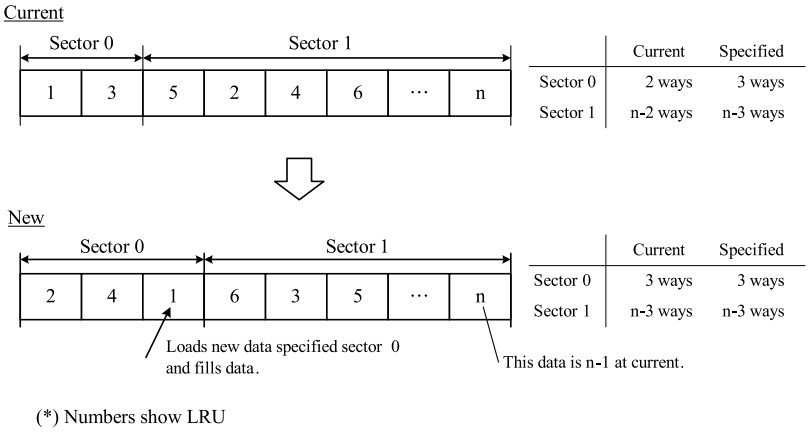


Figure 2.8 – Sector cache behaviour when data with sector tag is loaded and the sector capacity is currently lower than specified.

Taken from [Fuj21a, Fig. 12-2]

In Figure 2.8, the current capacity of sector 0 is 2 ways and was increased to the new maximum capacity of 3 ways. On the other hand, the current capacity of sector 1 is n-2 ways and was decreased to the new maximum capacity of n-3 ways. Thus, the current capacity of sector 0 is lower than specified and the current capacity of sector 1 is higher than specified. In this case, when data with specified sector 0 is loaded, instead of the LRU-block of sector 0, the LRU-block of sector 1 is evicted and replaced with the new data of sector 0. The capacities of the sectors changed to the specified values.

Figure 2.9 shows the same initial and specified capacities with different LRU numbers, but data with specified sector 1 is loaded. The LRU-block in the set is in sector 0, but since sector 1 was specified and the current capacity of sector 1 is not lower than the specified capacity, the LRU-block of sector 1 (number n-1) is evicted. The capacities of both sectors did not change in this case.

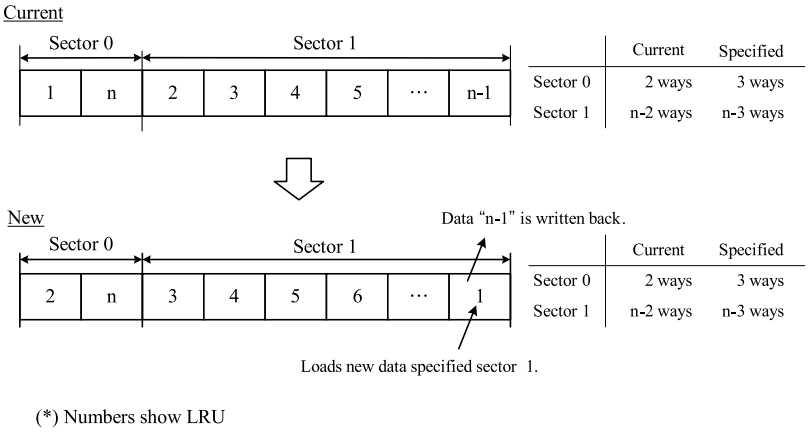


Figure 2.9 – Sector cache behaviour when data with sector tag is loaded and the sector capacity is currently higher than specified.

Taken from [Fuj21a, Fig. 12-3]

2.3.2 Sector Cache Replacement Policy

Unfortunately, Fujitsu did not publish every detail of the A64FX sector cache mechanism in the available A64FX documentations [Fuj20b] and [Fuj21a]. Also the replacement strategy of the L1D and L2 caches was not published. A more detailed description of the sector cache mechanism can be found in the SPARC64 VIIIfx extensions documentation [Fuj10]. According to [Fuj10], all sets have the same sector sizes and a capacity counter keeps track of each sector's consumed area within each set. Each sector must implement its own LRU policy to determine the LRU-block of the sector. It is not documented if the caches use true-LRU and how multiple (partitioned) LRU policies are managed. Perarnau and Sato state in [PS] that the SPARC64 VIIIfx L2 cache uses a PLRU policy.

It is reasonable to assume that the A64FX sector cache mechanism is similar to that of the SPARC64 VIIIfx and that the replacement of the L2 cache likely obeys a PLRU policy. The underlying replacement policy implementation of the A64FX sector cache could be similar to that in [CLS06], where the authors combine a *way partition binary tree* with a binary bit-tree to implement a partitioned tree-PLRU algorithm in a way-partitioned cache. However, the replacement policy of the A64FX could also be based on the bit-PLRU policy. Regarding the available information provided by the Fujitsu documentations, a model is derived how the bit-PLRU policy can be efficiently extended to a partitioned policy – we call it *partitioned bit-PLRU policy* – and is explained below. The policy shall reproduce the sector cache behaviour described above and approximate LRU.

Partitioned Bit-PLRU Policy

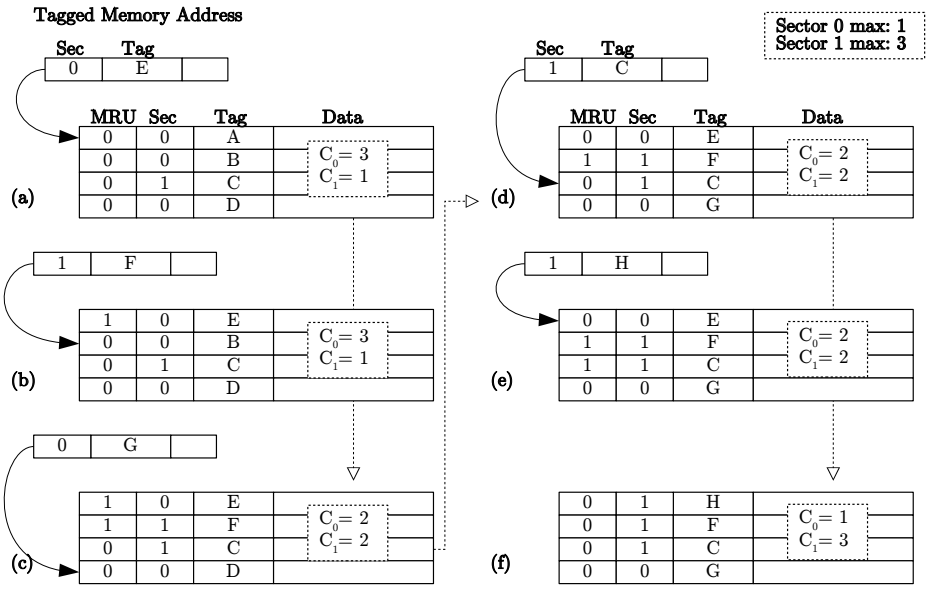


Figure 2.10 – Partitioned bit-PLRU policy in a set of a 4-way set-associative cache with two sectors. Five tagged memory accesses are made. The cache's state before and after the accesses is shown from (a) to (f). The **MRU** column indicates the most recently used cache lines w.r.t. each sector. The sector id is updated on access and stored in the **Sec** column. C_0 and C_1 are the sector capacity counters.

1. The cache is extended by an array of sector id bits (Sec) and MRU-bits. The MRU-bits are initialized to 0.
2. When a cache **hit** occurs:
   - Set the MRU-bit of the cache line to 1 and update its sector id according to the sector cache operating mode.
3. When a cache **miss** occurs:
   - Compare the sector capacity counter of the specified sector in the address with the corresponding maximum sector capacity to determine the sector from which the evicted cache line is selected.
     a) **counter $\geq$ max. capacity**: select from specified sector for eviction.
     b) **counter $<$ max. capacity**: select from other sector for eviction. Increment and decrement the corresponding capacity counters by 1.
   - Choose the uppermost cache line whose MRU-bit is 0 **and** whose stored sector id matches the previously selected sector to be replaced from.
   - Replace the selected cache line, set its MRU-bit to 1 and update its sector id according to the sector cache operating mode.
4. As soon as all MRU-bits in the cache lines whose stored sector id matches the selected sector are set to 1, they are set to 0, except in the replaced cache line.

An example of the partitioned bit-PLRU policy for a set within a 4-way set-associative cache with two sectors is illustrated in Figure 2.10 for five tagged memory address accesses. The sector cache operating mode is chosen such that the sector id is updated on access. In the end, the most recently used cache lines (F,G,C,H) are kept and the consumed areas of the sectors meet the specified maximum capacities. Additional overhead occurs for the sector select and capacity counter logic compared to bit-PLRU. To our best knowledge, this policy mimics the sector cache behaviour described in the documentations.

2.3.3 Sector Cache Configuration

To use the sector cache function in user software, the related system registers must be configured and the virtual addresses of memory instructions must be tagged with the sector id. Accessing the system registers can generally be performed with the MRS and MSR instructions of the ARMv8 instruction set. The MRS instruction moves data from a system register to a general-purpose register and the MSR instruction moves data from a general-purpose register to a system register. Registers with a register name suffix "EL0" may be accessible from every EL while registers ending with "EL1" require EL1 or higher [Fuj20b, Arm21].

1. Enabling the HPC tag address override function
2. Setting the access control of the system registers
3. Setting the allocation and operation control of the system registers
4. Configuring the maximum sector capacity of the L1D and L2 caches
5. Tagging of memory accesses (loads, stores, prefetches) to specify sectors on data access

These steps can be performed either manually or with the *Fujitsu software compiler package*. Each of these steps is explained in this section. A more detailed description can be found in the Fujitsu HPC Extension documentation [Fuj20b]. For the sake of readability, the registers are referenced without their prefix and suffix below. The purpose, full name and the shared domain of the relevant registers are listed in Table 2.2. The *RES0* fields in the registers are unused and reserved.

Enabling the HPC Tag Address Override Function

The system register FJ_TAG_ADDRESS_CTRL controls the ARMv8 memory tagging and the HPC tag address override function for the hardware prefetch assistance and the sector cache. It contains fields that activate the sector cache (*SCE{0|1}*) and the prefetch assistance (*PFE{0|1}*). Only when the *SCE* field is set to 1, the sector id of a tagged address is valid. The fields *TBO{0|1}* activate the ARMv8 top-byte ignore function and must also be set (see Figure 2.11).

Setting the Access Control

The system register SCCR_CTRL (Figure 2.12) controls the access rights to the sector cache configuration registers. It has two fields, *el1ae* and *el0ae*. Both fields must be set to 1 access the sector cache configuration registers from EL0 (see Figure 2.12).

Table 2.2 – Tag address and sector cache control and setting registers.
Shared domain PE: one register per PE.
Shared domain CMG: one register per CMG.

Purpose	Register Name	Shared domain
HPC tag address override control	IMP_FJ_TAG_ADDRESS_CTRL_EL1	PE
Access control	IMP_SCCR_CTRL_EL1	PE
Allocation and operation control	IMP_SCCR_ASSIGN_EL1	PE
L1D capacity setting	IMP_SCCR_L1_EL0	PE
L2 maximum capacity setting	IMP_SCCR_SET{0|1}_L2_EL1	CMG
L2 capacity setting	IMP_SCCR_VSCCR_L2_EL0	PE

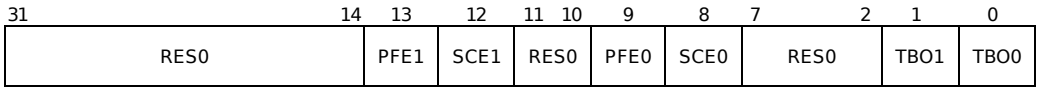


Figure 2.11 – HPC tag address override control register.

Taken from [Fuj20b]

Setting the Allocation and Operation Control

The operating mode can be set in the *mode* field of the system register SCCR_ASSIGN (see Figure 2.13). When the *mode* field is set to zero, the sector id of the accessed cache line is updated to the sector id specified in a tag, otherwise it is kept.

The *assign* field controls which of the two L2 maximum capacity setting registers is updated when a PE sets the L2 capacity by writing to its (*window*) register SCCR_VSCCR_L2.

The *default_sector* field specifies the default sector in case the sector id is not specified in a tag.

Setting the L1D and L2 Sector Cache Capacities

The maximum number of L1D cache ways allocated to each sector can be assigned by setting the *l1_secN_max* fields (N is the sector id) of the L1D sector cache capacity setting register SCCR_L1 (see Figure 2.14). The L2 sector cache configuration is a bit more involved, because the L2 cache is shared by all PEs of one CMG. The *l2_sec0_max* and *l2_sec1_max fields* of the two registers SCCR_SET{0/1}_L2 define the sector capacities of sector group 0 and sector group 1 within one CMG. Setting the corresponding register from EL0 is performed by writing to the window register SCCR_VSCCR_L2 from one PE. The update of one PE is

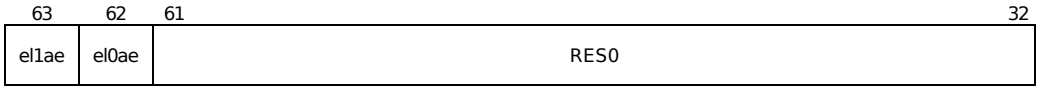


Figure 2.12 – Sector cache access control register.

Taken from [Fuj20b]

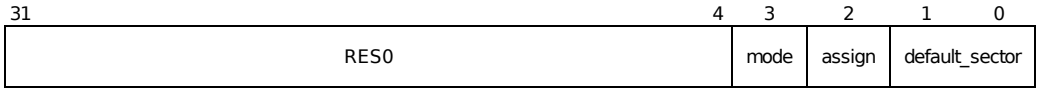


Figure 2.13 – Sector cache allocation and operation control register.

Taken from [Fuj20b]

valid for all PEs of the same CMG.

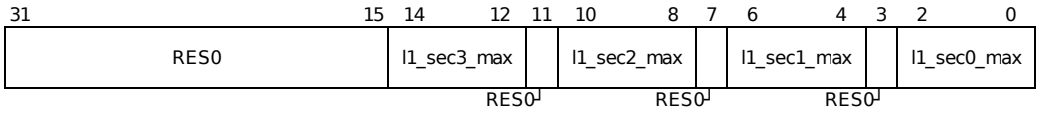


Figure 2.14 – L1D sector cache capacity setting register.

Taken from [Fuj20b]

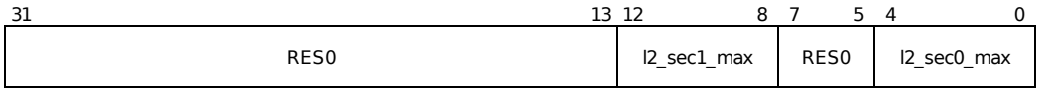


Figure 2.15 – L2 sector cache capacity setting registers.

Taken from [Fuj20b]

Tagged Address Allocation

The upper 8 bits – the HPC tag – of a tagged address are shown in Figure 2.16. The *pf_func* field controls the hardware prefetch assistance and the *SBZ* fields are ignored. The two bits of the *sector_id* field specify the sector id (0-3).

Sector Cache using the Fujitsu C/C++ compiler

The Fujitsu software compiler package includes compilers for the C, C++ and Fortran language that leverage the use of the Fujitsu HPC extensions for software developers through compiler directives, called Optimization Control Lines (OCLs). Instead of a manual configuration, the setup of the system registers is taken over by the compiler and the setting registers are made accessible from a user application. In this thesis, the Fujitsu C/C++ Compiler (FCC) was used for compilation and the sector cache configuration.

Listing 2.1 shows an example of the OCL usage for the sector cache. The L2 cache is partitioned into $n1$ ways for sector 1 and 16-$n1$ ways for sector 0. The directives #pragma statement scache_isolate_way and #pragma statement end_scache_isolate_way mark the beginning and end of the cache partitioning. The number of L2 cache ways reserved for sector 1, and optionally L1D cache ways, has to be provided after the directive (see line 2). Pointers and arrays can be assigned to sector 1 through the OCL #pragma statement scache_isolate_assign. The end of the assignment has to be marked with a matching #pragma statement end_scache_isolate_assign. In this example, a is assigned to sector 1 and all other accesses go to sector 0.

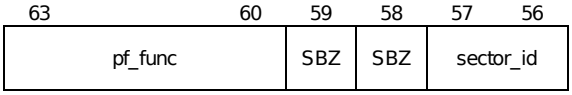


Figure 2.16 – Address tag for the Fujitsu HPC extensions.

Taken from [Fuj20b]

```
1  /* Reuse the array a that can fit in n1 ways in the L2 cache */
2  #pragma statement scache_isolate_way L2=n1 // [L1=n2]
3  #pragma statement scache_isolate_assign a
4  for(int j = 0; j < n; j++) {
5  #pragma omp parallel for
6      for(int i = 0; i < m; i++) {
7          a[i] = a[i] + b[j][i];
8      }
9  }
10 #pragma statement end_scache_isolate_assign
11 #pragma statement end_scache_isolate_way
```

Listing 2.1 – Sector cache example.

The OCLs for the sector cache can either be applied to a region of code using the begin and end delimiters as shown in the example code, or to a whole function. The interface for whole function OCLs is very similar: instead of a #pragma statement [...], the directive begins with #pragma procedure [...] and does not require an end delimiter because it applies to the whole function it is placed in.

Listing 2.2 shows the relevant fields of the sector cache system registers state from a user application point of view using FCC. Line 5-9 is the output of the tool *pmsecstat*, which allows to read the registers from user space. Line 5 shows that configuring the sector sizes is made accessible from EL0. Line 6 shows that the sector cache operates in mode 0 (see section 2.3.3). It also shows that the register SCCR_SET0_L2 (sector group 0) is used for the L2 cache partitioning and that the default sector is sector 0. Line 7 and 8 show that only sector 0 and sector 1 of sector group 0 are used with the OCLs and that the number of cache ways specified in the OCL applies to sector 1. All cores of the A64FX are configured the same using FCC.

```
1  #pragma statement scache_isolate_way L2=7 L1=3
2  system("pmsecstat");
3  #pragma statement end_scache_isolate_way
4
5  SCCR_CTRL     : el1ae=1 el0ae=1
6  SCCR_ASSIGN   : mode=0 assign=0 default_sec=0
7  SCCR_L1       : l1_sec3=0 l1_sec2=0 l1_sec1=3 l1_sec0=1
8  SCCR_SET0     : l2_sec1=7 l2_sec0=9
9  SCCR_SET1     : l2_sec1=0 l2_sec0=2
```

Listing 2.2 – Sector cache register state using the Fujitsu compiler with partitioning scheme L2 cache 7:9 ways and L1D cache 3:1 ways.

From a practical point of view, there are some drawbacks to this interface. For example, if a pointer aliases the assigned isolated pointer or array, the compiler does not generate a tagged memory instruction for accesses via the aliasing pointer [PS12]. In this case, the default sector is used if not stated otherwise explicitly in code. The same applies to arrays and pointers passed as function arguments. However, the differentiation is also necessary. Another problem is that the compiler does not automatically use the sector cache or give application developers advice on where the feature could improve code performance. A tool-based approach could help programmers identify regions of code where this is the case. The reuse distance metric discussed in the next section can form the basis of such a tool.

2.4 Reuse Distance

Reuse distance or LRU stack distance is a measure for program locality (data reuse) that can be used to estimate the number of cache misses of an application that runs on a machine with a cache with (pseudo-) LRU replacement policy. The reuse distance metric is purely a property of programs and not machine-specific [ACP02].

Definition 1. The reuse distance of a reference to element x is the number of **distinct** data elements that have been referenced since the last access of x, or ∞ if x has not been referenced before [SPP10].

2.4.1 Reuse Distance Calculation

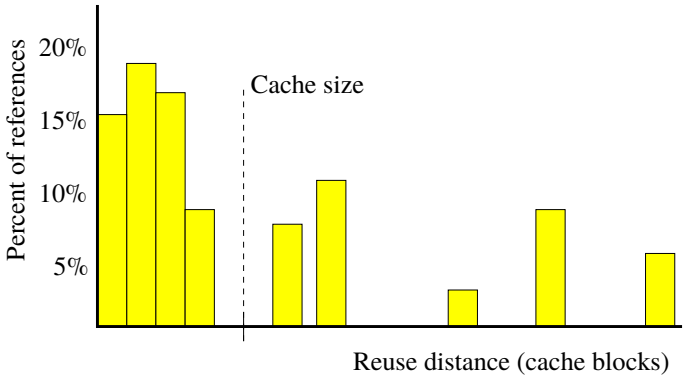


Figure 2.17 – Reuse distance histogram. References with reuse distance lower than the cache size are cache hits, others are misses.

Taken from [JZTS10, Fig 1.]

Reuse distance can be calculated using a stack processing algorithm based on the LRU replacement policy – the LRU stack algorithm. Mattson et al. introduced the LRU stack algorithm in [MGST70], where they evaluated various replacement strategies for virtual memory paging – LRU being one of them – using page reference traces of typical applications. However, cache behaviour can be modelled using cache line references instead of page references. Today, reuse distance is often used to analyze the cache behaviour of applications. The number of cache misses in fully associative caches with LRU replacement policy is computed exact, and can be approximated in set-associative caches by reuse distance. In a fully associative LRU

cache with space for n cache lines, a reference with reuse distance $d < n$ will hit. A reference with reuse distance $d \geq n$ will miss [ACP02, BD01].

Definition 2. A trace T is a sequence of memory references: $T = (x_0, x_1, \ldots, x_n)$.

The LRU stack algorithm is shown in Algorithm 1 in its most simple form. It takes a trace of memory references as input and associates the reuse distance with each reference. Each time when a memory location was referenced, its reference is pushed on top of a stack. Older references sink towards the bottom of the stack. If a location is referenced for the first time, its reuse distance is defined as ∞. If a location was already referenced before, its reference is already present somewhere in the stack. When it is referenced again, its current distance to the top is the reuse distance. The reference is moved to the top of the stack after the distance is recorded. The stack can be represented as a doubly-linked list.

The algorithm's output is a sequence of stack distances that can be coalesced into a stack distance histogram. An example of such a histogram is shown in Figure 2.17. The reuse distance is plotted on the x-axis while the number of references with the corresponding reuse distance, typically related to the total number of references made, is plotted on the y-axis. References with reuse distance lower than the cache size are cache hits, others are misses.

The algorithm can be divided into three phases: the *search phase* is required to check if the reference is already present in the stack and its current position. The reuse distance is obtained during the *count phase*. Bringing the most recent reference to the top of the stack is done in the *update phase* [ACP02].

The search phase can be performed in $\mathcal{O}(1)$ operations using a hash map. The update phase involves simply removing and inserting an element in the beginning of a linked list and is also cheap. The expensive part is the count phase. In a naive implementation, the stack is traversed until the reference is found and, in the worst case, is located at the bottom of the stack. Almasi et al. provide an overview of stack processing methods, including more sophisticated approaches, in [ACP02].

Algorithm 1 LRU stack distance

```
stack ← ∅
for all memory references x in T do
    if x ∉ stack then                                   ▷ search phase
        x.dist ← ∞
    else
        x.dist ← DISTANCE(x)                            ▷ count phase
        REMOVE(stack, x)                                ▷ update phase
    end if
    INSERT(stack, x)                                    ▷ inserts on top
end for
```

2.4.2 Concurrent Reuse Distance

While reuse distance can predict the cache behaviour of a single threaded application, it does not reflect the interaction of multiple cores accessing a shared cache in a shared-memory environment and the distances may change when a sequential program is parallelized. For example program transformations that create parallel loops can change the stack distance

of references within the loop [BD01]. Jiang et al. [JZTS10] extend the concept of reuse distance for shared caches by the introduction of *concurrent reuse distance*.

Concurrent reuse distance is defined as the number of distinct data elements that *all sharers of a cache* access between the current and the previous references to the same data element. This definition is no longer hardware-independent because its value depends on the relative execution speeds of cache sharers. Typically, reuse distance information is obtained by instrumenting an application's memory accesses. The instrumentation changes the execution speed, making this definition problematic for independent concurrent applications, but the relative speed of concurrent threads within an application remains the same [JZTS10]. Schuff et. al [SPP10] propose a thread interleaving with one cycle per instruction per core. A probabilistic approach is taken in [JZTS10]. The interleaving can be performed either during or after the instrumentation.

2.5 Related Work

Soft-OLP

Lu et al. developed the binary instrumentation framework Soft-OLP [LLD+09] that performs automatic software cache partitioning of "*objects*" based on OS page coloring. Objects are e.g. dynamically allocated memory regions in their terminology, and called *data structures* throughout this thesis and also in next discussed paper [PS12]. The authors analyze serial program traces at the whole-program level to obtain two types of reuse histograms: *per object* and *inter-object interference* histograms. Applications are instrumented and profiled with varying training input parameters. A polynomial curve fitting function is applied on the obtained reuse histograms to extrapolate the reuse distances for other input parameters. By assigning objects to cache partitions e.g. in the CG and LU benchmark of the NAS parallel benchmarks, a cache miss and run time reduction is achieved.

Tool-based Approach for the Sector Cache

The closest related work is [PS12] by Perarnau and Sato. The authors find that making good use of the sector cache is hard without deep knowledge about the code locality and the memory hierarchy of their considered system (SPARC64 VIIIfx processor). They develop a tool to identify code regions that could benefit from the sector cache to simplify the optimization of memory-intensive HPC applications when they are ported to their system. The authors design a binary instrumentation tool to measure the locality of program data structures in order to determine the best sector cache configuration. The locality measurement is based on reuse distance. Application binaries are instrumented using Intel PIN [LCM+05].

To predict the cache misses in case a certain data structure is isolated from others, they calculate two reuse distance histograms: one for addresses that belong to the data structure and one for those outside the address range of the data structure. Reuse distance is calculated using Olken's algorithm [Olk81].

The user has to provide the symbol names of the considered data structures. The program's DWARF debugging information, which is standard debugging information format under Linux, is used to associate memory references with the data structures.

A handcrafted 2-d multigrid solver is used for the evaluation. The multigrid method applies a stencil kernel to matrices of varying sizes. This class of applications was chosen

because stencil kernels are typically memory-intensive and the matrices of the multigrid solver each have different cache size requirements for optimal reuse. The authors find that their tool can predict a near optimal sector cache configuration, reducing the number of cache misses up to 20%. In the follow-up paper [PS], the tool is applied to the NAS parallel benchmarks and the sector cache is found to be beneficial to the CG and LU benchmarks. As future work, they want to compute reuse distances for each function, instead of just the whole-program reuse distances, in order to refine the profiling.

Sector Cache on the A64FX

In [AML+21] the authors provide an architectural analysis of the A64FX CPU and also investigate the HPC extension features of the A64FX processor. The sector cache is applied to dense- and sparse matrix-vector multiplication. They find that by isolating the non-reusable matrix data in a small sector of the L2 cache, they can extend a high performance level to larger vectors (from 2MB to 5MB) compared to execution without sector cache in the dense matrix-vector multiplication, because data can be kept in L2. The performance of the sparse matrix-vector multiplication was be improved up to 30%, but this strongly depends on the input matrix.

3 Methodology

It is not always clear beforehand when and if the sector cache feature can improve an application's performance. Even the hardware manufacturer states that it is difficult to use the sector cache function meaningfully [Fuj21b]. A method of detecting these cases can simplify the application design or porting process. In this work, a tool-based approach is followed and is explained in the following chapter. The metrics used to evaluate the sector cache effectiveness and the tool are defined at the end of the chapter.

In order to gain a better understanding of the effects on performance and behavior using the sector cache, as well as an upper limit for the potential improvement in performance, a number of microbenchmarks are also developed. They are described and evaluated in chapter 5.

3.1 Data Structure Interference

Let us consider the following example of a trace shown in Figure 3.1, but first let us define terms in order to be able to explain the basic idea of data structure interference and its application to the sector cache with the help of the reuse distance.

Definition 3. X is the set of all **unique** references in a trace T.

Definition 4. A data structure $D \subseteq X$ is a subset of X.

Definition 5. The complementary data structure $\overline{D}$ is defined as $X \setminus D$.

The example trace consists of references belonging to two data structures: A and B (B is the complementary to A and vice versa in this case). The block with reference $a1$ is accessed twice and the reuse distance of the second access is $RD = 5$. If we split the trace by data structure into one trace containing all references to A and one trace for $\overline{A}$, the reuse distance of the second access to $a1$ is reduced to $RD = 1$. Accesses to B *interfered* (increased the reuse distance) with accesses to A. Assuming we have a fully associative LRU cache with space for four cache blocks, the second access to $a1$ would be a miss. However, if the cache is partitioned into one partition with at least two blocks dedicated to A and one partition with the remaining quota allocated to $\overline{A}$, the access to $a1$ would be a hit and exactly such a partitioning is possible using the sector cache. The reuse distances of B are not negatively affected by the partitioning and the sector cache can improve the cache behaviour in this example. The example is typical for a code region where one data structure is accessed repeatedly (A), while other data (B) is streamed in a loop.

This leads us to the following general approach: an application's memory trace is recorded and for each data structure of interest, the trace is split into two traces: $T(D)$ and $T(\overline{D})$ and their reuse distances are calculated. From the difference of the reuse distances, compared to the original trace, it is possible to calculate the benefit of isolating D in a partition. The trace can be recorded by instrumenting the application binary to obtain the application's *reuse distance profile*.

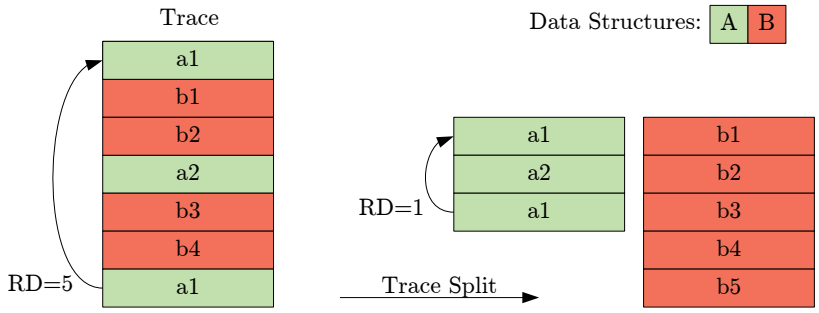


Figure 3.1 – Data structure interference. Non-reusable data of data structure B (red) interferes with reuse of A (green). The reuse distance of the second reference to a1 is reduced from $RD = 5$ (left) to $RD = 1$ after extracting the accesses to A from the trace (right).

3.2 Program Phases

The locality properties of a program typically depend on its current program phase (code region). For example, an application might first do some initialization of data, then process data with one algorithm and afterwards process data with another algorithm. The initialization and different algorithms can have completely different access patterns and localities. *Whole program analysis* would ignore this fact and it does not apply to the dynamic cache partitioning capabilities of the sector cache, because the same partitioning that significantly reduces cache misses in one program phase might increase the misses dramatically in another. Thus, the partitioning has to be dynamically adopted according to the current phase. It is crucial to take this into account, but defining and detecting program phases is difficult. Program phase detection is a whole area of research [DS03] and can be supplemented by reuse distance as shown in [SZD04].

Typically, applications are structured in functions that fulfill a certain task or implement an algorithm, and it is reasonable to assume that each function has its own static locality property. Throughout this work, we define one program phase for each function. The code region granularity might be too high, or too low in some cases, but the assumption should roughly hold.

3.3 Data Structures

Not only identifying program phases is difficult, but also determining which data structures should be considered is challenging. In general, a data structure could be **any** subset of X. But calculating the reuse distances of the split traces for each possible subset of X (powerset $\mathcal{P}(X)$) would require the stack processing of $|\mathcal{P}(X)| = 2^{|X|}$ possible traces. This is not feasible and the considered data structures must be restricted.

3.3.1 Data Structure Selection

Most virtual addresses within a trace refer to either heap-allocated memory, or to stack-allocated memory and larger memory regions often reside on the heap. The stack usually contains variables of basic types or small arrays. We restrict the data structures of interest

to dynamically allocated heap memory with a size above a certain threshold. The threshold is chosen in the order of magnitude of the smallest possible cache partition size. Rationale: most of the accesses in scientific applications are made on large dynamically allocated memory within loops.

3.3.2 Data Structure Composition

There are cases where the simultaneous isolation of two or more heap-allocated memory regions leads to better results compared to isolation of a single memory region.

Definition 6. The composed data structure of two data structures D_i and D_j is the union of both data structures: $D_{ij} := D_i \cup D_j = D_{ji}$.

If there are N_D heap-allocated memory regions, there are $N_D(N_D - 1)/2$ possible combinations of them. In order to obtain their reuse distances, the number of times the stack processing algorithm must process their split traces is in the order of $\mathcal{O}(N_D^2)$. If N_D is kept low by preselecting data structures, this is still feasible. Stack processing of the composition of all possible triplets of data structures already requires $\mathcal{O}(N_D^3)$ passes. However, it is possible to obtain the reuse distances for the composition of three or more data structures with a number of passes in the order of $\mathcal{O}(N_D^2)$ by construction from the reuse distances of single data structure traces and their interference with others. This was shown in [LLD+09], but not applied in this thesis.

3.4 Optimal Cache Partitioning

Although the A64FX sector cache can in principle divide the cache into more than two partitions, the FCC's compiler directives only allow a maximum of two sectors to be specified. The directives are used for the configuration, and it is also difficult to imagine when having more than two partitions could further improve the performance of a single application. Thus, only partitioning schemes with two partitions are considered.

A partitioning is defined by the capacities c_0 and c_1 of sector 0 and sector 1 respectively. The number of cache misses given a partitioning $C = (c_0, c_1)$ and isolated data structure D can be estimated for a set-associative (pseudo-)LRU cache from the reuse distances $RD_D(x)$ and $RD_{\bar{D}}(x)$ of the split traces:

$$\text{miss}(C, D) = \sum_{RD(x) \geq c_0} RD_D(x) + \sum_{RD(x) \geq c_1} RD_{\bar{D}}(x) \tag{3.1}$$

The optimal cache partitioning w.r.t. the defined data structures is the partitioning that minimizes the total number of cache misses:

$$(C, D)_{opt} = \arg\min_{C,D} \left(\text{miss}(C, D)\right) \tag{3.2}$$

Note that one of the valid configurations is the configuration without sector cache partitioning: $C = (c_{max}, 0)$ and $D = X$, $\bar{D} = \emptyset$.

3.5 Metrics

Metrics are required for three different purposes in this thesis: to measure program locality, effects due to the sector cache and the success of the tool-based approach. Program locality is measured using the stack distance of memory references as explained in section 2.4. The effect of the sector cache is measured based on cache misses, cache miss rate, utilized bandwidth and the speedup due to the sector cache. The *cache miss reduction* (difference in cache misses) with and without the sector cache optimization is used as a metric to measure the success of a sector cache configuration. The tool's predictiveness is evaluated by comparison of the predicted and the measured number of cache misses. The total numbers of misses are not of interest as long as they are high enough to be meaningful. Thus, the *relative error* is chosen as metric to measure the predictiveness. These metrics are obtained by instrumentation of programs, as explained in section 4.1.

The relevant performance monitoring events and derived metrics for measuring cache performance are taken from the *A64FX Microarchitectural Manual* [Fuj21a] and listed below. Due to an error in the performance monitoring unit of the A64FX, correction terms must be applied to some of the L2 cache events. They are listed in the *A64FX PMU Events Errata document* [Fuj20a] and were applied.

Performance Monitoring Events

- L1D_CACHE_REFILL / L2D_CACHE_REFILL:
  Number of L1D/L2 cache misses
- L1D_CACHE_REFILL_DM / L2D_CACHE_REFILL_DM:
  Number of L1D/L2 cache misses (demand)
- L1D_CACHE_REFILL_PRF / L2D_CACHE_REFILL_PRF:
  Number of L1D/L2 cache misses (prefetch)
- L2D_CACHE_WB:
  Number of writebacks from the L2 cache
- EFFECTIVE_INST_SPEC:
  Number of committed instructions instructions

Metrics

- **Bandwidth** = (L2D_CACHE_REFILL + L2D_CACHE_WB) * 256 / time
- **Cache miss rate** = *(cache misses)* / EFFECTIVE_INST_SPEC
- **Speedup** $S = t/t_{SC}$ is the ratio of wall-clock run times without using the sector cache and using the sector cache
- Relative **cache miss reduction**: $1 - (\textit{cache misses})_{SC} \,/\, (\textit{cache misses})$
- **Relative error** $= |\, 1 - (\textit{cache misses})_{pred} \,/\, (\textit{cache misses})_{meas} \,|$

4 Implementation

This chapter describes the implementation of the tool designed to detect code regions where the sector cache can improve cache behavior. The tool can be viewed in three parts: profiling, stack processing, and reuse distance profile analysis. Profiling and stack processing are implemented in C++ and based on the *pindist* tool [WB16] by Weidendorfer and Breitbart. The implementation of each of the three parts is described in the following sections. Profiling is also required to measure the impact of the sector cache on cache behavior.

4.1 Profiling and Instrumentation

Gathering information about a program's behavior is called *profiling* and achieved by *instrumentation*. Instrumentation is an essential part of software analysis and performance optimization. Typically, measurement probes are inserted into application code, often augmented by the additional counting of hardware performance events. The actual measurement occurs during execution of the instrumented binary and the measured metrics are recorded. Binary instrumentation, in contrast to source instrumentation, does not require any modification of the source code [DFF+03, HW10]. Instrumentation is performed for two distinct purposes in this thesis: to obtain the reuse distance profile of an application and for the measurement of performance events on the A64FX.

- **PAPI** [MBDH99] is a source instrumentation library that enables transparent reading of hardware performance counters across different architectures. To count the number of performance events that occur while executing a section of code on the A64FX, the associated hardware performance counters are measured before and after the section.
- The **Intel PIN** [LCM+05] binary instrumentation framework is used for the reuse distance profile generation and explained below.

4.1.1 Intel PIN

PIN is a dynamic binary instrumentation framework that supports the IA-32, x86-64 and MIC ISAs. Many of Intel's profiling and analysis tools use PIN. Tools created with PIN are called *pintools* and can be written in C, C++ or assembly. PIN is dynamic because it can instrument the binary code *just-in-time* (JIT), right before the code is executed. User-defined instrumentation code (*analysis routines*) can be inserted at different levels of granularity: instruction, basic block, function and image granularity. The pintool programmer defines where analysis routines are inserted using *instrumentation routines* of the PIN library [Int21].

PIN can run in two different modes: *JIT mode* and *probe mode*. In probe mode, the application and the replacement routines are run natively. This can improve performance compared to JIT mode, but instrumentation is possible only at image and function granularity. JIT mode, on the other hand, allows instrumentation at finer granularity. The input to

PIN is the application binary image, but the actual application's instructions never get executed in JIT mode. Instead, the application runs in a virtual machine (VM) where the *JIT compiler* repeatedly fetches chunks of code from the binary to generate new instrumented code from it. The instrumented code is cached for reuse and executed by the dispatcher as illustrated in Figure 4.1 [Int15, Int21].

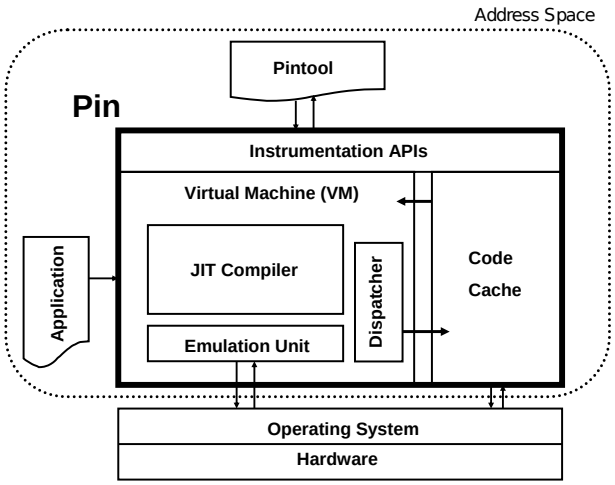


Figure 4.1 – PIN software architecture.

Taken from [Int15, Fig 2.]

The application, PIN library and pintool share the same address space, but do not share any libraries. In particular, each part uses its own copy of glibc [Int15]. PIN also supports multi-threading. Application threads execute instrumented code in parallel and must be synchronized if required. Limited access to the symbol and debug information of the application image is also available [Int21].

4.1.2 Memory Instructions

Instrumentation of memory instructions is required to obtain their addresses as input to the stack processing algorithm. The instrumentation is performed in JIT-mode, because instruction granularity is required. Calls to an analysis routine performing the stack processing are inserted before every memory instruction. Write instructions within the binary, for instance, are identified using the instrumentation routine INS_MemoryOperandIsWritten . The insertion of the analysis routine is performed using the instrumentation routine INS_InsertPredicatedCall. The address and size of the memory access is obtained using the instrumentation routines INS_MemoryOperandCount and INS_MemoryOperandSize.

There are two general approaches: either the whole memory trace is recorded first and processed afterwards (*offline*), or each memory reference is processed while the instrumented application is running (*online*). An **advantage** of offline processing is that the trace can be processed multiple times without recording the trace another time and that recording the trace of individual threads can be performed without thread synchronization. However, the references must be interleaved in a meaningful way to calculate concurrent reuse distance and synchronization points must be inserted and recorded to be able to reconstruct the thread timing. This is one example of a major **disadvantage** of processing offline: all the required

metadata besides the order, address and size of memory references must also be recorded. The trace processing was decided **online**. For performance reasons, however, this decision should be carefully reconsidered. Sampling of references at the cost of accuracy can also be considered to improve performance [SKP10].

Thread Interleaving

When the trace is processed online using multiple threads, they must act on shared global data (hash map and linked list) of the chosen stack processing algorithm (see section 4.2). Their accesses must be synchronized to keep consistency and correctness. The analysis routines and the synchronization can influence the total order of threads. A simple approach would be to not pay any attention to the ordering. But the resulting calculated concurrent reuse distance from the interleaved trace may be meaningless.

In this thesis, an interleaving of *one memory instruction per thread* is chosen. Other interleaving schemes are proposed e.g. in [SPP10]. The interleaving is implemented using a cyclic queue-based spinlock that leads to a round-robin scheduling of the threads when locked and unlocked before and after each memory instruction (critical section). Because PIN does not provide an implementation of the C++ atomic library, the C atomic library is used instead. Applying the lock slows down the instrumented application. For example, using 12 threads, a slowdown of $\approx$ x10 was measured compared to sequential execution.

4.1.3 Data Structures

Only heap-allocated data structures above a certain size (5KB) are considered, as reasoned in section 3.3. The tool identifies heap-allocated memory by replacing the C standard memory allocation functions such as *malloc* or calloc with an analysis routine. Within the analysis routine, the original function is invoked and the line of code, return pointer and size of each dynamic memory allocation are stored. Whether or not a reference belongs to a particular data structure can be determined by checking whether its virtual address is in the range of the starting address and the starting address plus the size of the memory allocation. When a cache line is shared between two distinct data structures, it is assigned to the first data structure that accessed it. The function replacement is be implemented with PIN by searching the C standard library image for names using RTN_FindByName and replacing the functions using RTN_ReplaceSignature.

4.1.4 Program Phases

As described in section 3.2, each function is assigned a program phase. When loading the application image via PIN, the application image sections are searched for functions and each function is instrumented by inserting calls at the entry and exit of the function (see Listing 4.1).

Storage is allocated for reuse distance counts of each code region. When a code region enters or exits, a snapshot of the current global reuse distances is taken. The difference in global reuse distance counts before and after the region is the number of accesses made within the region and their associated reuse distances. The differences are accumulated at each region exit.

A region can be traversed multiple times and include other regions. For example, the main function includes all other regions. In case function-granularity is inappropriate, it is possible to create a custom code region by marking the begin and end in the source code.

```
if (IMG_IsMainExecutable(img))
// loop over all sections in the image:
for(SEC sec = IMG_SecHead(img); SEC_Valid(sec); sec = SEC_Next(sec)){
// skip non-executable sections:
if (SEC_TYPE_EXEC != SEC_Type(sec)) continue;
// loop over all functions in the section:
for(RTN rtn = SEC_RtnHead(sec); RTN_Valid(rtn); rtn = RTN_Next(rtn)){
      RTN_InsertCall(rtn, IPOINT_BEFORE, ...); // on entry
      RTN_InsertCall(rtn, IPOINT_AFTER, ...);  // on exit
} }
```

Listing 4.1 – Searching an application image and inserting analysis routine calls to other functions using PIN.

4.2 Stack Processing Algorithm

The exact reuse distance of memory references might not be always of interest. To be able to infer if a memory access is a cache miss, it is sufficient to determine if its reuse distance is below or above the cache capacity. The markers algorithm, described in [ACP02], can be modified to efficiently determine a range of the reuse distance of a memory access instead of its exact value, as shown in [KHW91]. The resolution of the resulting reuse distance histogram decreases, but as an advantage and in contrast to most other stack processing algorithms, the time complexity of the count phase is independent of the memory trace's locality and only proportional to the constant resolution R. The time complexity for the stack processing of the whole trace is then proportional to the length of the trace times the resolution ($\mathcal{O}(|T|\cdot\mathrm{R})$), because the search phase and update phase can be performed in $\mathcal{O}$ operations (see subsection 4.2.2). Exact tree-based algorithms, such as [Olk81], have time complexity of $\mathcal{O}(|T| \cdot logD)$, where D is the (locality-dependent) mean stack distance.

In the modified markers algorithm of [KHW91], the space of reuse distances is divided into a sequence of n ranges (buckets). Each bucket B_i is defined by its minimum reuse distance min_i. min_{i+1} is the minimum reuse distance of the bucket with the next higher reuse distance B_{i+1} in the sequence. All accesses with reuse distance $min_i \leq RD(x) < min_{i+1}$ are assigned to bucket B_i. The last bucket in the sequence has infinite minimum reuse distance and accounts for compulsory misses. Algorithm 2 shows pseudo-code of the algorithm described in [KHW91]. The algorithm was also implemented in [WB16] and the implementation of the authors is used as base for the implementation in thesis.

4.2.1 Data Structure Interference Detection

To detect data structure interference, Algorithm 2 is applied to the split trace of each considered data structure. Two stacks for each data structure are maintained instead of a single stack. This essentially simulates the cache behavior of a partitioned fully associative LRU cache for the isolation of each data structure. The extension is shown in Algorithm 3 as pseudo-code. For performance reasons, accesses referencing the same cache line as the previous reference (zero reuse distance) are ignored. They do not alter the stack because

Algorithm 2 Markers algorithm with reuse distance buckets

```
1: stack ← ∅
2: markers[...] ← ∅
3: buckets[...] ← 0
4: next_bucket ← 1
5: for all memory references x in T do
6:     if x ∉ stack then
7:         INSERT_TOP(stack, x)
8:         if stack.size = markers[next_bucket].distance then
9:             markers[next_bucket].position ← stack.end          ▷ activate new marker
10:            next_bucket ← next_bucket + 1
11:        end if
12:        increment_buckets(markers.position)    ▷ stack elements sliding above marker
13:        buckets[∞] ← buckets[∞] + 1                                  ▷ (count phase)
14:    else
15:        REMOVE(stack, x)
16:        INSERT_TOP(stack, x)
17:        buckets[x.bucket] ← buckets[x.bucket] + 1  ▷ read bucket of access (count phase)
18:        x.bucket ← 0
19:    end if
20: end for
```

the reference's cache line must be already on top of the stack and they do not contribute to cache misses.

The output of Algorithm 3 is the reuse distance buckets of the split traces for each data structure and is stored with additional metadata in *csv*-format. The buckets minimum distances are set according to the possible cache partition capacities of the L1D and L2 caches of the A64FX.

4.2.2 Search Phase

The search phase is implemented using the unordered hash map of the C++ standard library with the block addresses of the references as the key and the iterators within the LRU stacks as values. When a reference is made, the hash map is searched for the block address, and if found, its iterators are retrieved as a vector. A new entry in the map is made if it was not found. Given enough hash buckets, hash table access and update require $\mathcal{O}(1)$ operations. The number of necessary hash buckets can be approximated with $|X|$, the number of distinct references in the trace [ACP02].

Since only the block address of the references is used as the key, and the block offset is masked, the last bits of the keys are always zero. This leads to a poor mapping of the hash function to hash buckets using the default C++ hash function for pointers, because the keys are always multiples of the offset length. Therefore, the hash function is replaced by a user-defined hash function which first shifts the reference's block address to the right by the number of bits of the offset length. This significantly speeds up the table lookup compared to using the default hash function. The authors of [KHW91] use the hash function $bucket = (block\ address) MOD (hash\ table\ size)$.

Algorithm 3 Extension to the markers algorithm for data structure interference detection.

```
stacks_D[...] ← ∅
stacks_D̄[...] ← ∅
[...]                    ▷ same as Algorithm 2 line 2-4 with separate variables for D and D̄

last ← 0
for all memory references x ∈ T do
    if last = x then     ▷ x maps to the same cache line as the previous reference
        continue
    end if
    last ← x
    for all data structures D do
        if x ∈ D then
            [...]        ▷ same as Algorithm 2 line 6 with stack = stacks_D[D]
        else
            [...]        ▷ same as Algorithm 2 line 6 with stack = stacks_D̄[D]
        end if
    end for
end for
```

4.3 Reuse Distance Profile Analysis

During analysis, a recommended sector cache configuration and isolated data structure is calculated from the reuse distance profile for each code region. To identify the symbol name of the data structure, it is annotated with the file name and line number where the memory is allocated in source code. In addition, the predicted cache miss reduction and number of cache misses using the recommended sector cache configuration as well as the predicted number of cache misses without using the sector cache are calculated. The best sector cache configuration for each code region is found by a simple brute-force search on the reuse distance profile for all possible configurations. A reuse distance histogram can be created for manual inspection of the locality properties (see Figure 5.5 for an example).

The recommendation is then used to modify the source code accordingly. Whether the configuration results in better cache behavior and whether the measured cache metrics are close to the predicted values is evaluated by profiling the modified program on the A64FX.

5 Evaluation

The sector cache on the A64FX is a relatively understudied hardware feature. In order to gain a better understanding of specific hardware behaviors using the sector cache, isolated from the noise normally introduced in real programs, a set of minimal artificial computational kernels (microbenchmarks) is first developed. The kernel's source code is instrumented to measure relevant occurring performance events and the obtained profile is analyzed.

The developed tool serves to indicate opportunities for the application of the sector cache in real-world programs. The NAS parallel benchmarks [BBB+91] were decided to evaluate the tool and to furthermore the sector cache. There is an implementation of PIN for ARM ISAs [HK06], but to the best of our knowledge it is no longer supported. Therefore an x86-binary serves as input to the pintool to record the reuse distance profiles of the benchmarks. If the analysis indicates an advantage using the sector cache, the source code is modified accordingly. The original and modified source code are instrumented and compiled using FCC to be run on the A64FX. The resulting performance metrics of the modified code are then compared against the tool's predictions and the performance metrics of the original code.

5.1 Experimental Setup

Two systems are used in this work: the A64FX processor is used for measurements concerning the sector cache and the x86-based 64-core AMD Epyc 7742 processor is used to record reuse distance profiles. Because we wanted to exclude NUMA-effects, we used at most 12 cores because there are 12 cores grouped in each of the four available CMGs on the A64FX. Thread pinning and placement is enforced using OpenMP [DM98] by setting the environment variables *OMP_PROC_BIND=close* and *OMP_PLACES=cores* on both systems. On the A64FX, they must be set *close* and *cores* anyways, otherwise the L2 sector cache becomes ineffective in parallel execution using the Fujitsu sector cache library.

Microbenchmarks are repeated such that each measurement takes at least one second and the number of repetitions, total run time as well as the performance events that have occurred are recorded. The memory allocations are aligned at page boundaries.

Reuse distance profiles are recorded using 1 and 12 threads, but only once because of the instrumentation overhead.

NAS parallel benchmark measurements are repeated 10 times and the mean and standard deviation of the run time as well as the performance events that have occurred are recorded.

A64FX Setup

The sector cache measurements are performed on the A64FX processor. The binaries for execution on the A64FX are compiled using FCC 4.5.0 in "*trad mode*" us-

ing the compiler flags *-Kfast -Kopenmp -Kocl*. *-Kfast* sets the highest possible optimization level and also enables the HPC tag address override. *-Kocl* must be set to configure the Fujitsu HPC extensions using FCC, otherwise the compiler ignores the Fujitsu-specific compiler directives (OCLs). *-Kopenmp* is set to use the Fujitsu OpenMP library for multi-threading. However, even for single-threaded execution *-Kopenmp* must be set for the L2 sector cache to become effective. This is not documented in the FCC manual [Fuj21b] and found by experiment. In addition, the environment variables *FLIB_HPCFUNC=true* and *FLIB_SCCR_CNTL=true/false* are set to enable and disable the sector cache function. The *Fujitsu huge page library* [Fuj21b] with a page size of 2MiB is used by setting the environment variable *XOS_MMM_L_HPAGE_TYPE=hugetlbfs*. PAPI v6.0.0.4 is used to access the hardware performance counters.

AMD Epyc Setup

The binaries for the reuse distance profile recording are compiled using gcc v7.5.0 using the compiler flags *-O1 -g -fnoinline*. The optimization level is set to *-O1* to prevent the compiler from merging dynamic memory allocations. Otherwise, memory allocation code lines retrieved by PIN may differ from the line in source code. *-g* and *-fnoinline* are required to access debug symbols and to prevent the compiler from function inlining. The function entry and exit of inlined functions cannot be instrumented using PIN. The pintool is linked with the PIN library v3.19 for the binary instrumentation.

5.2 Microbenchmarks

To better understand how much performance can potentially be gained under which circumstances using the sector cache, we performed measurements on two artificial kernels. They represent the two simplest cases we can think of where the sector cache can contribute to a speedup of the application:

1. Access reusable data, **then** access non-reusable data long enough to evict most of the reusable data from cache in a loop (*Kernel1*).
2. Access reusable data **while** accessing non-reusable data in a loop (*Kernel2*).

The access pattern is chosen sequential in both cases to enable autovectorization, spatial locality and hardware prefetching. This resembles the typical access pattern of many scientific applications. Other results may be observed when there is indirection in the accesses or when the access pattern (e.g. random) cannot be predicted by the prefetcher. The effect of vectorization is discussed at the end of this section using Kernel2 as an example.

5.2.1 Kernel1 - Alternating Access

Kernel1 accesses two arrays a and b alternately in an inner loop. a is reused in every iteration of the outer loop. b is a large array with streaming access and never reused (see Listing 5.1). The size of a (na) is chosen in the order of magnitude of the largest possible L1D or L2 cache partition capacity respectively. This is expected to have the maximum positive effect when a is isolated. The L1D partition size is 3 cache ways (48KiB) and the L2 partition size is 14

cache ways (7MiB). The number of subsequent accesses to b (nb) is varied from few accesses up to a number of accesses large enough to evict most of a, if a is not isolated in a partition separate from b.

```
#pragma procedure scache_isolate_assign a
for (int iter = 0; iter < NITER; iter++) {
    // access a
    for (int i = 0; i < na; i++) a[i] += a[i];
    // evict a from cache by accessing non-reusable b
    for (int i = 0; i < nb; i++) b[iter*nb + i] += b[iter*nb + i];
}
```

Listing 5.1 – Kernel1: alternating access to reusable data (a) and non-reusable data (b).

Expected Behaviour

The sector cache is expected to improve performance when $na + nb > C$. To be specific: when $nb >$ 16KiB for the L1D cache and when $nb >$ 1MiB for the L2 cache. Leaving out prefetching, there are two contrary effects: the amount of data from a that is driven out of the cache before a gets reused increases with nb. This leads to an increase of the advantage by isolating a in a sector with increasing nb. On the other hand, the time where a is unused also increases with nb, compensating the positive effect of isolating a at some point for large enough nb. This is expected to lead to a peak in the speedup and diminishing returns for $nb \to \infty$.

Performance

Figure 5.1 shows the performance and cache miss rate due to prefetching of Kernel1 for three different values of na without sector cache and when isolating array a, as well as the resulting speedup due to the sector cache. The L1D cache is shown on the left, the L2 cache is shown on the right.

The **L1D** cache behaves as expected: from $nb >$ 16KiB, the performance without sector cache decreases because cache misses begin to occur when $na + nb > C$. This is where the sector cache starts improving performance. The speedup reaches a peak at $nb =$ 30KiB with a maximum value of $S_{L1D} = 1.449$ when na is equal to the sector size. When na is above or below the sector size, the advantage decreases.

The **L2** cache behaves different: the performance without using the sector cache even increases when more cache misses occur at $na+nb > C$. At this point, as can be seen from the cache miss rates, the prefetcher starts to recognize the streaming access pattern to a and b and the demand miss rate vanishes. The performance using the sector cache, on the other hand, remains at a consistently high level regardless of nb. The effect on performance due to the sector cache can only be observed for values of nb where the prefetch mechanism constantly detects the streaming access pattern and thereby avoids demand misses. This is the case from $nb >$2000KiB where a maximum speedup of $S_{L2} = 1.025$ is observed when na is about equal to the sector size (7100KiB). If na is just over the sector size at 7400KiB, performance is even worse when using the sector cache. Conversely, if na is significantly larger than the sector size at 7700KiB, the sector cache will not improve or degrade performance. Observing the cache miss rates reveals the reason. If na is just slightly larger than the sector size, the

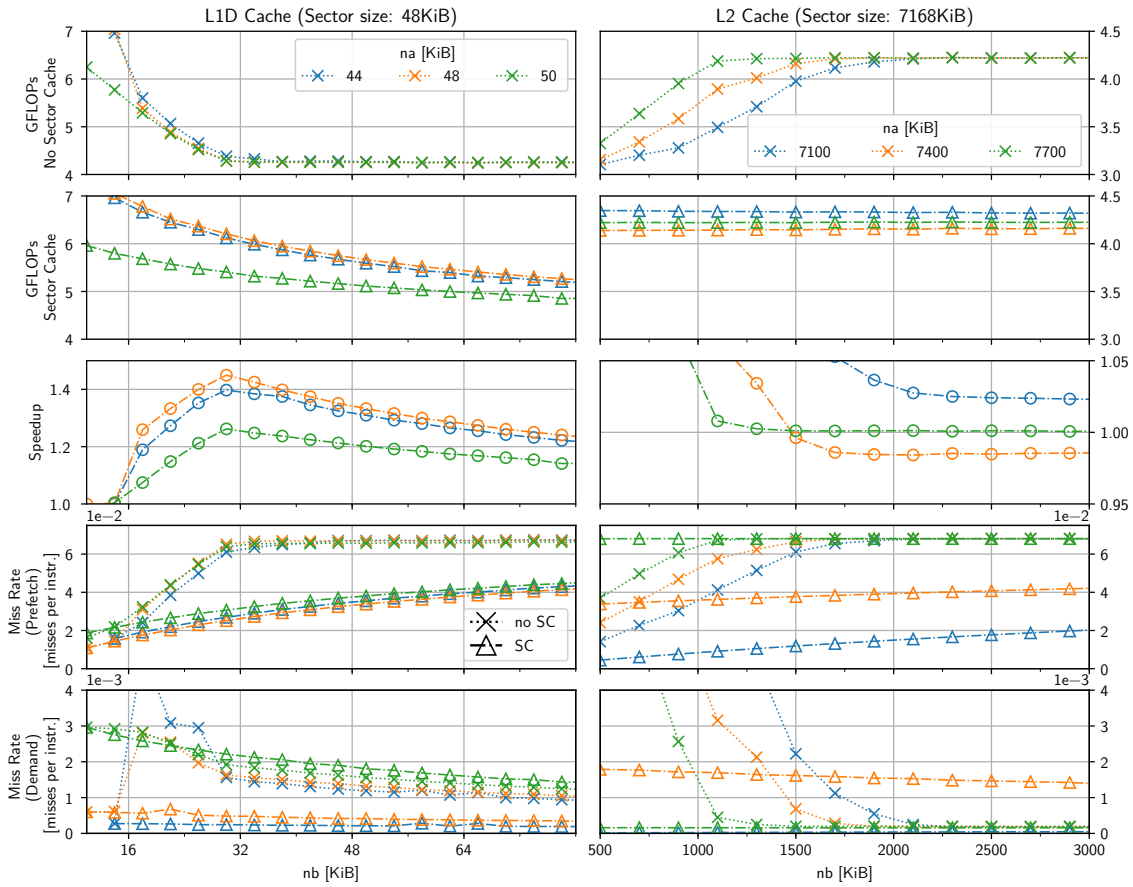


Figure 5.1 – Performance, speedup, and cache miss rate of Kernel1 without using the sector cache and using the sector cache with maximum partition sizes for the L1D cache and L2 cache. The size of the reusable data a is set close to the partition size.

prefetcher is not fully effective and does not completely avoid demand misses. However, if na is significantly larger than the sector size, the prefetcher becomes fully effective.

Summary

1. Maximum measured speedup for alternating access to reusable and non-reusable data (Kernel1) was $S_{L1D} = 1.449$, $S_{L2} = 1.025$.

2. Best speedup is achieved when the size of the reusable data matches the sector capacity and its size should not be above the sector capacity.

5.2.2 Kernel2 - Simultaneous Access

Kernel2 accesses two arrays – a and b – simultaneously in an inner loop (see Listing 5.2). The whole array a and a section of b is accessed in each iteration of the outer loop. The sections of b are never reused and each section has the same size as a. The size of a is chosen in the order of magnitude of the L1D and L2 cache capacity respectively. Array a is isolated

in a sector of varying capacity. Array b does not fit into any of the cache levels ($nb = 0.5$GB) and must be fetched from memory. But it is still small enough to avoid NUMA-effects.

```
#pragma procedure scache_isolate_assign a // isolate a
for (int iter = 0; iter < NITER; iter++) {
/* #pragma omp for */
    for (int i = 0; i < na; i++) {
        a[i] += a[i];
        b[iter*na + i] += b[iter*na + i];
}   }
```

Listing 5.2 – Kernel2: simultaneous access to reusable data (a) and non-reusable data (b).

Expected Behaviour

Let us assume a fully associative LRU cache with capacity C without partitioning. In this case, data from a will not be replaced with data from itself or data from b as long as $na \leq C/2$ holds since data from the arrays is loaded in equal shares in each iteration. An increase in the cache miss rate is expected when $na > C/2$ without the sector cache. Using the sector cache, the cache miss rate is expected to increase once the size of a exceeds the sector capacity.

Sequential Performance

Figure 5.2 shows the performance and cache miss rates due to demand misses and prefetch misses of Kernel2 for several partitioning schemes in sequential execution and in dependence of the size of a. The x-axis shows the size of a and is chosen in units of cache ways. na is also referenced in units of cache ways in the following ($C_{L1D} = 4$, $C_{L2} = 16$).

Let us first consider the kernel's behaviour **without** sector cache. The **L1D** cache behaviour is as expected: when $na \approx 2$, the demand miss rate increases and the performance decreases. The prefetcher, which was already active for array b, becomes active for array a as well and the prefetch miss rate doubles. However, the demand miss rate does vanish completely. The **L2** cache behaviour differs from expectation: the demand miss rate increases already once $na > 6$ holds. This indicates a different replacement policy than LRU. Performance drops by up to 20% once $na > 6$, but as soon as the prefetcher becomes fully effective, the demand miss rate vanishes and performance returns to its previous level.

The **L1D** sector cache can extend the high performance level up to $na = 3$ using 3 cache ways. The **L2** sector cache has no considerable effect on performance in sequential execution. Nevertheless, the cache miss rate can be kept at a lower level for larger na.

Parallel Performance

The inner loop of Kernel2 is parallelized to measure the impact on performance due to the sector cache when contention on shared resources increases. Figure 5.3 shows the performance, speedup and bandwidth of Kernel2 using multiple cores.

The performance behaviour using 2 cores is the same as for single-threaded execution. The "speedup" peaks using 1-2 cores result from prefetching, as previously discussed. However, using 4 cores, the bandwidth is saturated once a must be fetched from memory and the sector cache can improve performance. The effect increases with the number of active cores up to 8 cores. Using more than 8 cores does not improve performance or speedup any further

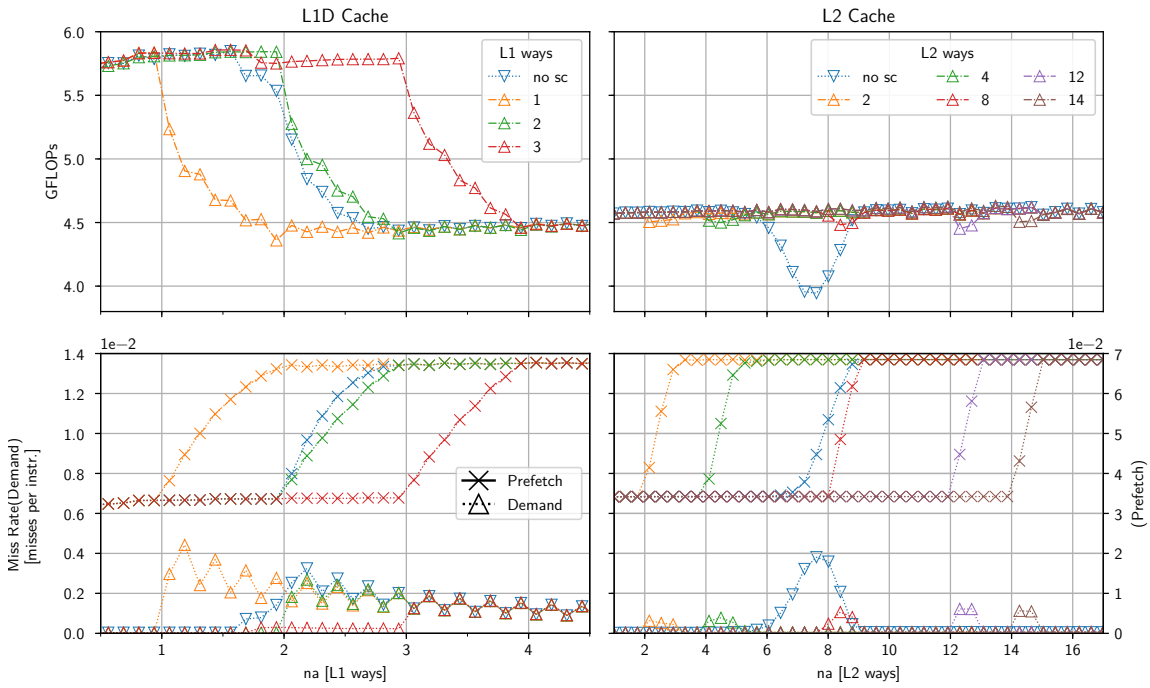


Figure 5.2 – Performance and cache miss rate of Kernel2 in sequential execution for varying partition sizes. The x-axis is in units of cache ways (L1D cache has 64KiB and 4 ways; L2 cache has 8MiB and 16 ways).

because the bandwidth is already saturated using 8 or more cores, even when a can be kept in the L2 cache.

Summary

1. Maximum measured speedup due to the L1D sector cache in serial execution with simultaneous accesses to reusable and non-reusable data was $S_{L1D} = 1.23$.

2. The L2 sector cache does not improve performance for Kernel2 in sequential execution using the huge page library.

3. The performance can benefit most from the L2 sector cache in memory-intensive parallel kernels because the L2 sector cache reduces the amount of contention to bandwidth.

4. Maximum measured speedup in parallel execution with simultaneous accesses to reusable and non-reusable data was $S_{L2} = 1.95$ using at least 8 cores.

Vectorization

In this experiment, an autovectorized and unvectorized version (compiled with the additional flag *-Knosimd*) of Kernel2 are compared. The size of the reusable data is set to values that were proven to result in a high speedup during the previous experiments. The size of a (44KiB and 6000KiB) is set slightly below the chosen partition sizes for a (48KiB and 6MiB). Because each core has a private L1D cache, the size of a is scaled with the number

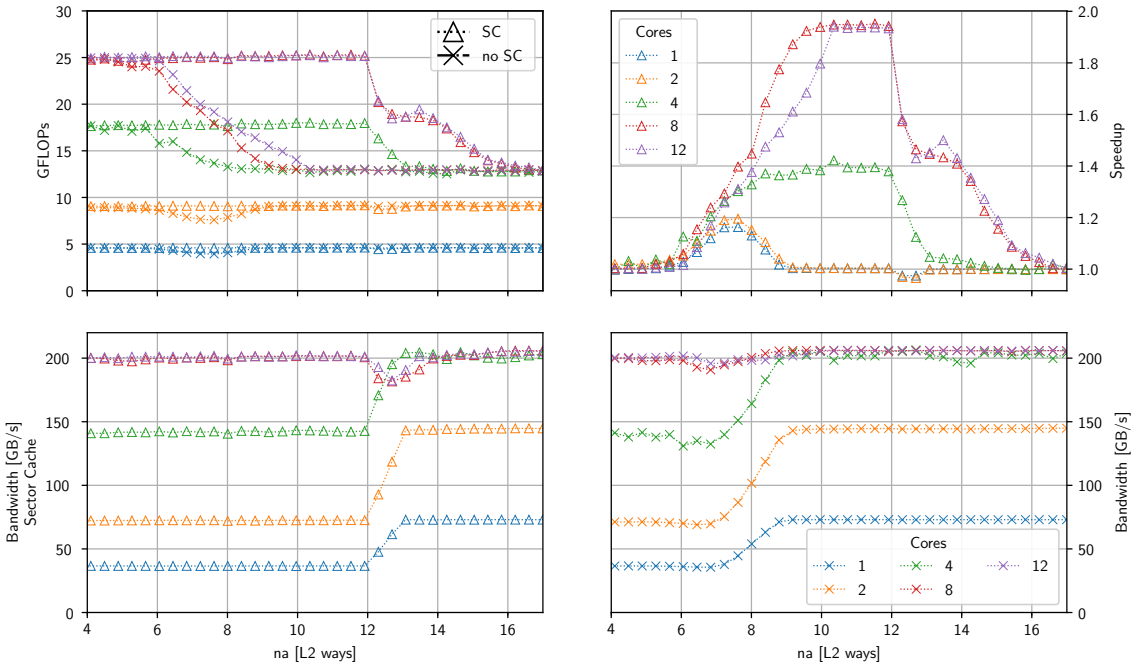


Figure 5.3 – Performance, speedup and bandwidth of Kernel2 for varying number of active cores without using the sector cache compared to using a sector partition size of 12 L2 cache ways for a. The x-axis is in units of cache ways (L2 cache has 8MiB and 16 ways).

of active cores for the L1D measurement. To avoid bandwidth saturation effects during the L1D experiment using multiple cores, nb is set to 6000KiB. That way, a and b both fit into L2 cache. False sharing is avoided by data alignment.

Figure 5.4 shows the sector cache speedup and cache miss reduction for the L1D cache on the left and for the L2 cache on the right. The number of cache misses (not shown in the graph) and the cache miss reduction does not depend on the vectorization. The speedup for the **L1D** cache is roughly independent of the number of active cores, but higher using vectorized code. However, the **L2** speedup of the unvectorized code is negligible, except once the contention to the bandwidth is high using 12 cores. In contrast, the performance of the vectorized code does profit from the L2 sector cache using 3 or more threads. The maximum speedup of factor 2 makes sense: if the contention to the bandwidth is the bottleneck and the cache miss reduction is 50%, the performance can double. The conclusion is that code must be “fast” (memory-intensive) to improve the performance with the L2 sector cache.

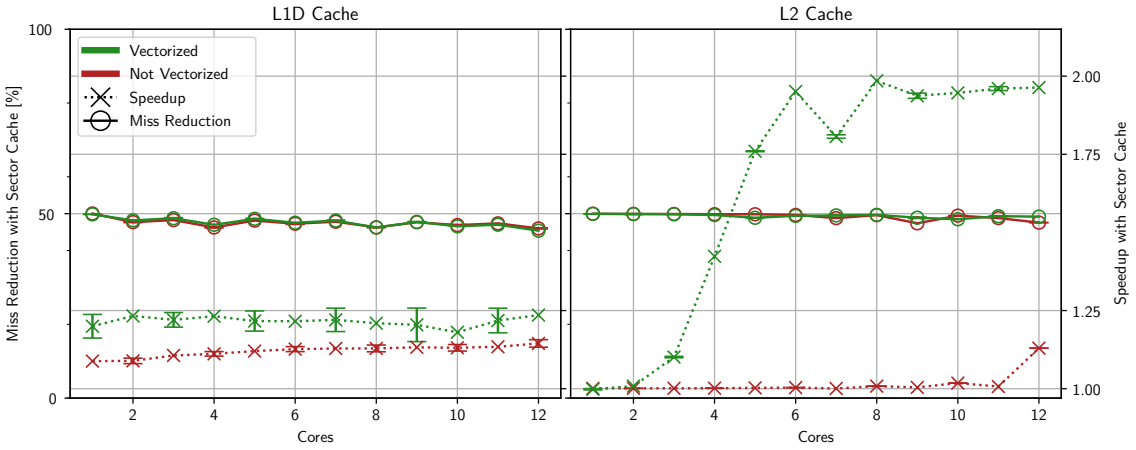


Figure 5.4 – Speedup and cache miss reduction due to sector cache optimization of Kernel2 for vectorized code to unvectorized code with the same sector cache configuration for varying number of cores

5.3 Evaluation of the Tool-based Approach

To evaluate the tool-based approach, we performed a proof of concept on the Dense Matrix Transposed Vector Multiplication (DMTVM) from [AML+21] and selected the NAS parallel benchmarks for further evaluations. The NAS benchmarks are well accepted among other scientists and they offer a wide spectrum of typical HPC applications. The original NAS benchmarks are written in Fortran. Because the tool was designed to identify data structures by replacing the C standard dynamic memory allocation functions, a fairly recent C++ implementation [LGM+21] was chosen. The implementation supports serial execution and, among other parallel programming models, an OpenMP implementation. The working set data can be allocated with the C dynamic memory allocation functions without further modification of the code. Since the profiling involves a huge overhead, the iteration count was decreased to achieve shorter run times during profiling. This does not change the locality of the applications.

Eight benchmarks are included in the NAS parallel benchmarks: BT, CG, EP, IS, LU, FT, MG, SP, and seven different working set sizes: S, W, A, B, C, D, E (ascending order). We instrumented the benchmarks with working set sizes from A up to C and evaluated them, using the recommended sector cache configuration. Class S and W have very small working set sizes and Class D and E have a memory requirement exceeding the available memory (32GB) of a single A64FX compute node for some of the benchmarks. The EP (Embarrassingly Parallel) and IS (Integer Sort) benchmarks showed no temporal locality. The tool also did not predict any considerable advantage using the sector cache for the BT (Block Tri-diagonal solver) and SP (Scalar Penta-diagonal solver) benchmarks. For the FT (3D fast Fourier Transform) benchmark, the tool predicted $\approx 10\%$ L1D cache miss reduction using the sector cache, but the measurements showed no effect. Those benchmarks are not further discussed.

As shown in the previous section, the benefit of the sector cache strongly depends on the working set size. When the tool detects a code region where the sector cache may improve performance for one input size, it most likely cannot be applied to other input sizes. The

same applies to the parallelized OpenMP versions, since reuse distances may change due to parallelization. The benchmarks with their corresponding class where the tool detected beneficial code regions are discussed in this section. Most benchmarks and problem classes showed no benefit from using of the sector cache. However, the tool has indicated that the sector cache brings a benefit in the benchmarks **CG**, **MG** and **LU**. They are discussed below. The code of the loop responsible for most cache misses is shown if it is not too long.

5.3.1 Proof of Concept

DMTVM was chosen to verify the tool, because the authors of the paper [AML+21] showed that the multiplication performance can be improved with the sector cache for certain sizes of the result vector. It is similar to Kernel2 and simple enough to illustrate the approach.

There are three functions and three data structures in the application: the *main* function, the *init* function where the data structures are initialized, and *dmtvm*, where the multiplication takes place. The multiplication operates on the input matrix m, the multiplicand vector b and stores the result in the vector x. The code of the function *dmtvm* is shown in Listing 5.3. x is reused in every row of the multiplication, m is streamed and never reused and b can be ignored regarding memory accesses, because only one element of b is loaded per row. m interferes with the reuse of x and doubles the reuse distances of accesses to x, because equal amounts of data from x and m are required during the calculation of each row.

```
1 #pragma procedure scache_isolate_assign m
2 /* #pragma omp for */
3     for (int i = 0; i < nrow; ++i) {
4         for (int j = 0; j < ncol; ++j) {
5             x[j] += m[i * ncol + j] * b[i];
6     }}
```

Listing 5.3 – DMTVM code based on [AML+21].

Figure 5.5 shows the output of the profiling as reuse distance histograms. Instead of a single reuse distance histogram, we obtain one reuse distance histogram for each data structure – namely b, x, m and the global address space. Because only the file and line of code of the memory allocation is obtained during instrumentation, the identifier of the variables is inserted manually for the sake of clarity. The number of the double precision floating-point matrix rows is set to 500 and the number of columns is set to 5000. This leads to the sizes of $sizeof(b) = 4KB$, $sizeof(x) = 40KB$ and $sizeof(m) = 20MB$. Thus, x can fit in 3 ways of the L1D cache and m does not fit in the L2 cache. In the *main* function (third row), accesses to the global address space (first column) have only reuse distances higher than the L1D capacity. The accesses with $RD(x) = \infty$ are attributable to the *init* function (second row) and are compulsory misses. Accesses with other reuse distances occur during the multiplication (first row). Isolating b (second column) does not change the reuse distances and can be omitted as option. Isolating x (third column) shifts the reuse distances of accesses to x (blue) below the L1D capacity between $32KiB \leq RD(x) < 48KiB$. Other accesses (orange) are not negatively affected. This is exactly as expected, because x is reused frequently and $sizeof(x) = 40KB$. The isolation of m (fourth column) shifts the reuse distances of accesses to its complementary data structure below the L1D cache capacity.

The output of the analysis is shown Table 5.1. Again, identifiers are inserted manually for clarity. The optimal sector cache configuration is the isolation of m in the smallest possible

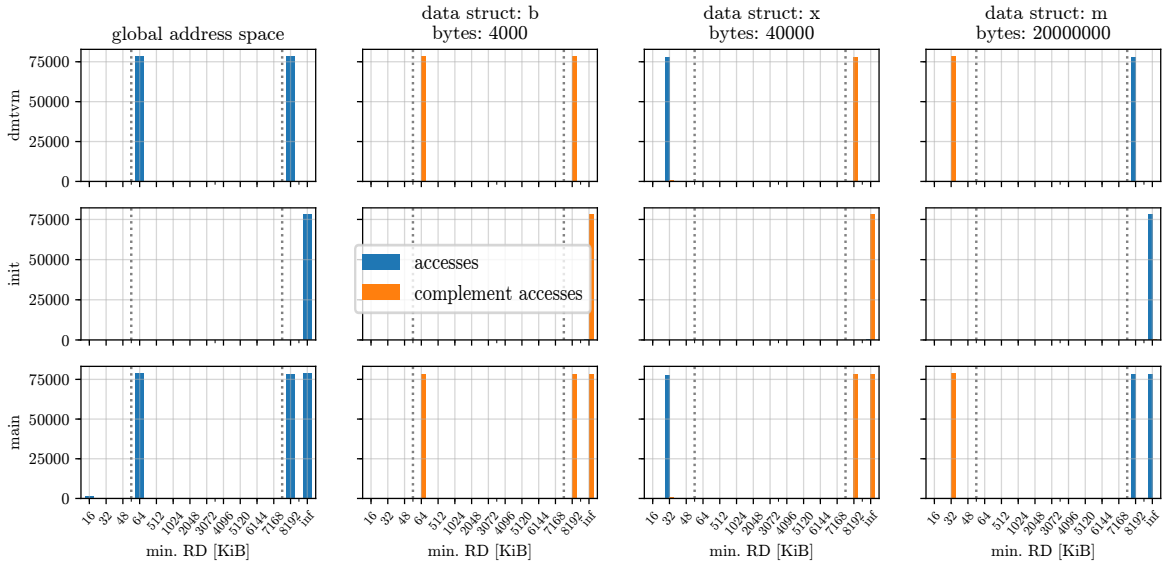


Figure 5.5 – Reuse distance histograms of the DMTVM data structure split traces.

Table 5.1 – Tool's output of the sector cache configuration for DMTVM. The number of L1D cache misses can be reduced significantly by isolating m in the function *dmtvm* in a partition with quota of one cache way.

region	data struct	cache level	nways	misses sc	misses nosc	reduction [%]
main	m	1	1	156824	236875	33.79
main	m	2	2	156583	157742	0.73
dmtvm	m	1	1	78447	157277	50.12
dmtvm	m	2	2	78237	78450	0.27

sector of each cache level and halves the L1D misses when applied. This is again as expected, because half of the accesses are made to x which can be reused when it is isolated from the interfering accesses to m. Isolating m in a small sector slightly decreases the cache misses compared to isolating x in a sector of appropriate size, because it has a minor positive affect on the reuse of b. The configuration has been applied and the measurements of the L1D and L2 misses on the A64FX are in line with the predictions of the tool. The measured L2 misses in the *dmtvm* function remained unchanged at around 80000 when the sector cache was applied. On the other hand, the measured L1D misses went down from 142000 to 81000.

Similar results could be observed for the L2 cache partitioning. The reuse distance histogram did not change much in parallel execution, because the threads share the work on the result vector and the entire vector is used in every row. The sequential configuration, in which non-reusable data is isolated in a sector with minimum capacity, can also be applied to the parallel execution.

5.3.2 CG

CG is a conjugate gradient program spending most of its time in the function *conj_grad* on a sparse matrix-vector multiplication: $q = a \cdot p$. q and p are dense vectors and a is the sparse matrix stored in Compressed Row Storage (CRS) format. The code is shown in Listing 5.4. *rowstr* and *colidx* are the row and column index arrays.

```
#pragma procedure scache_isolate_assign a, colidx
for(j = 0; j < lastrow - firstrow + 1; j++){
    suml = 0.0;
    for(k = rowstr[j]; k < rowstr[j+1]; k++){
        suml += a[k]*p[colidx[k]];
  }
  q[j] = suml;
}
```

Listing 5.4 – Sparse matrix-vector multiplication in the function *conj_grad* of the CG benchmark in the NAS parallel benchmarks.

Table 5.2 shows the tool's recommended sector cache configuration of the CG benchmark class C using 1 or 12 threads. It is found that isolating *colidx* and a in a minimal cache space of the L2 cache (2 ways) is the best option. This prevents the reusable vectors q and p from being thrashed by the streaming access to *colidx* and a, and is in line with the findings of other researchers for sparse matrix-vector multiplications e.g. in [LLD+09] and [AML+21].

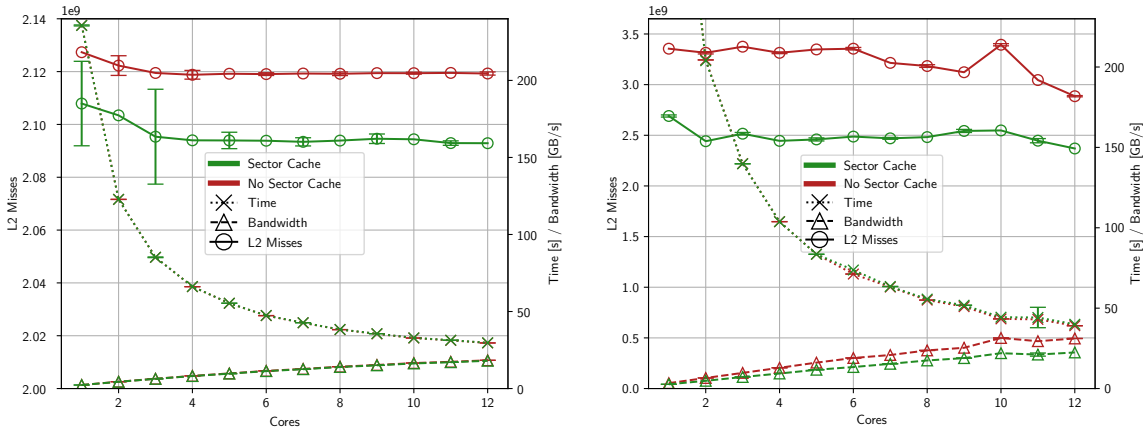
Table 5.2 – Average measured and predicted L2 cache miss reduction using the sector cache in the function *conj_grad* of the CG benchmark Class C.

region	class	cores	data struct	L2 ways	reduction m. [%]	reduction p. [%]
conj_grad	C	1	a, colidx	2	0.92	1.37
conj_grad	C	12	a, colidx	2	1.25	1.23

Performance Results

Figure 5.6a shows the measured L2 cache misses, run time and bandwidth of the CG benchmark with and without using the sector cache optimization and varying number of cores. Note that the y-axis for the L2 misses does not start at zero, so the sector cache effect becomes visible. The predicted and measured L2 cache miss reduction is just $\approx 1\%$ and the run time did not improve – the utilized bandwidth is far below the available bandwidth of the HBM. Nevertheless, the tool's prediction fits the measurement and the sector cache was effective.

Using the same configuration leads to a cache miss reduction in case of the CG benchmark, regardless of the number of threads. The optimization by isolating low temporal locality data in a minimal cache space also applies to multiple threads. Another matrix layout may lead to higher improvements for sparse matrix-vector multiplications as recently shown in [AML+21].



(a) CG benchmark class C

(b) LU benchmark class B

Figure 5.6 – L2 misses, run time and bandwidth using the same sector cache configuration and varying number of cores.

5.3.3 LU

LU is a lower-upper Gauss-Seidel solver, spending most of the time in the function *ssor*. A variety of other functions is invoked in *ssor* and the codes are too long to be shown or analyzed. Applying the sector cache based on the tool's recommendation was quite tedious. The pointers to the heap allocations are passed as function arguments multiple times. They have to be marked as isolated in the invoked function implementations with their corresponding identifier. This is error-prone, when the argument list is long and the identifier within the invoked function does not match the identifier within the calling function.

However, the tool predicts that the number of L2 misses can be reduced in class B and C by 45% and 14% respectively in sequential execution and in class C by by 38% and 17% using 12 threads. For class C, the isolated data structures and optimal sector cache configurations differ depending on the number of cores and input size, as shown in Table 5.3. For class B the optimal sector cache configuration is the same using 1 or 12 threads because, as previously in the CG benchmark, data with low temporal locality is isolated in a minimal cache space.

Table 5.3 – Average measured and predicted L2 cache miss reduction using the sector cache in the function *ssor* of the LU benchmark.

region	class	cores	data struct	L2 ways	reduction m. [%]	reduction p. [%]
ssor	B	1	u, b	2	18.18	45.37
ssor	B	12	u, b	2	17.88	38.24
ssor	C	1	rsd, b	13	8.94	14.29
ssor	C	12	rsd, a	13	-	16.84

Performance Results

Figure 5.6b shows the measured L2 cache misses, run time and bandwidth of the LU benchmark class B with and without using the sector cache optimization and varying number of cores. The optimization is effective, independent of the number of threads. This was not true for the configuration used in class C. In fact, applying the sector cache in class C using 12 threads even increased the number of L2 cache misses. The measurement significantly differs from the predicted cache miss reduction. However, the tool could not make an accurate prediction on the number of cache misses in the first place. These inaccuracies are discussed at the end of this section.

5.3.4 MG

MG is a 3-dimensional multi-grid method on a sequence of meshes. Most of the time is spent in the function *mg3P* in calls to the functions *resid* and *psinv* within a loop. *resid* computes the residual: $r = v - Au$. *psinv* applies an approximate inverse as smoother: $u = u + Cr$. Both functions perform similar stencil operations on the 3-dimensional grids with two inner loops iterating over the x-direction. The second inner loops of the functions do not autovectorize with FCC and are vectorized using a compiler directive. This does not invalidate the computation. The outermost loop iterates over the z-direction and is parallelized. The code for *resid* is shown in Listing 5.5.

```
#pragma procedure scache_isolate_assign u
  double u1[M], u2[M];
#pragma omp for
  for(i3 = 1; i3 < n3-1; i3++){
    for(i2 = 1; i2 < n2-1; i2++){
      for(i1 = 0; i1 < n1; i1++){ // does autovectorize
        u1[i1] = u[i3][i2-1][i1]   + u[i3][i2+1][i1]
               + u[i3-1][i2][i1]   + u[i3+1][i2][i1];
        u2[i1] = u[i3-1][i2-1][i1] + u[i3-1][i2+1][i1]
               + u[i3+1][i2-1][i1] + u[i3+1][i2+1][i1];
      }
      for(i1 = 1; i1 < n1-1; i1++){ // does not autovectorize
        r[i3][i2][i1] = v[i3][i2][i1]
                      - a[0] * u[i3][i2][i1]
                      - a[2] * ( u2[i1] + u1[i1-1] + u1[i1+1] )
                      - a[3] * ( u2[i1-1] + u2[i1+1] );
  }}}
```

Listing 5.5 – 3-d stencil in the function *resid* of the MG benchmark in the NAS parallel benchmarks.

In single-threaded execution and problem size class C, the tool finds that isolating r in 13 L2 ways in the function psinv and u in 13 L2 ways in the function resid significantly reduces the L2 cache misses (see Table 5.4).

Sequential Performance

The total execution time without sector cache was $t = (89.63 \pm 0.14)s$. After applying the sector cache optimization, the run time $t_{SC} = (88.80 \pm 0.22)s$ was reduced on average by $\Delta t = 0.83s$. Even though the total number of L2 cache misses was reduced by $\approx 26\%$, the run

Table 5.4 – Average measured and predicted L2 cache miss reduction using the sector cache in the functions *resid* and *psinv* of the MG benchmark Class C. Using 12 threads, the cache misses are not measured for each function individually, but for the whole function *mg3p*.

region	class	cores	data struct	L2 ways	reduction m. [%]	reduction p. [%]
resid	C	1	u	13	30.71	28.46
psinv			r	13	26.21	45.51
resid	C (scaled)	12	u	13		38.08
psinv			u	3		35.44
mg3p					13.84	

time decreased only by $\approx 1\%$. However, the tool could indicate an appropriate sector cache configuration and predicted the cache miss reduction with high accuracy. Differentiating between program phases was useful, because the data structure that had to be isolated depended on the current function.

Parallel Performance

Simply adding multiple threads using the same sector cache configuration as well as input size had a negative effect on the number of L2 cache misses and run time. Programmers familiar with stencil codes will likely guess, that the reuse distances shift when the outer z-loop is parallelized due to the resulting domain decomposition in z-direction. When the input size of the 3-dimensional grid is scaled in z-direction by the number of threads, the sector cache does indeed decrease the L2 cache misses using the same configuration as used for single-threaded execution. The size in y-direction was divided by the number of threads to avoid memory overcommitment and to keep equal total amount of work. The grid size dimensions in dependence of the number of threads is set to $(512,\ 512/\mathit{threads},\ 512 \cdot \mathit{threads})$.

Without scaling and using 12 threads, the tool indicated no advantage in the sector cache, but it did in the z-scaled MG benchmark. However, the resulting recommended configuration (B) differs from the configuration (A) found by profiling the single-threaded MG benchmark.

Figure 5.7 shows the z-scaled MG performance, bandwidth and run time using multiple cores for config A on the left and config B on the right. Although the number of L2 cache misses and utilized bandwidth was decreased significantly by the sector cache, there was no improvement in run time. Even using all cores of a CMG, the utilized bandwidth is still far below the available 256 GB/s. Configuration A performed better than configuration B using 1 thread – configuration B performed better than configuration A using 12 threads.

However, the MG benchmark showed the relationship between the number of threads, input parameters and the shift in reuse distance due to interference. It should in general be possible to infer a pattern in the shift of reuse distances, depending on the number of threads and input parameters, by applying a fitting function on the record of a few samples of the reuse distance profiles for a varying number of threads and (small) input parameters. It has been shown in [LLD+09] that this is possible, but only for input parameters.

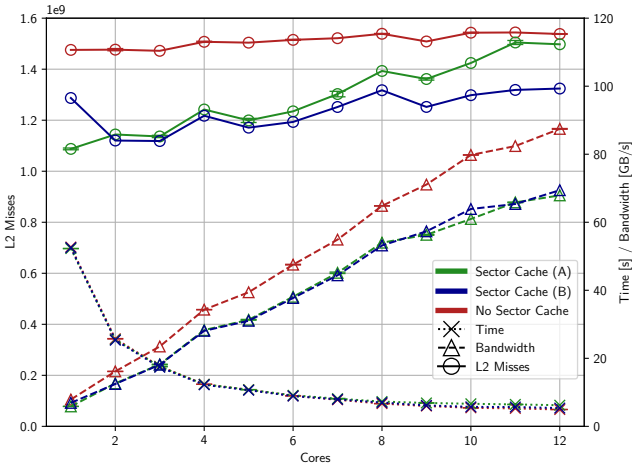


Figure 5.7 – L2 misses, run time and bandwidth of the MG benchmark class C (z-scaled) using the recommended sector cache configuration for 1 thread (A) and 12 threads (B) and varying number of cores.

5.3.5 Tool Accuracy

A key requirement is the tool's ability to make reliable predictions about the approximate number of cache misses. Table 5.5 shows the relative error of the predicted L1D and L2 cache misses compared to the measured performance monitoring events. The tool always predicted less cache misses than actually occurred. For the FT benchmark in particular, the prediction does not reflect the measured cache misses at any cache level. One of the problems is that PIN cannot instrument ARMv8 ISA binaries which is used in the compiled binaries by FCC. An x86 binary had to be compiled with another compiler for the reuse distance profiling. The instructions are different and the compiler's optimizations may also differ. This can change the locality of the application. Another issue is that the tool assumes true-LRU and does not take cache associativity into account. Conflict misses are ignored without modeling the associativity of the caches and the actual replacement policy is likely a PLRU policy. Additionally, stack accesses are ignored during instrumentation, as also previously done in [WB16], but this may not be valid when large stack arrays are used. Prefetching is also not modeled in the tool. Many factors may cause the inaccuracies and they require further investigation.

Table 5.5 – Relative error of the L1D and L2 cache miss predictions compared to the measurements. The total number of L2 misses in EP is too low for the relative error to be meaningful.

bench / class	rel. error L1 [%] A	B	C	rel. error L2 [%] A	B	C
BT	47.3	46.7	46.5	8.1	6.7	11.1
CG	5.8	2.4	1.4	8.0	32.8	37.5
EP	0.1	0.2	0.3	-	-	-
FT	37.8	41.9	44.1	81.2	85.7	86.2
LU	6.8	22.0	11.0	9.4	15.5	7.9
MG	1.7	1.7	35.0	4.1	2.0	0.1
SP	67.8	62.2	64.4	9.6	7.7	11.7

6 Conclusion

Throughout this thesis, we have studied and analyzed the sector cache behavior on the A64FX – one of the fastest HPC CPUs developed so far. We have identified typical use cases where the sector cache can improve performance or keep a high performance level for larger input sizes. Moreover, we have designed a binary instrumentation tool with the ability to record reuse distance profiles of a program's separated data structures. With the help of the tool, we could identify several benchmarks of the NAS parallel benchmark collection that benefit from the sector cache. Furthermore, we showed that the number of cache misses of these benchmarks can be significantly reduced by applying the tool's recommendation for code modifications concerning the sector cache.

It is difficult to improve performance using the sector cache and many codes will likely not benefit from it. To improve performance with the sector cache, the program's code must be memory-intensive in the first place. However, even when the sector cache does not improve run time, its ability to reduce the occurrence of cache misses makes it still a useful feature. The memory subsystem is responsible for a major part of the energy consumption in computing and reducing the number of cache misses can significantly improve energy efficiency.

Two different types of use cases have been identified by the tool: either data with high temporal locality is isolated in a sector of appropriate size, or data with low temporal locality is isolated in a sector of minimal size. The first use case strongly depends on the program's input size and is not independent of parallelization. However, the second use case applies to single-threaded execution as well as to multi-threaded execution, because the reuse distance of accesses to data with low temporal locality will likely not shift below the cache size due to parallelization.

During this work it became clear that in general the recommendations of the developed pintool apply only to the number of threads and input sizes used during profiling. Sampling the reuse distance profiles for varying numbers of threads and input parameters to extrapolate the reuse distance for different numbers of threads and parameters is is needed to make the tool more practical, because the tool's overhead is too high. From a practical point of view, it is cumbersome to use the line of code of a dynamic memory allocation to identify a data structure. Especially when the return pointer is passed through multiple functions, it is very error prone and time consuming to identify which pointer should be isolated. It would be much better if the identifier used in the function is provided directly by the tool. The additional inclusion of a program's debug information can greatly enhance the tool.

Future Work

As a future work, it would be interesting to design an experiment to find the underlying hardware mechanism of the sector cache replacement policy. Also finding more code where the sector cache is beneficial by applying the tool to other benchmarks or compute kernels could be part of future work. However, the performance and predictive power of the tool should be improved in advance. As mentioned, the tool is not very user-friendly in its current state. Improving the user interface or even including such a tool as a compiler feature to generate automatic code transformations for the sector cache would be an interesting task as well.

Acronyms

Listings

List of Figures

Bibliography

[ACP02] George Almási, Călin Caşcaval, and David A Padua. Calculating stack distances efficiently. In *Proceedings of the 2002 workshop on Memory system performance*, pages 37–43, 2002.

[AML+21] Christie Alappat, Nils Meyer, Jan Laukemann, Thomas Gruber, Georg Hager, Gerhard Wellein, and Tilo Wettig. Execution-cache-memory modeling and performance tuning of sparse matrix-vector multiplication and lattice quantum chromodynamics on a64fx. *Concurrency and Computation: Practice and Experience*, page e6512, 2021.

[Arm21] Arm Limited, Kamikodanaka, Nakahara-ku, Kawasaki, 211-8588, Japan. *Arm® Architecture Reference Manual - Armv8, for A-profile architecture*, issue g.b edition, July 2021.

[BBB+91] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks summary and preliminary results. In *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165. IEEE, 1991.

[BD01] Kristof Beyls and Erik D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, volume 14, pages 350–360. Citeseer, 2001.

[Bel66] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[BJM11] Rajeev Balasubramonian, Norman P Jouppi, and Naveen Muralimanohar. Multi-core cache hierarchies. *Synthesis Lectures on Computer Architecture*, 6(3):41–55, 2011.

[CLS06] Wen-tzer Thomas Chen, Peichun Peter Liu, and Kevin C Stelzer. Implementation of a pseudo-lru algorithm in a partitioned cache, June 27 2006. US Patent 7,069,390.

[DFF+03] Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White. *Sourcebook of parallel computing*, volume 3003. Morgan Kaufmann Publishers San Francisco^ eCA CA, 2003.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[DMS+97] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. Top500 supercomputer sites. *Supercomputer*, 13:89–111, 1997.

[DS03] Ashutosh S Dhodapkar and James E Smith. Comparing program phase detection techniques. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 217–227. IEEE, 2003.

[Fra19] Vincenzo Frascino. Arm v8. 5 memory tagging extension. In *Linux Plumbers Conference*, 2019.

[Fuj10] Fujitsu Limited, Kamikodanaka, Nakahara-ku, Kawasaki, 211-8588, Japan. *SPARC64™ VIIIfx Extensions*, version 15 edition, April 2010.

[Fuj20a] Fujitsu Limited, Kamikodanaka, Nakahara-ku, Kawasaki, 211-8588, Japan. *A64FX PMU Events Errata*, version 1.0 edition, November 2020.

[Fuj20b] Fujitsu Limited, Kamikodanaka, Nakahara-ku, Kawasaki, 211-8588, Japan. *A64FX specification Fujitsu HPC Extension*, version 1.0 edition, November 2020.

[Fuj21a] Fujitsu Limited, Kamikodanaka, Nakahara-ku, Kawasaki, 211-8588, Japan. *A64FX Microarchitecture Manual*, version 1.5 edition, June 2021.

[Fuj21b] Fujitsu Limited, Kamikodanaka, Nakahara-ku, Kawasaki, 211-8588, Japan. *FUJITSU Software Compiler Package C User's Guide*, version 1.0l20 edition, March 2021.

[Han98] Jim Handy. *The cache memory book*. Morgan Kaufmann, 1998.

[HK06] Kim Hazelwood and Artur Klauser. A dynamic binary instrumentation engine for the arm architecture. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 261–270, 2006.

[HP17] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 6th edition, Nov 2017.

[HP20] James J Hack and Michael E Papka. The us high-performance computing consortium in the fight against covid-19. *Computing in Science & Engineering*, 22(6):75–80, 2020.

[HVA+16] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 657–668, 2016.

[HW10] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Inc., USA, 1st edition, 2010.

[Int15] CAT Intel. Improving real-time performance by utilizing cache allocation technology. *Intel Corporation, April*, 2015.

[Int21] Intel. Pin 3.21 user guide, 2021. accessed on Jan/21/2020.

[JWN10] Bruce Jacob, David Wang, and Spencer Ng. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[JZTS10] Yunlian Jiang, Eddy Z Zhang, Kai Tian, and Xipeng Shen. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *International Conference on Compiler Construction*, pages 264–282. Springer, 2010.

[KHW91] Yul H Kim, Mark D Hill, and David A Wood. Implementing stack simulation for highly-associative memories. *ACM SIGMETRICS Performance Evaluation Review*, 19(1):212–213, 1991.

[KMCV10] Kamil Kędzierski, Miquel Moreto, Francisco J Cazorla, and Mateo Valero. Adapting cache partitioning algorithms to pseudo-lru replacement policies. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[KW03] Markus Kowarschik and Christian Weiß. An overview of cache optimization techniques and cache-aware numerical algorithms. *Algorithms for memory hierarchies*, pages 213–232, 2003.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200, New York, NY, USA, 2005. Association for Computing Machinery.

[LDDB+21] Núria López, Luigi Del Debbio, Marc Baaden, Matej Praprotnik, Laura Grigori, Catarina Simões, Serge Bogaerts, Florian Berberich, Thomas Lippert, Janne Ignatius, et al. Lessons learned from urgent computing in europe: Tackling the covid-19 pandemic. *Proceedings of the National Academy of Sciences*, 118(46), 2021.

[LGM+21] Júnior Löff, Dalvan Griebler, Gabriele Mencagli, Gabriell Araujo, Massimo Torquati, Marco Danelutto, and Luiz Gustavo Fernandes. The nas parallel benchmarks for evaluating c++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems*, 125:743–757, 2021.

[LGY+16] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*, pages 406–418. IEEE, 2016.

[LLD+08] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and Ponnuswamy Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378. IEEE, 2008.

[LLD+09] Qingda Lu, Jiang Lin, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P Sadayappan. Soft-olp: Improving hardware cache performance through software-controlled object-level partitioning. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 246–257. IEEE, 2009.

[MBDH99] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710. Citeseer, 1999.

[MGST70] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[Mit16] Sparsh Mittal. A survey of cache bypassing techniques. *Journal of Low Power Electronics and Applications*, 6(2):5, 2016.

[Mit17] Sparsh Mittal. A survey of techniques for cache partitioning in multicore processors. *ACM Computing Surveys (CSUR)*, 50(2):1–39, 2017.

[OFK12] Simone Ferlin Oliveira, Karl Fürlinger, and Dieter Kranzlmüller. Trends in computation, communication and storage and the consequences for data-intensive science. In *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 572–579, 2012.

[Olk81] Frank Olken. Efficient methods for calculating the success function of fixed-space replacement policies. Technical report, Lawrence Berkeley Lab., CA (USA), 1981.

[PLM09] Sascha Plazar, Paul Lokuciejewski, and Peter Marwedel. Wcet-aware software based cache partitioning for multi-task real-time systems. In *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[PS] Swann Perarnau and Mitsuhisa Sato. Discovering cache partitioning optimizations for the k computer.

[PS12] Swann Perarnau and Mitsuhisa Sato. Toward automated cache partitioning for the k computer. *IPSJ SIG-HPC*, 2012.

[SIT+20] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, et al. Co-design for a64fx manycore processor and" fugaku". In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[SKP10] Derek L Schuff, Milind Kulkarni, and Vijay S Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 53–64, 2010.

[SPP10] Derek L Schuff, Benjamin S Parsons, and Vijay S Pai. Multicore-aware reuse distance analysis. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[SXS+15] Yakun Sophia Shao, Sam Xi, Viji Srinivasan, Gu-Yeon Wei, and David Brooks. Toward cache-friendly hardware accelerators. In *HPCA Sensors and Cloud Architectures Workshop (SCAW)*, pages 1–6, 2015.

[SZD04] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. *ACM SIGPLAN Notices*, 39(11):165–176, 2004.

[WB16] Josef Weidendorfer and Jens Breitbart. Detailed characterization of hpc applications for co-scheduling. In *Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications*, page 19, 2016.

[XS20] Wenjie Xiong and Jakub Szefer. Leaking information through cache lru states. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 139–152. IEEE, 2020.

[YHKS12] Toshio Yoshida, Mikio Hondo, Ryuji Kan, and Go Sugizaki. Sparc64 viiifx: Cpu for the k computer. *Fujitsu Sci. Tech. J*, 48(3):274–279, 2012.

[YTY21] Kohji Yoshikawa, Satoshi Tanaka, and Naoki Yoshida. A 400 trillion-grid vlasov simulation on fugaku supercomputer: large-scale distribution of cosmic relic neutrinos in a six-dimensional phase space. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2021.

[ZDS09] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102, 2009.