博士論文

Doctoral Dissertation

Discovery of Memory Performance Problems Using Hardware Instruction Sampling

(命令サンプリングによるメモリ性能問題の検出手法)

2020-06-05提出

Christian Rudolf Helm

ヘルム クリスティアン ルドルフ

Student ID 48-167418

Advisor Prof. Kenjiro Taura

指導教員 田浦 健次朗 教授

東京大学

THE UNIVERSITY OF TOKYO

Abstract

In high-performance computing, many performance problems are caused by the memory system. Because such performance bugs are hard to identify, analysis tools play an important role in performance optimization. Today's processors offer feature-rich performance monitoring units (PMU). Information, which is not available through software-based techniques can be obtained, and low overhead profiling is possible. One of the features offered by the PMU is instruction sampling. It allows better attribution to code and data, and it provides more detailed information about memory accesses compared to previous hardware-based profiling methods.

The instruction sampling information is already used by some performance analysis tools. They present the data to the user for manual analysis. Some of the previous tools provide automatic discovery of performance problems. They are specialized for one specific performance problem and cannot detect other performance problems. In contrast, we combine the automatic detection of two different performance problems. We also identify the cause for those problems and add manual analysis features. We show that it is a viable, low overhead approach to collect data from the whole application first and then find different potential performance problems from the recorded data.

One of the problems that we can automatically discover is DRAM contention. We introduce a new approach based on latency measurement. This approach can benefit from the precision of instruction sampling to identify specific code locations and objects that are responsible for the DRAM contention. It can also differentiate harmless high bandwidth consumption from contention, consider the effectiveness of prefetching, and measure the severity of contention. The practical implementation of such a diagnosis system on CPUs is difficult. In modern CPUs, there is an abundance of performance counters and only superficial documentation. Different types of counters for bandwidth or latency, that seemingly measure the same thing produce different results. There is no in-depth understanding of those performance counters, and naive usage may lead to incorrect measurements. We compare various hardware latency and bandwidth measurement methods on CPUs by using micro-benchmarks. We show results of Intel Haswell, Broadwell, and Skylake systems. With our experiments, we show how and why performance counters for bandwidth and latency differ. Only the counters inside of the memory controller correctly measure bandwidth. Latency measured by instruction sampling is suitable to find DRAM contention, even though it consists of DRAM delays and in-core delays. Based on these experimental results, we establish our new detection method for bandwidth contention. We can identify three different causes for DRAM contention. First, NUMA imbalance that indicates inefficient usage of NUMA resources. Second, the memory channel imbalance indicates inefficient usage of the available memory channels. Third, a low row buffer hit rate that can also slow down DRAM accesses.

Another common performance problem is false sharing. False sharing is hard to detect manually because its occurrence depends on the data layout and cache line size. Despite numerous previous efforts, detecting false sharing is still difficult, and previous tools could not identify some cases of false sharing as we show in this work. Our approach can differentiate false and true sharing. It can identify objects and source code lines where the accesses to falsely shared objects are happening. Our approach uses information from the hardware coherency protocol to find shared cache lines. In a second step, unintentionally shared cache lines are identified by analysis of access patterns of threads. A challenge is the exact specification of conditions, that samples must meet, for false sharing to occur. The specification must be tight enough to not cause false positives, but loose enough to require only a few samples for detection.

We implemented these detection methods in an open-source tool called PerfMemPlus. The tool design is simple, provides support for many existing and upcoming processors, and the recorded data can be easily used in future research. PerfMemPlus also has manual performance data exploration features.

We show that PerfMemPlus can automatically report performance problems across a wide range of systems and benchmarks. First, we use artificial benchmarks that generate a configurable load on the memory system and benchmarks that deliberately cause false sharing and true sharing. Second, we compare known and detected performance problems in the PARSEC and Phoenix benchmarks. Additionally, we present case studies that show how PerfMemPlus can pinpoint memory performance problems in the PARSEC benchmarks and machine learning applications. The average profiling overhead of our tool is around 5%.

Acknowledgements

First, I would like to thank Professor Taura for this help and support during my time at Taura Laboratory. I also want to express my gratitude to the members of the thesis committee, namely Professor Hu, Professor Kondo, Professor Shioya, Professor Suda, and Professor Toyoda for their helpful suggestions. The advising professors Irie and Toyoda also gave me useful feedback during several discussions. I am grateful for the discussions with Professor Akiyama about DRAM performance issues and PMU measurement options.

I would like to thank everyone who was involved in the TP-PARSEC project, which has become an important part of my research. I collaborated with An Huynh, Shintaro Iwasaki, Wataru Endo, Byambajav Namsraijav and Professor Taura on this project. Also, the introduction to the N3LP benchmark by Yuchen Qiao was a great help for this research. Finally, I would like to thank all professors, staff and students from the University of Tokyo, who gave me advice to improve this research during the years.

This work was supported through a scholarship from the ministry of education, culture, sports, science, and technology of Japan (MEXT). This work was partially supported by a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO). My stay in Japan was supported by the German Academic Exchange Service (DAAD).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

If an application shows unsatisfactory performance or bad parallel scaling, the often tedious process of performance optimization starts. There can be a variety of different performance problems, and performance analysis tools are often used to diagnose them. Simple execution time profiles, like gprof [32], cannot identify if there is a performance problem and what kind of performance problem it is. Specialized parallelism profilers, like Delay Spotter [41] or Aftermath [21], can identify a lack of parallelism or inefficient scheduling. However, many potential performance problems are caused by the interaction of hardware and software. Especially the memory system, with its complex architecture and shared resources, is the bottleneck for many applications. A tool specialized for analyzing memory accesses can be helpful to diagnose such problems. Especially if it provides automatic detection of performance problems.

Modern processors have a performance monitoring unit (PMU) that can record information about the interaction of software and hardware. Hardware-based measurement is agnostic of the application's implementation and can have low overhead [3, 91, 123]. It is also possible to get information about the internal state of the processor that is not available through other methods. Among the features of the PMU, instructions sampling [22] is a promising feature. Instruction sampling enables precise identification of code locations at the granularity of individual instructions. This is not possible using performance counters. The precision of instruction sampling is particularly helpful for finding problems in unfamiliar code. In addition, the accessed data address can be recorded, and information about memory accesses, such as the latency or cache hit status is available [52, Chapter 18]. This is particularly useful for finding memory-related performance problems.

In the past, researchers have already explored how to find performance problems using the instruction sampling data [30,31,55,61,69,72,73,83,84,103,104,120,128,131]. However, there are many more aspects hidden in the instruction sampling data that are unexplored. Only for a few memory performance problems, methods to find them using instruction sampling have been developed. There are approaches for automated detection of performance problems and approaches for manual exploration of the data, often supported by visualizations. They are all separate tools that rely on their specific performance data. So far, no approach has been made to find different types of memory performance problems using only data captured with one profiling run and analyzed by one tool. Using several of the existing tools to find different kinds of memory-related performance problems makes their application impractical for realistic use cases.

While there are many different memory performance problems, the limited DRAM bandwidth of today's systems is a major contributor to unsatisfactory performance in HPC applications [64]. Especially parallel scaling to many cores is often limited by the DRAM bandwidth because the DRAM is shared by all cores. However, the simple measurement of consumed bandwidth is not suitable to identify DRAM contention. For example, if the DRAM bandwidth consumption of an application is at 95% of the maximum system bandwidth, it allows no conclusion whether this application is suffering from DRAM contention or not. Either the application is satisfied with the available bandwidth, or the application actually issues more requests than the DRAM can handle. In the latter case, the application experiences a slowdown due to the limited DRAM bandwidth. Because the measured bandwidth is limited at the system maximum, the measured DRAM bandwidth can not express the degree of DRAM bandwidth contention.

Existing tools often analyze the application as a whole and do not point out the origin of a performance problem. The precise location of the DRAM contention source is helpful to develop a performance optimization. Especially when working on complex and unfamiliar applications. If single instructions and accessed objects can be identified, the comprehension of the problem becomes easier, and finding a solution for the DRAM contention is also made simpler. For example, if a specific object is identified, its allocation strategy can be modified.

NUMA systems are widespread. By making use of multiple processors and their own DRAM, higher total system bandwidth is available compared to single socket systems. But applications must be designed to make use of the available resources. In some cases, applying interleaved allocation to specific objects can improve performance. But it is challenging to understand the allocation behavior from the source code and to predict whether interleaved allocation can bring a performance benefit. Not only the imbalance in resource usage is important. If there is a significant imbalance in the memory usage but no DRAM contention, the NUMA specific optimization usually does not yield a performance benefit. This phenomenon occurred in some of our case studies in Section 7.9. Thus it is important to consider the DRAM bandwidth contention together with NUMA imbalance.

Other than NUMA imbalance, DRAM internal performance problems can also cause bandwidth limitations. Today's systems use multiple channels to increase the bandwidth. But if only a fraction of the channels are used, or if the channel usage is skewed, the full bandwidth of the system can not be reached. Existing tools do not consider this source of performance degradation. DRAMs are equipped with row buffers. Those buffers can serve as a cache, and like for any other cache, exploiting locality is crucial for good performance of the memory system.

The measurement and especially the verification of hardware performance counters for DRAM internal problems is difficult. Verification requires a benchmark that can trigger DRAM internal performance problems on purpose. To do this, the knowledge about the DRAM addressing that happens in the memory controller is required. But for current Intel server-class processors, this information is not publicly available.

False sharing is another problem that is well known and can have huge performance implications. Its occurrence depends on the data layout specified in the code, the hardware architecture, and run-time components such as the memory allocator. Thus false sharing can be hard to find manually in the source code. False sharing is easy to fix with padding or aligned allocations. But the objects, that are affected by false sharing have to be known to apply this optimization. Instruction sampling provides suitable features required to detect false sharing.

A challenge when using hardware performance monitoring lies in the abundance of possibilities and the superficial knowledge and documentation of the performance monitoring features. With performance counters, hundreds of different events can be monitored. With instruction sampling, the number of captured samples can be in the range of millions. Each sample has more than 20 attributes. There is little documentation from the hardware vendors. What counters really measure, what the instruction sampling attributes really mean, and how they are implemented is unknown to the public. Experimental evaluation is required to find out what the reported values actually mean. Only then it is possible to make reliable measurements and to implement performance problem detection methods based on those hardware features.

Every new processor generation brings new and updated hardware performance monitoring features. For a useful performance analysis tool, it is necessary to support a wide range of hardware and to provide support for upcoming hardware with minimal changes to the tool. Some previously developed tools became unusable on current hardware and require large efforts to keep them running on current hardware [61,83,98]. The recorded instruction sampling data can be huge, and there are many aspects to explore in this data. Existing tools do not enable researchers to easily explore and reuse instruction sampling data for the development of new performance analysis methods [30,31,55,61,73,128]. A way to easily sort, filter, arrange, and display the instruction sampling data is required for researches to explore all possibilities of performance analysis with instruction sampling.

1.2 Contributions

To address the above-mentioned issues, we make the following contributions:

- A method that automatically discovers false sharing and attributes it to source code lines and objects using instruction sampling data. In addition, true sharing, inter-object false sharing, and intra-object false sharing can be differentiated.
- A method to identify DRAM contention and imbalanced NUMA resource usage. It has the following advantages over existing methods:
  - It differentiates harmless high bandwidth consumption from performance-limiting DRAM bandwidth contention.
  - It provides a metric that expresses the severity of contention.
  - It automatically detects DRAM contention to avoid the manual analysis of performance data.
  - It works on the local memory of single-socket systems as well as in NUMA systems with multiple processors and local and remote memory.
  - It can identify sub-problems such as NUMA imbalance, channel imbalance and increased row buffer conflicts.
  - It precisely attributes the DRAM contention to instructions and objects.
- A method for the measurement and visualization of DRAM memory channel imbalance and row buffer hit rate.
- A method for reverse engineering the DRAM addressing of Intel processors using performance counters.
- A future-proof lightweight profiler and analysis tool that implements the above-mentioned detection methods. It also enables users to explore all aspects of instruction sampling data and provides visualizations. The profiling works with fully optimized code and only requires debug information in the binary. No source code, hardware or OS modification is necessary.
- An evaluation that shows how our methods can detect DRAM contention and false sharing in artificial benchmarks and realistic applications. Case studies that demonstrate how our tool can use instruction sampling data to locate different kinds of performance problems.

To achieve the above-mentioned goals we rely on the following key ideas.

Tool Implementation

We implement a tool called PerfMemPlus described in Chapter 5. It is based on Linux Perf, which we extend with various different components to provide features that are not available in the standard Perf tool. We use Perf because of its wide and continuously updated hardware support and its availability on Linux systems. It already provided most of the features that we need for performance analysis. We add components for resolving dynamically allocated objects and configure it optimally for analysis of memory performance problems. To enable easy exploration of the captured data, we store instruction sampling data in a database that can be queried using SQL. Finally, we implemented our newly developed automatic detection methods and support for manual analysis of instruction sampling data in an analysis tool.

Automatic Discovery of Memory Performance Problems

Our newly developed methods for automatic detection of performance problems are described in Chapter 4. Our approach to detect main memory contention is based on the memory access latency. The latency of a DRAM access in the uncontended state is a constant hardware characteristic. Loading data from a memory can be done with a fixed latency. If bandwidth saturation occurs, the load request is delayed, and the total latency to complete the load instruction increases. This relationship is known in queuing theory. When the arrival rate (bandwidth requirement of the application) is higher than what the system can process in a certain time (maximum hardware memory bandwidth), the time required for queuing and processing (latency) of the requests will increase.

We use experiments with micro-benchmarks to find a way to measure a DRAM access latency that can be used for our automatic DRAM contention detection method. We also explore different options to directly measure the consumed bandwidth of an application. Our conclusion shows that there is only one way to correctly measure an application's bandwidth and that other methods do not include accesses caused by the hardware prefetchers. For latency measurement, we show that the instruction sampling latency is not the pure DRAM access latency, because it includes delays that come from the in-core processing of the instruction. Nevertheless, our experiments show that it is suitable for the detection of bandwidth boundness and that it has better attribution to program locations compared to bandwidth measurement and considers the effects of prefetching.

For finding false sharing, we rely on data about cache coherency provided by the hardware. Instruction sampling provides data about the coherency status of accessed cache lines. The hit modified flag (HITM) indicates that the cache line, in which the requested data resides, is shared with another core and has been previously modified. The idea to use the hit modified flag was proposed before [77]. But no concrete algorithm to identify false sharing and to differentiate it from true sharing is given in this earlier publication, and objects cannot be pointed out. We add such a differentiation and can give a clear answer whether there is false sharing or not without any further manual interpretation. We can also report the objects affected by false sharing.

Evaluation

The evaluation in Chapter 7 is based on two types of benchmarks. First, artificial micro-benchmarks, that introduce a known amount of slowdown due to false sharing or DRAM contention. Second, the PARSEC benchmarks to show that our approach also works for large applications. We rely on established benchmarks and compare our results with the existing literature.

Chapter 2

Background

This chapter introduces the basics of memory architectures, performance monitoring, and the specific performance problems false sharing and DRAM contention.

2.1 Hardware and Memory Architecture

The CPU speed has been increasing much faster than memory speed [38, Figure 5.2]. That is why every modern processor employs techniques to provide faster access to data. This section explains those techniques and the potential performance problems they might cause. There are many possible implementations of a memory architecture. This description focuses on current Intel processors because we only use those in our work.

2.1.1 Memory Hierarchy and Cache

Because the main memory is slow, faster cache memory is used [95, Chapter 5.1]. The caches provide faster access to data but are limited in size. Figure 2.1 shows a typical hardware architecture. It has three cache levels from level 1 (L1 Cache) to level 3 (L3 Cache). Overall the observed latency depends on where the requested data is actually stored. For example, in an Intel Nehalem processor, a local cache hit in L1 cache has a latency of 4 cycles, and an access to a remote DRAM has a latency of 310 cycles. Respectively the read bandwidth is limited to 9.1 GB/s on a remote DRAM compared to 45.6 GB/s on a local L1 cache [37]. Caches can accelerate data access based on two principles. First, temporal locality; Data that was accessed is likely to be accessed again soon. Second, spacial locality; After data was accessed nearby data will likely be accessed.

Cache Size

The size of an L1 cache is usually in the order of tens of Kilobytes. The L2 cache is below one Megabyte, and L3 caches are smaller than 100 Megabytes [87]. Faster memory often uses SRAM technology, which is much more expensive than DRAM. The size of the cache will be limited due to economic reasons. A lookup in a larger cache takes longer than in a small cache. This also limits the cache size. Processed data is often larger than the caches. Extra effort is necessary to organize data in such a way to make the most efficient use of the caches.

If a cache is full and additional data should be stored, the eviction of existing data is necessary. This situation causes so-called capacity misses. The data which is evicted needs to be chosen. A typical

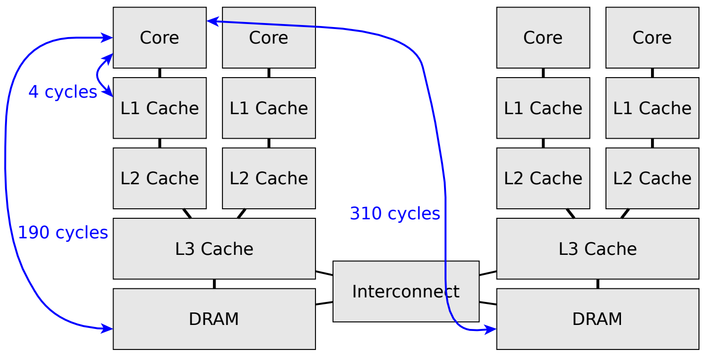


Figure 2.1. A typical multi-processor system with three cache levels and NUMA memory arrangement. Blue lines and numbers show the access latency from the core to memory. The latency was measured by Hackenberg et al. [37].

choice is to evict the least recently used data. The data which has not been touched for the longest time is evicted.

An illustrative example of the optimization for cache usage is matrix multiplication. Even if the entire matrices are too large to fit into caches, it is possible to service most of the memory accesses from the cache if the order of data access is chosen wisely. A naive implementation is shown in Figure 2.2. The values in the resulting matrix c are calculated one after another without paying respect to the order of the data required from the matrices a and b.

The cache hit ratio can be improved by calculating blocks of the resulting matrix c. This way the data needed from a and b can be reused before it is evicted. After a block of c is done, the next block of c is calculated. The block size must be set so that the one block of a, b and c fits into the cache. This strategy is implemented in the code in Figure 2.3. This code was specialized for a certain matrix size and cache size with the constants defined in lines 2 to 4. In this specific example, one a processor with an L1 cache size of 32 KB, the L1 hit rate was increased from less than 1% to over 99%.

```
mat<M,N> operator* (mat<M,K> a, mat<K,N> b) {
  mat<M,N> c;
  for (long i = 0; i < M; i++) {
    for (long j = 0; j < N; j++) {
      for (long k = 0; k < K; k++) {
        c(i,j) += a(i,k) * b(k,j);
      }
    }
  }
  return c;
}
```

Figure 2.2. A naive matrix multiplication code. Values throughout the whole range of the matrices are accessed during a loop iteration.

```
mat<1000,100> operator* (mat<1000,784> a, mat<784,100> b) {
  static const long li = 50;
  static const long lj = 50;
  static const long lk = 49;
  mat<1000,100> c;
  for(long ii = 0; ii < M; ii+= li) {
    for(long jj = 0; jj < 100; jj+= lj) {
      for(long kk = 0; kk < 784; kk+= lk) {
        for (long i = ii; i < ii+li; i++) {
          for (long j = jj; j < jj+lj; j++) {
            for (long k = kk; k < kk+lk; k++) {
              c(i,j) += a(i,k) * b(k,j);
            }
          }
        }
      }
    }
  }
  return c;
}
```

Figure 2.3. An optimized matrix multiplication code with cache blocking. The constants li, lj and lk define sections of processed data.

Cache Lines

Caches are organized in cache lines [95, Chapter 5.2]. The CPUs that we use in this thesis have a cache line size of 64 Bytes. It is the granularity in which the cache is managed. Even if small data is requested, there is always one complete cache line loaded into the cache. Eviction and coherency are also managed based on cache lines. The cache line organization can be beneficial in case of good spacial locality. Because data nearby the requested data can be loaded as part of the same cache line. But if the additionally loaded data is not used, unneeded data was loaded, using up memory bandwidth and polluting the cache. The data layout plays an important role in how well the cache lines can be utilized. An example is shown in Figure 2.4. In the first loop, the element z of Point remains unused, but it will take up space in the cache. In contrast, the optimized data layout in Figure 2.5 moves z into a new struct. In the first loop, all fields of the struct are used. Each cache line contains only useful data.

```
struct Point { double x; double y; double z; };
Point array[N];
for (long i = 0; i < N; i++) {
  D[i] = array[i].x + array[i].y;
}
...
for (long i = 0; i < N; i++) {
  H[i] += array[i].z
}
```

Figure 2.4. A data layout that leads to unused data being loaded into the cache.

```
struct PointXY { double x; double y; };
struct PointZ { double z; };
PointXY array1[N];
PointZ array2[N];
for (long i = 0; i < N; i++) {
  D[i] = array1[i].x + array1[i].y;
}
...
for (long i = 0; i < N; i++) {
  H[i] += array2[i].z
}
```

Figure 2.5. An optimized data layout. All data that is loaded into the cache will be used.

Cache Coherency

Lower cache levels, like L1 and L2, are usually private caches. Data is kept coherent in all private caches by the hardware [38, Chapter 5.8]. Software developers do not need to consider the correctness of data updates in the cache. But hardware coherency can lead to performance penalties. An update of data in one cache must be reflected on the other caches. It causes additional traffic between caches, and access to stale data will cause additional delays. For example, in the Intel Nehalem processor, a cache accesses can take between 38 cycles and 83 cycles depending on the coherency state [37]. A typical protocol for managing coherency is MESI [118], named after its four states modified, exclusive, shared, and invalid.

Set-Associative Cache

Caches are organized in sets [95, Chapter 5.3]. A set consists of blocks. Data can not be freely placed into a cache. Instead, the set where the data is placed is determined by the address of the data. The block to use within a set can be chosen freely by the cache controller. This set arrangement can lead to situations where the cache as a whole is not full, but the designated set for the data is already full. A cache miss that originates from this phenomenon is called a conflict miss. The accessed data addresses are important for the occurrence of conflict misses. Repeated access multiples of 2^x can cause increased conflict misses. The value of x depends on the cache layout.

2.1.2 Virtual Memory

Virtual memory enables the safe sharing of memory among multiple processes. It also removes limitations of small main memory from application developers and shifts responsibilities to the operating system [95, Chapter 5.4]. Every address that is accessed must be translated from the virtual to the physical address. To speed up this translation, there is a hardware cache called translation lookaside buffer (TLB). If an address is not present in the TLB, then an additional memory access needs to be done to translate the address. This leads to additional delays in the original memory access. The memory is segmented into pages. Random accesses over large address spaces and the usage of small page sizes leads to increased TLB misses. The default page size on many systems is 4 KB. Although 2 MB and 1 GB are also often available if they are activated. The TLB is split into multiple levels. An Intel Broadwell system has 64 entries in the first level for data, 128 entries in the first level for code, and 1536 entries in the unified second level [126].

2.1.3 Parallelization

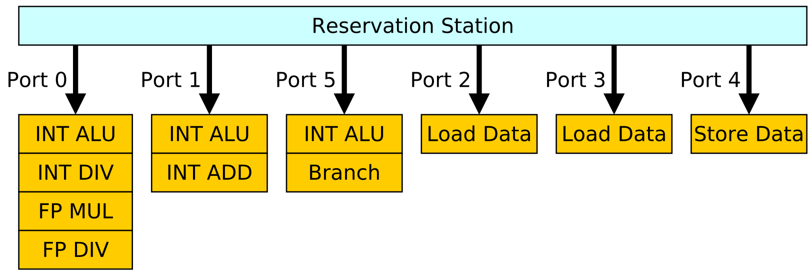


Figure 2.6. The execution ports of a processor. Instructions can be executed in parallel as long as their required slots are available and dependencies between instructions are satisfied.

Out-of-order execution allows a processor to schedule other instructions while it is waiting for data [95, Chapter 4.10]. This is a way to hide memory access latency. Instructions are scheduled to execution ports. Each of the ports has a specific function. Parallelization is limited to instructions that need to use different ports. Dependencies between instructions also limit the parallelization options. Another limitation of out-of-order execution is the limited number of instructions in the reorder-buffer. At a given time, the processor only knows about those instructions and can only schedule instructions for execution that are in this buffer. The mix of instructions and the dependencies between instructions are important to get a performance gain from out-of-order execution.

The dependencies between the instructions influence how well the latency of memory accesses can be hidden. The code in Figure 2.7 is a streaming loop. Sequential loads of a and b can be issued in parallel because there are no dependencies. Thus the latency can be hidden, and it is a bandwidth bound program. In contrast, the code in Figure 2.8 is a pointer chasing application. The next address to access is only known after the current load is complete. Because of this dependency between instructions, there is no benefit from instruction-level parallelism. This application is latency bound.

```
for(long i = 0; i < n; i++) {
  c[i] = b[i] + a[i];
}
```

Figure 2.7. A streaming loop. The sequential elements of a and b have no dependencies. Load instruction can be processed in parallel.

```
uint64_t* p = begin;
do {
  p = (uint64_t*)*p;
while (p != begin);
```

Figure 2.8. A pointer chasing loop. The next address to access is only known after the load is complete. The load instructions can not be parallelized. Adapted from [12].

A processor has multiple line fill buffers. These buffers hold memory requests that missed the L1 cache. The memory system can then process multiple outstanding misses in parallel. The processors used in the experiments for this thesis all have a line fill buffer size of 10 entries. If all the line fill buffers are occupied, no further L1 misses can be handled, and the processor will be stalled.

A processor has multiple memory controllers (also called channels) that are responsible for the communication with DRAM. The total bandwidth of a processor can only be achieved if all of those channels are used. Each channel needs at least one physical DRAM module.

Processors consist of multiple cores, but certain resources exist only once. Resources in the uncore part of a CPU are shared between all cores. Those resources include the L3 cache, memory controllers, and link interfaces to other processors. Contention can easily occur in those shared resources because multiple cores share them, and all cores together can produce more requests than the shared resource can handle. However, in processors after and including the Haswell generation, the L3 cache is sliced. This means that each core owns a fraction of the L3 cache, but all cores can access all L3 slices [87]. Because of this cache layout, bandwidth contention on the L3 cache is no longer a problem.

2.1.4 Store Instructions

Memory store instructions are usually not written directly into memory. Instead, the data is placed into store buffers [95, Chapter 5.2]. From the store buffers, it is then written to the memory. The processor core completely hands off the responsibility to store the data to the memory system. The core is not stalled because of a store. Exceptions are special memory fence instructions that guarantee that all stores are completed. A core can also be stalled if the store buffer is full. A full store buffer is the only situation where stores cause a stall in the processor. Thus store instructions are often ignored in performance analysis. However, stores contribute to the memory bandwidth and must be taken into account when looking at the memory bandwidth.

Non-Temporal stores, also called streaming stores are store operations issued by special instructions. Usually, a store allocates the stored data in a cache. In contrast, non-temporal stores write directly into the main memory and skip all caches. This is particularly useful if it is known that the stored data will not be used in the future. It saves the cost of reading data from memory into the cache before the store actually happens. This is called a write-allocate method of organizing stores. Compilers can automatically generate non-temporal stores or they can be generated by using annotations in the source code.

2.1.5 Prefetching

The prefetchers try to find patterns in the memory accesses [20]. Based on those patterns, they to guess which data will be needed soon and load it into caches. There are separate prefetchers for each cache level. The exact mechanisms of how the patterns are detected are not known. But concepts like the adjacent cache line prefetcher and detection of sequential accesses are common. Because of prefetchers, even a few memory accesses can cause a high load on the memory system because the prefetchers issue many more requests. If the memory access pattern of an application is random, the application can not benefit from the hardware prefetchers.

2.1.6 Non Uniform Memory Access

Today multi-socket systems are common. In those systems, every processor has its own DRAM. All DRAMs belong to the same shared address space, and data is kept coherent on all processors by the hardware. But an access to a remote DRAM will result in higher latency than an access to a local DRAM. Bandwidth to remote DRAMs is limited because of the used communication protocols between the processors (e.g. Intel QPI). Because there are multiple different sub-systems, each with their own processor and DRAM, the total memory bandwidth multiplies with the number of processors. There is a chance to achieve higher memory bandwidth, but the wrong allocation of data can have negative implications on performance. The allocation is managed by the operating system with page size granularity. On Linux, the default policy is first touch. A page is allocated on the memory of the processor that first touches a page. The first touch is not the request for memory allocation, but the first actual usage of the data.

2.1.7 DRAM

This section explains the terminology and principles of DRAM systems and the optimizations that modern processors use. Access to a single DRAM cell is slow. Therefore, many optimizations are in hardware to parallelize and pipeline accesses to the DRAM cells. This explanation is based on a book by Jacob et al. [54, Chapter 10].

DRAM Layout and Terminology

The DRAM system consists of hierarchical levelsModern DRAM is organized in a hierarchical arrangement of channels, ranks, bank groups, and banks as shown in Figure 2.9.

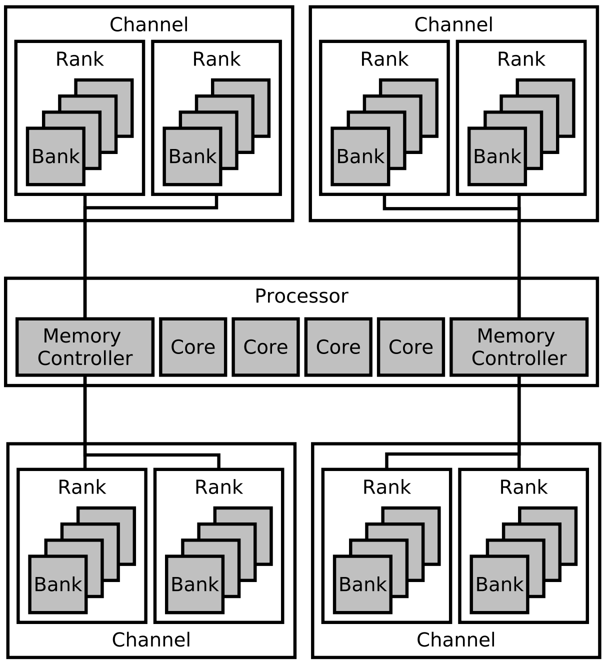


Figure 2.9. High level DRAM system organization.

Channel There can be multiple channels in a DRAM system. The channels can be accessed in parallel without restrictions. The total bandwidth is increased by distributing the requests to all available channels. Each channel needs separate hardware in the memory controller and separate DIMMs. Desktop systems usually have 1 or 2 channels and servers up to 8 channels.

Rank A channel has at least one rank but can also consist of multiple ranks. Pipelined requests at the rank level provide higher bandwidth. After sending the requested address to the first rank and waiting for data, the next rank can already receive the next address. Shared address, command, and data busses limited the available parallelism. Rank-to-rank switching penalties limit the performance.

Bank A rank consists of multiple banks. Independent accesses to different DRAM banks can occur in parallel and requests to different banks can be pipelined. Shared address, command, and data busses limit the parallelism. A bank consists of rows and a row buffer as shown in Figure 2.10.

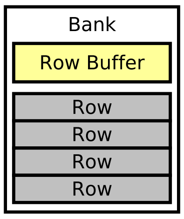


Figure 2.10. A DRAM bank consists of rows and a row buffer.

Bank Group Bank groups are an addition in the DDR4 standard [33]. More banks increase the parallelism but also increase the prefetch length. A further increase of the prefetch length over DDR3 would not match the cache line size of 64 bytes and result in a performance penalty. Thus bank groups were introduced in DDR4. Bank groups allow higher parallelism without increasing the prefetch length.

Row A row is a group of storage cells that are activated in parallel. A row is often called DRAM page.

Row Buffer A row can only be accessed when it is in the row buffer. The row buffer is essentially a cache that can hold a single element. If a row is not cached additional delays occur upon access. There are three different states upon a row access. First, if the requested row is cached it can be accessed immediately. Second, if the row buffer is empty, the requested row needs to be loaded into the buffer before the access is possible. Third, if the row buffer is occupied with a row different from the requested row, the currently cached row needs to be written back first. Then the requested row can be loaded into the buffer.

Column A row consists of multiple columns A column of data is the smallest addressable unit of memory.

Memory Controller

The memory controller takes requests from the CPU cores, accesses the DRAM, and returns the requested data.

Request Scheduling A memory controller can re-schedule incoming data requests. For example, It can prioritize accesses to rows that are already in the row buffer before accessing data that resides in a different row. In this way, the row buffer hit rage can be increased. For example, a request to an already open bank may be scheduled ahead of another request to a different row of the same open bank.

Row Buffer Management There are two strategies for managing the row buffer. They are known as the open-page and close-page management policy. Open-page management keeps the active row in the buffer to increase the chance of a hit. Close-page management closes the row after access and keeps the buffer empty to avoid conflicts. Intel processors support both mapping schemes and one must be defined at initialization by the BIOS. Depending on the management policy the bits used for address mapping may change.

Address Mapping The hardware manages the mapping of physical address to channel, DIMM, rank, bank, and column. The mapping cannot be dynamically adjusted. It is usually set up in the start-up phase of the processor. It depends on the amount of DIMMs, in which slots they are placed, the type of DIMM, the type of processor, the number of nodes, and BIOS settings. Selected bits in the physical address are mapped to address bits of the different channels, ranks, and banks. When using single bits, there is the problem of bank address aliasing. It occurs when arrays are accessed concurrently with strided accesses to the same bank. Because of this problem, it is nowadays common to use bitwise XOR of multiple bits to generate component addresses.

Write Caching Because write requests are typically non-critical in terms of latency, they can be stored and given a lower priority compared to read requests. Also, back-to-back read and write requests can only be handled with additional delays and are avoided by the memory controller.

2.2 Approaches for Performance Profiling

There are several different approaches to get information about how the software is executed on the hardware.

2.2.1 Simulation

Simulators that are used for performance analysis use a simulated version of the hardware together with the real software. It is also possible to first record a trace of the software execution and then feed the trace into a simulator. Because the hardware is simulated, it is possible to gather information from any point in the hardware. This is not possible using real hardware. Detailed information about the hardware's inner workings can be obtained. For example, the differentiation of cache capacity misses and cache conflict misses often done using simulation. Another example is the generation of communication matrices that characterize the data sharing between threads [8]. The disadvantage of simulation is the high overhead. Because of this overhead, full system simulation is nowadays seldom used for performance analysis. For example, MemSpy [78] uses simulation to diagnose cache misses and their reasons. It can also find local and remote memory accesses. It has an overhead between 22 and 58 times. The approach by Weidendorfer et al. [124] for cache analysis has an overhead of 46 times. In combination with other approaches, the simulation effort can be reduced. For example, only the simulation of caches together with hardware metrics and instrumentation is used in Intel Advisor [44]. It reduces the overhead to about 7x [5].

2.2.2 Instrumentation

Instrumentation based performance analysis inserts probes into the software. This can be done manually, by the compiler or through binary instrumentation. A common tool to do binary instrumentation is Pin [51]. For the analysis of memory accesses, it is possible to insert probes at every memory access. Those probes can then record information about the memory access. A sampled instrumentation that only records information about a few memory accesses is also possible. The collected data is used to find memory performance problems. For example, Günther et al. [36] and Zhao et al. [134] show how to detect false sharing using data gathered through instrumentation. The vampir tool-set [59] uses instrumentation to gather information about the execution flow of parallel programs.

The overhead of instrumentation is generally lower than that of simulation but is still higher than hardware-assisted approaches. The overhead varies with the implementation and sampling rate of the instrumentation. The simple call graph profiler gprof introduces about 100% overhead [59]. In older versions of Intel profiling tools, that still rely on instrumentation, an overhead of 4 times is reported [115]. A false sharing detection tool based on instrumentation [134] produces a five times slowdown of the profiled application. A tool to create communication matrices of threads produces an overhead between 39x and 148x when using relaxed tracing and up to 6157x when doing a full and accurate profiling [18]. Other approaches that achieve a similar goal come with an average overhead of 110x [80] or 225x [81].

2.2.3 Hardware Performance Monitoring

Hardware-assisted performance measurement makes use of dedicated hardware features inside of processors to gather information. This hardware is called the performance monitoring unit (PMU). This method has the advantage of low overhead [3, 91, 123], and information from the hardware can be available, which is not accessible through pure software-based approaches. For example, the hardware coherency status can be read. Gathering data through the hardware works with any kind of software. Because of these advantages, we only rely on this hardware-assisted method in our work. We describe it in detail in the next section.

2.3 Hardware Performance Monitoring Unit

Hardware performance monitoring units (PMUs) of Intel processors offer different ways to do performance measurements.

2.3.1 Performance Counters

Performance counters are a well-established way to do performance measurements. They are available in nearly all kinds of processors. Although the types of events that can be monitored differ between models.

Performance counters count the occurrence of specific events. In the PMU, there is a small number of physical counters. There are separate counters in different parts of the processor. For example, there are counters in the core itself, but there are also counters in the memory controller. They can count events related to the component where they are located. There are over 600 events on Intel Broadwell processors that can be programmed to be counted on a specific physical counter.

To use a counter, the hardware is first programmed to count a specific event on a physical counter. The counting is done in hardware. It does not produce any overhead once it is set up. The counter can then be read in software. It can be done using an interrupt, that is set up to be triggered when the counter reaches a certain value. It is also possible to read the counter at certain time intervals or certain points of the software execution. For example, such points could be the entry and exit of functions. When the counter is read, contextual information can be saved. Reading the counter incurs an overhead. Reading it more often can create a more detailed profile, but it will also increase the overhead of profiling.

If more events than physical counters available should be monitored, events can be scheduled on counters during profiling. The software then needs to change the hardware configuration during profiling. This generates additional overhead. It also introduces inaccuracies because event counter values are extrapolated for the periods when the event is not counted. It works best for applications that show a uniform behavior during their execution.

Performance counters cannot be accurately attributed to specific code locations. Attribution is only possible to the intervals where counters are read. Short intervals lead to higher overhead. Still, with performance counters, it is not possible to get down to the level of individual instructions. There is another source of inaccuracy when using performance counters. Often the counter hardware does not guarantee that counter values are increased at the exact moment when an event occurs. There is a skid between the event occurrence and when the event is counted.

Depending on the location, a counter can either be attributed to a specific core or only to the whole processor. The counters that are located in the uncore part of the processor count values for all cores of the processor. Thus the counted value can not be attributed back to an individual core. Such counters can also not be isolated to specific running applications. That is why the usage of such counters requires extended privileges. It would be possible to use such a counter to gather information about other running processes.

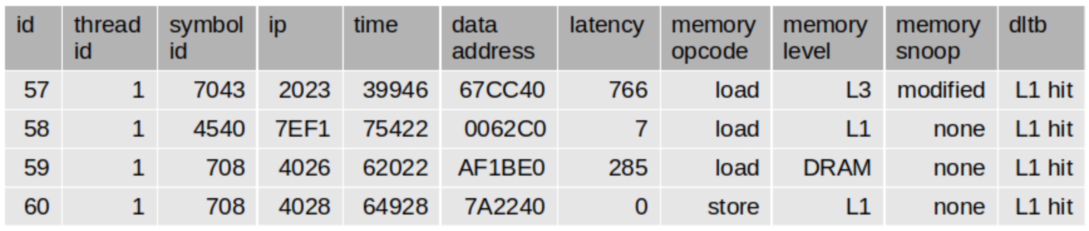
2.3.2 Instruction Sampling

Instruction sampling is a technique that is newer than performance counters. It was first introduced in the Digital Alpha 21264 processor [4,16]. It gained wider attention when AMD introduced it [22] in 2012. AMD calls this technology instruction based sampling (IBS). Intel introduced the Load Latency Performance Monitoring Facility and has continuously improved it. The current version is called precise event based sampling (PEBS) [52, Chapter 18] and is supported on all processors starting from the Nehalem generation. An exception are the Intel Xeon Phi processors that do not offer instruction sampling. In the Intel Ice Lake generation, the support of instruction sampling will be extended [25]. IBM POWER5 and the following generations also support the concept through the marked events feature [113]. While the general mechanism of instruction sampling is the same for all those implementations, our description of the features is specific for Intel processors of the Haswell, Broadwell, and Skylake generations.

Instruction sampling works by marking an instruction and observing its execution as it goes through the pipeline of the processor. Watched instructions are periodically selected. For load instructions, detailed information can be obtained. For example, the load latency, the actual place where the data was found (L1, L2, L3, remote or local DRAM), and the coherency protocol state at the time of access. The overhead of the sampling method is low because there is dedicated hardware for observing the instructions. The hardware needs to be set up once with an event that should be monitored and a memory region to write the results. The data buffer can be organized as a ring buffer, only saving the latest data. Or sampling can be stopped once the buffer is full. Data from this buffer region needs to be saved. It can be done using an interrupt that is triggered once the buffer is full. Or it can be done periodically, or also at certain points in the software execution. Saving the buffered data is what creates the overhead of instruction sampling. No context information is recorded by the software. Instead, everything is recorded by the hardware and placed in the defined memory region. A lower sampling period will lead to more samples being taken. The buffer will fill up more often. Depending on the method to read the buffer, this is how a lower sampling period can lead to higher overhead.

The advantage of instruction sampling is that it is precisely attributable to instructions and data. Each sample that is taken contains the instruction pointer of the executed instruction. With debug information in the binary, it can be resolved back to the exact source code line in the program. In addition, the accessed data address is also given in the sample. Also, information like the thread id and CPU core are recorded. Figure 2.11 is an example of what instruction sampling data looks like. It is an artificially created example. It shows the important attributes for memory performance analysis. In reality, there are many more attributes available. The rows are individual samples, and the columns show the attributes of each sample. In a realistic profile, thousands to millions of samples are taken.

Compared to performance counters, only a small number of events can be profiled using instruction sampling. Currently, those are events related to memory accesses and branch instructions. Thus, for some types of specific information from the hardware, performance counters are still required. A challenge of instruction sampling is that, as the name suggests, it is a sparse sampling approach.



id	thread id	symbol id	ip	time	data address	latency	memory opcode	memory level	memory snoop	dltb
57	1	7043	2023	39946	67CC40	766	load	L3	modified	L1 hit
58	1	4540	7EF1	75422	0062C0	7	load	L1	none	L1 hit
59	1	708	4026	62022	AF1BE0	285	load	DRAM	none	L1 hit
60	1	708	4028	64928	7A2240	0	store	L1	none	L1 hit

Figure 2.11. An example of artificial instruction sampling data. The rows are individual samples and the column are the attributes of a sample.

Approaches known from instrumentation, where all memory accesses are known, cannot be simply applied to sampling data.

Sampling Period and Sampling Rate

Because samples are taken, we need to clarify the meaning of sampling period and sampling rate. We do not consider the time between samples but the number of samples taken out of all events. Samples are taken periodically. The sampling period is the number of event occurrences between two samples. For example, if the sampling period is 100, then out of 100 occurred events, one sample is taken. The sampling rate is the inverse of the sampling period. In our example, the sampling rate would be $\frac{1}{100}$. A lower sampling period (higher sampling rate) means more samples are captured, and more detailed data is available. The sampling rate stays constant during profiling. Different types of events can be samples with different periods.

2.3.3 Tracing

Processors can record a trace of executed instructions. Intel calls this feature Processor Trace (PT) [101]. It is also supported by other processors. For example arm CoreSight [6]. This feature is often used for debugging. But it can also be applied for performance analysis. While instruction sampling and performance counters are useful to get an overall picture of an application's performance, tracing is more suitable to examine short phases of software execution in great detail. For example, the processing of individual frames in a video processing application. It is also often used to diagnose latency spikes in real-time processing applications. It is less suitable to diagnose throughput performance problems in HPC applications. Not every instruction is traced because the sequence of instructions ifs often known from the assembly code. But branch instructions are vital to reconstruct the execution flow. Thus usually branch instructions, their branch prediction, and actual branch taken are recorded. Still, there is a lot of data that needs to be captured. Usually, only a short sequence can be saved. The overhead of hardware-assisted tracing is low, in the range of 2% to 5% [109]. In this thesis, we do not consider tracing because our focus is on the analysis of memory accesses of whole applications.

2.3.4 Control Software

Typically a software layer is used between the hardware and the user who specifies the type and events of profiling. This software provides some abstractions form the hardware. It sets up the profiling, collects the data, and presents it in a readable format. Performance counters are supported by many different tools, but not all of them also support instructing sampling. For example, such tools include Linux Perf [65], PAPI [117], Intel Performance Counter Monitor [45], and Likwid [121].

Throughout this thesis, we only use Perf. It supports performance counters and instruction sampling. We use Perf because it is easy to install and run on Linux systems, no source code modifications are necessary for profiling, and the data export feature allows us to implement our own data analysis methods. Perf itself consists of a kernel component and a user-space component. The kernel component is also used by some of the other above-mentioned tools. This kernel component is accessible through the perf_events syscall. We use the Perf user-space component to control the PMU.

2.4 Specific Memory Performance Problems

In this thesis, we introduce automatic detection of two specific performance problems: false sharing and main memory bandwidth contention. In this section, we explain them in more detail. Why they occur, what the performance implications are, and how to fix them.

2.4.1 False Sharing

False sharing is a problem that originates from the cache line granularity organization of caches and the hardware coherency mechanisms. False sharing is the problem where data that is not intended to be shared is kept in the same cache line. A cache line may contain multiple different objects or multiple single elements of an array. The update of one data element in the cache of a core will invalidate the copy of the cache line in another core's cache. Upon the next access, the other core will experience a cache miss because the data in its cache is outdated. This happens even if the data that both cores access has no logical connection. Invalidation happens just because the unrelated objects share the same cache line. False sharing is always an unwanted phenomenon. It is not a correctness bug. It only affects the performance. A write access is always involved when false sharing happens. A read-read pattern does not cause false sharing.

An example of a memory access sequence that causes false sharing is shown in Table 2.1. Assume that object 0 and object 1 are placed in the same cache line. At time 1 the cache line is loaded into the cache of core 0. At time 2 the cache line is also loaded into the cache of core 1. The write access at time 3 invalidates the copy in the cache of core 1. When core 1 wants to read object 1 at time 4 the copy in the cache is no longer valid even though object 1 itself has not been modified. This is false sharing.

Table 2.1. A memory access sequence that causes false sharing.

Time	Core 0	Core 1
1	Read Object 0	
2		Read Object 1
3	Write Object 0	
4		Read Object 1

The occurrence of false sharing depends on the cache line size, compiler data layout, memory allocators, and the source code. Because of all those influences, it can be hard to find by reading the source code. There were several cases where false sharing was unnoticed until applications were analyzed using specialized false sharing detection tools. Chabbi et al. have found several previously unknown cases [13]. For example, software like Spin [112] and Libdes [85] suffer from previously unnoticed false sharing. Liu et al. [66] were the first to report some cases of false sharing in PARSEC and Phoenix benchmarks.

False sharing can be fixed by adding padding between objects so that independent objects are placed in different cache lines. Aligned allocation to cache line boundaries is also often necessary to ensure correct placement so that the data with padding utilizes one cache line completely.

True sharing

True sharing occurs when there is data in the same cache line that is intended to be shared. A modification of this data will also cause an invalidation. True sharing might be unavoidable because communication is necessary for some algorithms. However, this communication also comes with the cost of invalidating and updating the data. Finding excessive true sharing might show optimization opportunities for communication avoidance. Avoiding true sharing usually involves changes in algorithms and is usually more difficult than fixing false sharing.

Table 2.2 shows an example of memory accesses that cause true sharing. The difference in this sequence compared to the true sharing sequence is the access at time 1 and time 3. The object 1 is accessed by core 0. Thus the object 1 is shared between the two cores. Once the write is issued at time 3, the cache line becomes invalid on core 1.

Table 2.2. A memory access sequence that causes true sharing.

Time	Core 0	Core 1
1	Read Object 1	
2		Read Object 1
3	Write Object 1	
4		Read Object 1

Performance Implications

The effect of false sharing on performance can be severe. In an artificial example, up to 12x performance degradation on an 8 core system is reported [67]. Khan et al. report speedups for small but more realistic applications [57]. In an image histogram calculation, a speedup of 21% is reported. In a linear regression benchmark, the speedup reported is 10x. Even in big and complex applications, false sharing influence can be significant. Chabbi et al. [13] report speedups of up to 8.45 times in Libdes [85], which is a discrete event simulator and up to 2 times in the Spin model checker [112]. Usually, a higher core count makes false sharing more severe because more copies of data exist, are invalidated, and re-loaded or distributed. This can be seen in the data published by Chabbi et al. [13]. Thus, false sharing remains a significant performance problem as manycore processors spread.

Overall false sharing is a performance problem that can have severe implementations but is easy to fix with padding. However, to fix the problem it must be detected, and the location in the code where it occurs must be known, which is difficult to do without tool support.

2.4.2 Main Memory Bandwidth Contention

Memory bandwidth contention is the situation where the bandwidth of a memory is not high enough to fulfill all the requests of an application. In other words, the data that is needed for calculations is not delivered fast enough. This leads to delays in the processing, and the core will not be fully utilized. In today's systems, the bandwidth between the L3 cache and the DRAM is the lowest memory bandwidth in the system. Thus, we focus on the DRAM bandwidth contention.

The bandwidth required of an application running on a system depends on many factors. First of all, the intrinsic data demand of an application. It can be expressed as the arithmetic intensity or flops-to-byte ratio. Applications like sparse matrix-vector multiplication, sparse matrix transposition, and sparse triangular solver usually have an arithmetic intensity of less than 1 Flops/Byte. These applications are usually memory bound. Generally, stencil algorithms and FFT have medium arithmetic intensity.

But the details depend on the exact implementation. Algorithms like dense matrix multiplication are usually computed bound and have a high arithmetic intensity [64].

But not just the application itself has an influence, also the hardware plays an important role. Larger caches can reduce the amount of data that needs to be loaded from the main memory. Hardware prefetchers can issue additional memory loads. How effective they can work depends on the access pattern of an application. Parallel instruction execution and parallelism in the memory system increase the likeliness for bandwidth contention. Finally, the offered DRAM bandwidth is also different between systems. The memory controllers of processors only support a certain DRAM bandwidth, and also the speed of the DRAM is constrained. In NUMA systems an additional point has to be considered. Higher bandwidth is available for access to the local memory compared to the remote memory. Thus the data allocation is important for performance.

A difficulty in the detection of bandwidth contention is that high bandwidth consumption does not necessarily indicate contention. The bandwidth consumption might be just at the limit of the hardware, without the application suffering from it. Another point is the prefetchers, which may issue many memory loads in advance so that the memory bandwidth consumption is high, but the data all arrive at the core before it is needed. All these factors make bandwidth contention a problem that is difficult to detect.

Performance Implications

Main memory contention limits the performance of applications, but improving the performance can be a challenging task. Most optimization approaches focus on improving locality. This can be achieved by changing the order of memory accesses or optimizing the data layout. Often significant changes in algorithms are necessary to achieve speedups. In NUMA systems, an improved data allocation can help to speed up an application.

Performance gains reported in the literature range from as low as a few percent up to many times speedup and improved scalability. Liu et al. [69] report 8% speedup in AMG2006 [62] that was achieved with a better distribution of data across NUMA nodes. A similar optimization approach gave them a speedup of 12% in LULESH, 28% in streamcluster [11] and 53% in Needleman-Wunsch [15]. In the LULESH benchmark, Liu et al. [70] fuse two parallel loops to improve the performance by 12.9%. In the Sweep3D benchmark, a change of the loop structure leads to an improved cache hit rate and to a speedup of 2.5 times [76]. Another option is energy optimization. If a CPU core spends most of the time waiting for data from the memory. It is possible to save energy by reducing the speed of the CPU. An example is given by Molka et al. [88]. They reduced energy consumption by 4.6% without increasing the execution time of the benchmark. They decreased the CPU speed in regions of the code that are known to be memory bound.

Overall, DRAM contention is a problem that is limiting the performance of many applications. It is a problem that is caused by the hardware in combination with the software. The detection of DRAM contention is challenging because simple bandwidth measurement might be misleading. Fixing this problem can be as easy as changing an allocation setting. But often bigger changes, to improve the locality are required. The performance gains through software optimization can be significant.

2.4.3 DRAM Internal

DRAM internal performance problems can lead to increased access latency and lower achievable bandwidth. DRAM internal performance problems come from the interaction of different commands. The commands and their interaction are explained in detail by Jacob et al. [54]. Here we present a short summary of command iterations with a negative impact on performance.

Imbalanced Channel Usage Memory channels provide parallelism. If a channel is not used, or if the load on the channels is not equally distributed, the achievable bandwidth will decrease. The accessed addresses and the address to channel mapping determine which channel is accessed. Because the channels are independent, a speedup proportional to the number of channels can be achieved with optimal usage of the channels.

Imbalanced Bank Usage Banks also provide parallelism and multiple row buffers. If not all banks are used, the amount of data cached in row buffers decreases, and the options for pipelining and parallelization decrease. Like for the channel imbalance, the accessed addresses and address mapping influences the used banks.

Row Buffer Miss The row buffer can be considered a cache with a capacity of one row. If the requested row is in the buffer, the access can be served from the buffer. If the accessed row is not in the buffer, and the buffer is empty, the requested row must be loaded first. This incurs an additional delay. If the row buffer currently holds a row that is different from the requested one, even more delay will be added because the currently open row needs to be written to the DRAM array before loading the requested row. Like in a traditional cache the order of accessed addresses is important for a good cache hit rate. For row buffers, the limitation to one cached element makes this particularly important. Concurrently accessing threads share the same DRAM, so that a high number of threads increases the possibility of interference.

Chapter 3

Related Work

In the past, tools that present performance data and allow manual exploration were introduced. In addition, there are analysis tools that are designed for specific problems and come with automated detection features. We focus on tools that rely on hardware-assisted measurements, especially those that use instruction sampling. Some papers also explore the meaning of hardware performance data. In addition, we introduce work on DRAM internal performance optimization and address mapping reverse engineering.

3.1 Tools for Specific Performance Problems

Many of the existing tools are designed to help to find one specific performance problem. They are not suitable to find other kinds of memory performance problems.

3.1.1 NUMA Related Performance Problems

With the appearance of NUMA systems, data allocation became an important factor for performance. By default, the memory allocation is automatically handled by the operating system. Programmers are often not aware of the allocation decisions. Running an application developed for a uni-processor system on a NUMA system often does not bring the desired speedup. The tools introduced in this section can help with those issues.

Memphis [83] is a tool for finding and fixing NUMA related performance problems. The authors also present a paper about the application of Memphis on a Cray system [84]. This is the first tool that makes use of hardware instruction sampling on x86 processors for finding NUMA related performance problems. It uses a custom kernel module to interface the AMD instruction sampling hardware. The tool also tracks dynamically allocated objects. In the post-processing of the data, it generates a data-oriented, per-node profile. It reports the number of local DRAM and remote DRAM accesses for objects. This makes it useful for identifying remote memory accesses.

Memprof [61] is another profiler for NUMA multicore systems. It gathers data from different sources. First, the object life cycle tracking. Allocation functions are overloaded using the library preload feature and replace the existing allocation functions. Whenever memory is allocated, the call stack of the allocation is captured and written to a text file. There is one file per thread. In a post-processing step, the text files are merged into one binary file. This allocation tracker is simple and has one severe limitation. The memory is never freed to avoid the reuse of addresses. This makes the lookup of address to variable much easier, but it can cause problems for applications that make many allocations and frees. Second, the life cycle of threads is tracked through kernel hooks for the creation and destruction of threads. Third, memory access instructions are sampled to track memory accesses. They use a custom kernel module to control the hardware and record the data. It is limited to AMD processors. The overhead of profiling is around 5%.

Using this recorded data, the thread event flow and object event flow is built. The thread event flow lists the memory accesses performed by each thread. The object event flow shows which threads access an object. For each of those access entries, the latency, access type (read/write), and call chain is saved. All event flows are chronologically sorted. Using these event flows, indicators for performance problems can be found. For example, objects that are accessed from multiple threads running on different nodes.

The evaluation features that come with the tool are limited to text-based output. The percentage of remote dram accesses can be displayed. A memory profile lists the objects, functions, and object accesses, which cause the highest delays.

ScaAnalyzer [73] is the newest of a series of tools [69,71] published by Liu et al. They are successive improvements and share many common features. Those tools are implemented as an extension of HPCToolkit [1].

For gathering data, this tool relies on several libraries. As a low-level interface for controlling hardware instruction sampling, it uses the perfmon2 library. This enables wide hardware support. As a fallback (for example for ARM processors), this tool supports software IBS. It is instrumentation added by an LLVM extension during compilation. Libnuma is used to query the NUMA domain of addresses. To get the NUMA domain of a thread, a static thread to core mapping is enforced. Resolving variables from addresses is done by using the symbol table for static variables, capturing stack frames for stack-allocated data, and tracking allocations for dynamically allocated data.

The node for allocation is often decided based on the first-touch policy. Thus, the first access to a variable can provide useful insight. The identification of the first touch of a page is implemented using a custom SIGSEGV handler. First, new pages are created protected. Because of this, the SIGSEGV handler is called upon the first access to every page. Inside the handler, the call stack is recorded. Afterward, the original permissions of the page are restored and the original access can be executed. Successive accesses will not trigger the handler anymore. Using all those methods, the overhead of this tool is usually below 10 percent.

Their first tool [69] uses metrics to quantify the scalability loss to find the most promising optimization opportunities. The reported metrics help the user to identify remote memory accesses and the cache level in which a problem occurs. HPCToolkit-NUMA [71] adds a method to detect an actual place in the code where the allocation in a NUMA system is happening. Finally, ScaAnalyzer [73] introduces scalability analysis by running a workload on one and many cores. From the differences between both runs, scalability problems can be diagnosed. High latency and high scalability losses indicate a high benefit from optimization. High latency and low scalability losses indicate memory bottlenecks that are not related to scalability. Low latency always indicates that there is no benefit in optimization.

The memory architecture is separated into layers to simplify analysis. Private layer (L1 and L2), Shared Layer (Shared L3 and DRAM), and NUMA Layer (remote socket DRAM). Performance problems are attributed to one of the layers.

ScaAnalyzer and HPCToolkit-NUMA are GUI tools that allow browsing the source code, which is augmented with metrics. For variables, it shows the call paths for allocation, accesses, and the first touch of a selected variable. Showing the location of the first touch is a unique feature of this tool. It is useful for analyzing remote accesses because the first touch determines where a page will be allocated. Remote access problems are easy to discover, and performance problems can be attributed to specific memory hierarchy layers.

NUMA Visulizer is a tool by Weyers et al. [125], that provides a visualization, which is based on the physical hardware. It is an enhanced version of previous work [53]. It can show the communication between different nodes in a NUMA system. It uses Likwid [121] as the backend for reading the hardware performance counters. It uses counters from the memory controller and the QPI interface. It does not use instruction sampling.

For each socket, there is a visualization similar to a table. The cell entries show the QPI link utilization from one socket to another. The diagonal from the top left to the bottom right where the source and destination socket are the same shows the memory bandwidth utilization of that socket. The graph in each cell is a time-resolved display of the bandwidth. The maximum possible bandwidth is obtained by running a micro-benchmark before the actual analysis. This maximum value is then used to color the cells. Red cells indicate a bandwidth saturation problem. Attribution of high bandwidth to hardware sockets is possible but not to code or objects. This tool can not decide whether there is bandwidth contention or not.

DR-BW [128] is a tool that can detect remote memory bandwidth contention in NUMA Systems. It is based on machine learning using features extracted from the performance monitoring unit. The authors use DR-BW to find remote DRAM bandwidth contention on NUMA systems.

Data is recording is done with instructions sampling and the perf_events interface. This implementation supports Intel processors. The accessed data address, memory layer, latency, and originating CPU are recorded. One out of every 2000 memory accesses of every thread with latency higher than a threshold is sampled. The exact value of the threshold is not mentioned in the paper. The samples are associated with channels. A channel is a path from a CPU to a memory. Through the CPU id and the NUMA mapping, the source is known. From the accessed data address, the target memory can be resolved. An allocation tracker records dynamic memory allocations to resolve addresses to objects.

A decision-tree classifier answers the question if there is bandwidth contention on remote memories. It is trained with micro-benchmarks to extract useful features from the recorded PMU data. By varying the data size of the benchmarks, cases with and without bandwidth contention can be produced. The features include latency of memory accesses, number of remote DRAM, local DRAM and LFB accesses. Out of these features, a decision tree is created through the training with micro-benchmarks. If bandwidth contention was detected a root-cause diagnoser points out the data objects and the contribution of each data object to the contention.

NumaMMA [120] is a tool to analyze the memory accesses in a NUMA system. It collects data with instruction sampling (using the numap library [108, 119]) and provides visualizations to understand the data access and sharing patterns. It also collects data on global variables and dynamically allocated variables. The overhead is always below 12%. The results are displayed as an object profile. The access pattern can be displayed graphically. This display contains the address, time, and thread of access samples. From these visualizations, the authors conclude which allocation strategy fits best for specific objects in benchmarks.

Carrefour [27] is an operating system exetension that improves memory placement on NUMA systems. It is designed to avoid contention, not remote memory accesses. The authors report that remote accesses instead of local memory accesses decrease the performance by 20%. However, up to a factor of two of performance degradation was observed due to contention. Their OS extenstion uses the memory lantency to find problematic memory accesses. The latency is system specific. No exact method to determine it is specified. They use a metric called memory controller imbalance to decide if interleaved allocation is worthwhile. This metric is the standard deviation of the load across all memory controllers. The load is the number of requests per time unit. The metric is expressed as percent of the mean.

3.1.2 Data Structure and Array Layout

Through data layout optimizations the spatial locality can be improved. The following tools help to spot such opportunities.

ArrayTool [72] is a tool to find opportunities for array regrouping. Array regrouping is a technique to pack data from different arrays into an array of structs. Data needed in one iteration of a loop can be packed close together, sometimes even into the same cache line, to improve spatial locality.

Data is collected through instruction sampling and implemented for AMD Opteron processors. It also has an allocation tracker to identify dynamic memory allocations. The overhead in three analyzed benchmarks is between 14% and 22%.

First, the tool finds arrays with high memory access latency. Then access patterns are analyzed, and several constraints are checked. For example, only arrays with an overlapping lifetime can be grouped together. Objects that fulfill the constraints are then reported as opportunities for regrouping.

StructSlim [103] is a tool specially designed for finding opportunities to split structures. It's concept is similar to ArrayTool [72]. But instead of grouping data from different arrays into an array of structs, it is designed to find opportunities to do the opposite and split an array of structs into different arrays. If there is an array of structs, the struct size is large, and only a few elements of the struct are accessed in a loop, then this data layout can be bad for cache locality because a lot of unnecessary data (the unused fields of the struct) are loaded.

StructSlim collects data through instruction sampling and then does an offline analysis. The first step is to identify structures to split. The metric for selection is the contribution of individual objects to the total latency. In a second step, the previously selected candidates are analyzed for their memory access patterns. The access pattern indicates if there are unused elements in a struct. The overhead of this tool is between 2.05% and 18.3%.

LWPTool [131] is very similar to StructSlim [103] and Arraytool [72]. It combines the features of both tools. It uses the same methods as introduced in the previous publications to provide guidance on structure splitting and array regrouping.

3.1.3 Cache Miss Analysis

A cache miss can have different reasons. It can be a cold miss, conflict miss, or capacity miss. Depending on the type of miss, the optimization strategy is different. The following tools help to find excessive cache misses and classify their reason.

DProf [97,98] is a specialized tool to locate cache performance bottlenecks. DProf can show an object profile and classify cache misses. It can also indicate when an object is accessed from multiple cores. It only supports object-based analysis and cannot provide information about functions and source code lines.

DProf has four different views. First, the data profile. It is a list of datatypes sorted by the total number of cache misses. Second, the miss classification view. It shows which type of miss (capacity miss of conflict miss) is the most common for each data type. Third, the working set view. It shows which data types are the most active and how much are active at a given time. Last, the data flow view. It shows which functions access a given data type.

A modified Linux kernel is required to run this analysis tool. To get information about memory accesses, they use AMD IBS. Additionally, they use debug registers, which can be configured to trigger an interrupt once a specified address range is accessed. In the interrupt handler, the instruction pointer accessing the watched address is recorded. One object is tracked at a time on all CPUs. The tracked object is changed during the profile run to provide coverage of many but not all objects. Types and offsets are computed from addresses for statically allocated and dynamically allocated objects. Information from statically allocated objects is taken from the debug information, which is embedded in the binary. Allocations are tracked by modifying the Linux kernel allocator. Addresses are resolved at runtime. Allocations from outside of the Linux kernel need to be manually annotated. The overhead depends on the sampling rate and is between 2% and 14%. To detect conflict misses, a cache simulation is used, which incurs much higher overhead.

CCProf [104] is tool for lightweight detection of cache conflicts. It uses instruction sampling for data collection. The overhead in the considered benchmarks is between 1.1x and 1.51x with an outlier that has an overhead of 27x. This overhead is still much lower than that of previously used simulation-based tools. The main contribution of this paper is to use sparse sampling data for cache conflict miss detection. Before, full access traces and simulation, which incurs high overhead was used.

The main idea of the conflict miss detection is that imbalanced use of cache sets indicates conflict misses. The authors also define the metric of re-conflict distance that is the number of intermediate cache misses between two cache misses on a set. If the re-conflict distance is similar for each cache set, the cache sets are equally used. If there are sets with very low re-conflict distance, then those sets suffer from conflict misses.

3.1.4 False Sharing Detection

Some of the existing tools focus only on finding false sharing, while others can do an automatic repair of false sharing.

Detection and Automatic Mitigation

The following tools try to automatically fix false sharing if it is detected. The speedup of the automatic repair is usually lower than that of manually adding padding.

Sheriff [66] changes threads in an application to processes. It creates a simulated shared memory environment. By using per-thread page protection, memory accesses can be recorded. In the recorded memory accesses, false sharing can be detected. The overhead of this method is about 20%. The tool can also automatically mitigate false sharing by moving threads to processes.

Plastic [90] is an operating system extension for online detection and repair of false sharing. First, the HITM event is counted to find potential cache contention. Through various following steps, false sharing is confirmed. Those steps go from low overhead and coarse-grained to high overhead and fine-grained. They are implemented in modified page fault handlers and other virtual memory management features of the operating system. For fixing false sharing, it modifies the virtual to physical address mapping to move falsely shared data from the same cache line to different physical locations.

LASER [74] can find true sharing and false sharing during program execution. It has a low overhead of 2% on average. It relies on instruction sampling and the HITM flag. PMU configuration is implemented in a custom kernel module. Recorded samples are aggregated to specific program counter values. For each program counter value, the rate of HITM events per time is calculated. If the value is over a user-specified threshold, further analysis is done. A simple cache simulation model is set up, and instructions coming from the identified program counter locations are analyzed. From this access sequence, false sharing and true sharing can be differentiated.

LASER also features an online false sharing repair mechanism. It uses binary instrumentation with Pin to modify the store instruction handling of the previously identified code locations. Stores are redirected into a software store buffer, that writes to a thread-local memory, instead of the shared memory. Otherwise, it works like a hardware store buffer.

TMI stands for "Thread Memory Isolation" [17], which is a technique to mitigate the effects of false sharing. For detection, it uses HTIM performance counter with a very low sampling period. The instruction pointer of HITM accesses is known and the actual operation that is executed at this address can be looked up by dissembling the binary. Fixing false sharing is done by moving threads to their own processes. It works during program execution but has a high overhead.

Huron [57] can detect and fix false sharing. It uses two types of detection mechanisms. First, full instrumentation with LLVM and access pattern analysis with high overhead. It is required to record all load and store instructions and all memory allocations. They expect that it is applied during unit tests or similar. Thus they argue that this is an offline method. Second, they combine it with a lightweight analysis that runs during productive use of the application.

Their technique to fix false sharing is novel. Base on the data gathered through instrumentation, they are able to use useful data as padding data. Previous fixing approaches use buffer structures for writing or moving data to separate pages but do not use the analysis data for fixing. This new method results in higher speedups than any of the previous ones.

Only Detection

The following tools can detect if a program suffers from false sharing. Some of them can also point out the location and affected objects.

A machine learning approach using PMU data is introduced by Jayasena et al. [55]. Different counters from the PMU are put into a machine learning classifier. The data includes, among others, the HITM flag. This classifier has been trained with artificial benchmarks that contain, or not contain false sharing. The training process is specific to a certain machine. This tool does not point out objects affected by false sharing or source code locations that cause false sharing.

Predator [68] uses instrumentation performed by LLVM. It instruments all memory accesses to global and heap variables. This approach produces an overhead of 6 times. Based on the recorded access pattern, false sharing can be detected. It can also be differentiated from true sharing.

Cheetah [67] is working with PMU data to detect false sharing. The detection itself is based on address access patterns and similar to the concept of Predator [68]. Their method assumes that only one thread is running on each core and that there is no migration of threads to different cores. It also assumes infinite cache sizes and that data is held in a cache until it accessed by other threads. It can produce false positives if those conditions are violated. Because of hardware-assisted profiling it has an overhead of about 7% and a maximum of 30%.

Perf C2C [77] is a part of the Linux Perf tools. It uses the HITM flag in instruction sampling data to find modified cache lines. It can report readers and writers and their location in the source code. Perf C2C only relies on the HITM flag and can thus not differentiate true and false sharing. It is a command-line tool that is available in the standard Perf tools.

Intel VTune documentation [47] demonstrates which data can be used to find false sharing. Their approach is also based on the HITM flag. They show how Intel VTune needs to be configured and how the data in the tables can be interpreted. It is a manual approach and does not differentiate true and false sharing.

Feather [13] is a tool to detect false sharing on-the-fly. It uses the PMU to sample read and write accesses. Potential false sharing is identified by the number of HITM events. Once a potential false sharing candidate is found, debug registers are set up. The debug registers monitor addresses within a specific cache line. Once a write accesses to one of those monitored addresses occurs, false sharing is confirmed. Due to hardware limitations, only a part of the addresses can be monitored. The median overhead is between 3% and 8% but can be up to 2.09x. To achieve this low overhead, a custom Linux kernel is required.

3.2 Tools for Manual Exploration

Those tools usually cover a wider range of memory-related performance problems but come with no or little automated detection.

Linux Perf is a general-purpose profiler not limited to memory performance. Perf is a command-line tool and has the following features for analyzing memory accesses.

```
# Overhead        Samples  Memory access
    44.09%         265814  LFB hit
    19.05%        2596275  L1 hit
    15.90%          44470  Remote RAM (1 hop) hit
    10.14%          44119  Local  RAM hit
     8.92%          89702  L3 hit
     0.98%          36806  L2 hit
     0.56%           2646  Remote Cache (1 hop) hit
     0.34%           1311  L3 miss
     0.00%            243  Uncached hit
```

Figure 3.1. An example of the output of *perf mem rep –sort=mem –stdio*. It shows the memory access distribution grouped by memories for the whole program.

Figure 3.1 shows an output of the perf mem command. It shows where in the memory hierarchy, from L1 cache to DRAM, the requested data was found. The analysis was done for a whole program. With this method, it is not possible to attribute the performance problem to a specific code location. Thus, it can help the programmer get an idea of what is going on in the application, but it does not help pinpoint the location of the problem.

Another output of perf mem is an address centric profile. The aggregated delay of each address is displayed sorted from high to low. But only addresses of statically allocated objects can be resolved. But in most realistic applications, data is often dynamically allocated. It is not useful without knowing the dynamically addressed data and aggregating the addresses to variables.

There is no possibility of visualizing metrics using Perf. In addition to the text output shown in Figure 3.1, there is an interactive text interface (TUI). The evaluation of memory metrics is limited. Accessing the source code from this interface is supported, and there are basic filtering functions to restrict the printed data. The whole perf data must be re-read every time a new type of evaluation is requested.

MemAxes [30, 31] introduces new visualizations for data gathered through instruction sampling and annotation of the code. It uses a latency profile to point out significant functions and objects and a clustering mechanism to find interesting subsets in the data. The authors demonstrate that their visualizations are suitable for identifying unbalanced hardware utilization. To spot problems, the user needs to interpret the visualizations and draw conclusions regarding what type of performance problem is the limiting factor. It does not come with any automatic detection features. The data is collected using instruction sampling through the perf_events interface. In two case studies, the overhead is about 10%.

Intel VTune Amplifier XE is a general-purpose profiling tool, but it also has some specialized memory performance features [43]. Main memory bandwidth can be measured and attributed to the source code. But this tool cannot decide whether there is bandwidth contention or not. Which level of bandwidth usage is regarded as too high has to be set by the user. All features are accessible through a single GUI application. The source code can be viewed inside of the application. DRAM and QPI bandwidth can be visualized using a histogram. Based on this histogram, the code locations of high bandwidth usage can then be selected from a table. It offers more visualizations like a time-resolved display of the used bandwidth, and tables support many different memory-related performance metrics. It is also possible to do a data-centric analysis to show the objects responsible for high bandwidth utilization.

Data is obtained through a custom driver, which only supports Intel processors or by using a perf_events based driver. Performance counters and instruction sampling are used. Dynamic memory allocations and stack frames are tracked to resolve variables.

The Intel Performance Counter Monitor (PCM) [45] tools are a set of command-line tools for performance analysis. Internally they use perf or they can program the PMU directly as a fallback. Only performance counters are used by these tools. The following tools are available:

- pcm-core: Prints the IPC of every core in the system.
- pcm-tsx: Helps analyzing transactional memory. Shows completed and aborted transactions.
- pcm-numa: Reports the number of local and remote access for each core
- pcm-memory: Shows the read and write memory bandwidth used by each memory controller.
- pcm: Outputs IPC and cache miss rates for each CPU and shows QPI link utilization for the whole system. The output of the QPI link utilization is shown in Figure 3.2.

```
               QPI0     QPI1    |  QPI0   QPI1
----------------------------------------------------
 SKT    0       195 G    195 G   |    6%     6%
 SKT    1       246 G    247 G   |    7%     7%
----------------------------------------------------
Total  QPI outgoing data and non-data traffic :   884 G

          |   READ |  WRITE | CPU energy | DIMM energy
----------------------------------------------------
 SKT   0    341.67     150.87      9020.81      5709.44
 SKT   1    583.24     182.84      9332.79      5646.76
----------------------------------------------------
       *    924.91     333.71     18353.60     11356.20
```

Figure 3.2. An example of the output of the pcm tool. It shows the used QPI bandwidth.

The output can also be written to csv files for implementing own analysis scripts or visualizations. All the tools can only provide statistics over the whole program execution. They can not locate where the performance problems are coming from.

Aftermath [21] is a graphical tool for performance analysis of fine-grained task-parallel applications. It is not limited to memory performance but also has support for other hardware metrics like branch mispredictions. It has been developed to be used together with the OpenStream [99]. It reads a trace generated by OpenStream. Hardware metrics are recorded using PAPI [117], which relies on performance counters.

The main view is a Gantt chart timeline view of all cores in the system. It can be overlayed with hardware metrics, thus providing time-resolved analysis and attribution to tasks and cores. For example, the percentage of remote DRAM accesses can be shown. Using the execution time of tasks on certain cores can help identify NUMA problems. For example, consider a task where multiple instances executing the same code are running on all cores. If all task instances running on the same socket take more time than the task instances running on the other socket, it can indicate that the increased execution time is due to the placement of the task on the socket. A user can specify derived metrics that are calculated from existing metrics and then visualized in the tool. A connection to the source code is possible on the granularity of tasks but not on the level of individual instructions. An external editor can be started from the GUI for viewing source code. Certain tasks with an execution time in a certain range or tasks that write to a specified NUMA node can be selected.

3.3 Comprehension of Hardware Metrics

Some publications help to understand the performance data that is reported by CPUs.

The original Intel documentation [52] lists all available events. However, the descriptions are very short. To fully understand what a specific event is actually counting, it is often required to have some knowledge about the individual event counter and how it is implemented. The following studies take a more detailed look at the hardware measurement methods.

Eranian [24] introduces the possibilities of PMUs for performance analysis in 2008. Before that, PMUs were mainly used for verification purposes. He introduces ways to measure bus utilization, cache hit rates, NUMA access ratios, and latency measurement. It is based on older processors and is more of an introduction than a detailed discussion of different measurement methods.

The top-down approach by Yasin [129] is a method for structured performance analysis using PMU data. During the explanation of the approach, one can obtain some information about the inner workings of the counters and which counters can be useful for diagnosing specific problems. Such as the differentiation of frontend bound and backend bound applications. It does not go into the details of how to identify memory bandwidth limited applications. This work does not consider instruction sampling.

A study by Molka et al. [88] includes experimental evaluation of different counters. Their approach is to use micro-benchmarks to stress certain parts of the memory hierarchy and find correlations with performance counters. They report more detailed information than what is available through official documentation. The points discussed in this paper include the transfers between cache levels, and the differentiation of memory and latency bound applications. To the best of our knowledge, this is the most comprehensive prior work on this topic. It does not consider instruction sampling.

3.4 DRAM Address Mapping

A manual rowhammber based approach is shown by Seaborn [107]. First, the author uses information about the DIMM configuration from the Serial Presence Detect (SPD) ROM stored on the DIMMs themselves to build a hypothesis about the mapping. Then, the author uses a rowhammer tool that causes bit flips in the RAM. Such bit flips can be caused in neighboring rows that are in the same bank. This approach has the following disadvantages. First, the sample generation is not accurate. Bit flips are not guaranteed to occur and may also occur in rows that are not next to each other but further apart. The author describes that this occurred in the experiment and it required manual detection and removal of the outliers. Second, there is no algorithmic method for determining the addressing function. The author manually analyzes the reported addresses of successful bit flips to determine the addressing functions. While it works for this relatively old and simple PC-class processor, it is hardly possible to do a manual analysis on a more modern processor which has more complicated hashing and region based mappings.

A timing-based approach is introduced by Pessl et al. [96]. It is based on the principle that a row buffer hit results in a lower access latency than a row buffer conflict. They use pairs of addresses and repeatedly access the pairs. If both addresses in a pair are in the same bank, alternating accesses will lead to a relatively long delay due to row buffer conflicts. First, these address pairs and timing results are collected. In a second step, the linear xor functions are recovered from the data using a brute force search. They present results for several systems including Sandy Bridge, Ivy Bridge, Haswell, and Skylake, as well as Qualcomm and Samsung mobile processors. They have at most two channels and two ranks. The main disadvantage of this approach is the inaccurate attribution of physical addresses to components. It is based on measuring the timing of accesses, which can be easily disturbed. For example, the processing of other instructions in the pipeline may introduce additional delay. The memory controller is another source of inaccuracies because it can re-schedule DRAM access requests. This changes the timing and can change row buffer access behavior. Despite our best efforts, we could not reproduce the results on our machines. We suspect that such inaccuracies in the measurement lead to the inconsistent results that we have observed.

A performance counter based approach for L3 caches is presented by Maurice et al. [79]. The L3 cache is typically split into slices. The slices are addressed in a similar way as the DRAM components. Each slice has separate access counters, thus for each physical address, it can be determined which slice was accessed. Their approach uses two addresses that differ only by one bit. If the output (accessed cache slice) is the same for both addresses, the bit does not play a role in the result. If the output is different, then this bit is included in the calculation of the cache slice index.

A configuration reverse engineering approach is a method introduced by Hillenbrand [39]. The address mapping is configurable, and there are hardware registers that store the configuration. Reverse engineering of those configuration registers is done in this approach. The approach is to change the DIMM configuration of the servers, and then to monitor the changes in the configuration register space. The result is documentation of registers that goes beyond what is officially available by Intel. This study covers Intel Haswell and Broadwell systems.

3.5 Channel, Bank and Row Buffer Optimization

This section shows existing approaches to improve performance by reducing bank conflicts, channel conflicts and row buffer misses. All of the approaches require to know the hardware address mapping. They change the OS, runtime system, or memory allocator. A common pattern is to bind threads to resources such as channels or banks. The performance gains are moderate up to 14% speedup for software-based approaches and up to 21% for hardware modifications.

Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning is considering co-running applications [89]. The authors profile applications regarding their accessed DRAM rows, improve the request scheduling and map the applications to separate channels. On average their method achieves 11.1% higher throughput compared to application-unaware scheduling. The hardware is simulated because the required performance counters and possibilities to influence the channel mapping, and request scheduling do not exist in real hardware. The considered applications are from SPEC CPU2006.

Software-Hardware Cooperative DRAM Bank Partitioning for Chip Multiprocessors is an approach for lowering the interference on the DRAM between applications [86]. This work also assumes a system with multiple co-running applications. In their approach, the co-running applications are mapped to different banks. The maximum reported speedup is 14%.

Reducing NoC and memory contention for manycores by Chandru et al. [14] is a study that uses the Tilera processor. It has 4 memory channels and 8 banks on each channel. The default addressing uses the bits 13, 14 and 15 to determine the bank and bits 34 and 35 determine the channel. There is no XOR hashing. The memory controller schedules access requests so that open pages are prioritized and the requesting core is not starved. The optimized addressing restricts each core to a specific channel and bank. Cores always access the closest memory controller which reduces NoC contention. When one core uses only one bank the best performance was achieved. The performance results show about 10% higher bandwidth in Stream add.

Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems is an OS based page allocation scheme by Huang et al. [40]. This operating system changes the page allocation to assign banks to cores. It removes interference from multiple cores and improves performance by 7% on average. This research was done on simulated hardware.

A Load Balancing Technique for Memory Channels is introduced by Oh et al. [92]. They introduce the skewness metric to express the imbalance between memory channels. It is defined as the ratio of minimum value to the maximum value of the total number of requests. In their experiments, the skewness reaches up to 1.3 in one benchmark. In all other benchmarks, the channel usage is well balanced. Their optimization approach is a change in hardware that allows them to schedule requests to different channels at runtime. It brings at most 10% of speedup. This study covers HBM on GPGPUs.

Trading Cache Hit Rate for Memory Performance by Ding et al. [19] is an approach for increasing the row buffer hit rate. Hit rates are increased by changing the data layout. The maximum performance gain is about 14% in applications with irregular memory access pattern. Those are applications, where the indexes to arrays are computed by functions and no known at compile time. The optimization is implemented in the compiler and a runtime environment.

Micro-Pages [114] is an approach that aims for better utilization of the row buffer. They describe the phenomenon of reduced row buffer hit rates with a higher number of threads. They also observed that a large number of accesses, within heavily accessed OS pages, is too small, contiguous areas. The co-location of chunks from different OS pages in a row-buffer improves the overall utilization of the row buffer contents. They explore the reduction in OS page size and software or hardware assisted migration of data within DRAM to achieve the co-location. On average performance can be increased by 9% and at most by 18%. This research was done using simulated hardware.

Memory Row Reuse Distance is a metric by Kandemir et al. [56]. More specifically, they introduce to intra-core and inter-core row reuse distance. It is similar to the already known reuse distance which is often used for analyzing cache behavior. The row reuse distance is measured in simulated hardware. They also present an optimized off-chip memory request scheduling that improves the performance up to 21%. This optimization requires a hardware change in the memory controller.

3.6 Summary and Opportunities for Improvement

Some tools, like Memphis [83], Memprof [61] or DProf [97,98] implement their own low-level hardware interface. They only support outdated AMD processors, and it requires effort to make them run on current processor models.

All of the tools, that have automated detection features focus on one specific problem [61,69,71,71,72, 73, 83, 98, 103, 104, 120, 125, 128, 131]. They can not be used fo find other performance problems, that they were not designed for. For real usage, when there are multiple potential performance problems in an application, it can be a burden to use multiple different tools to identify all performance problems. Those tools also only capture data that is required for their single purpose. It makes it hard or impossible to re-use the data for finding other performance problems.

In contrast, the manual exploration tools do not have any automatic detection features [21, 30, 31, 43, 45]. They are versatile but require a deep understanding and manual effort to find performance problems.

False sharing detection has evolved and improved. Older approaches like Predator [68] have the burden of high overhead. Hardware-assisted measurements have reduced the overhead considerably [17, 17, 67, 74]. Detection results were improved. Older tools, like Sheriff [66] miss many cases of false sharing. Some tools can theoretically report false positives [67]. Others are incompatible with certain programming techniques, such as locks or atomics [17,66], as reported by Khan et al. [57]. Feather [13] needs a custom Linux kernel to do low overhead profiling, and Predator [68] and Huron [57] only work when compiled with LLVM.

The detection of accurate memory bandwidth contention is still an unsolved problem. High latency memory accesses can be pointed out, but no decision about memory bandwidth contention is done [61, 69, 71, 73, 83, 120] The DR-BW [128] approach was only demonstrated for remote DRAM contention. It is a machine learning approach and the exact mechanism of how it works and how the hardware measurement indicates bandwidth contention is not known.

Overall there is a lack of comprehension of the available hardware metrics. There are many options available for measurement, but few publications explore their meaning and applicability for performance analysis [24, 88, 129]. They do not cover instruction sampling.

With our approach, it is only necessary to record data once and then apply automatic detection methods for two separate problems on the same data. The two different problems that can be detected are false sharing and main memory bandwidth contention.

To the best of our knowledge, all of the optimizations for channel conflicts, bank conflicts, and increased row buffer hits are modifications of the OS, runtime system, compiler, or hardware [14, 19, 40, 56, 86, 89, 92, 114]. Optimization of an application itself is difficult because of virtual to physical address translation and the complex DRAM address mapping. Most of the studies are done using simulated hardware because on real systems the address mapping is unknown. Only a small performance increase of at most 18% was demonstrated for those kinds of optimizations. Studies demonstrate a common pattern, that an increased number of threads or co-running applications increase interference and thus reduce the hit rate in row buffers. In summary, the possible performance gain is low, but the required changes are high.

It is required to know the address mapping to study the channel, rank, and row buffer behavior in detail and to implement optimizations. There are existing reverse engineering approaches for the DRAM address mapping. However, they fail in practical application [39, 96]. A performance counter-based approach for L3 cache slices [79] is an inspiration, but no solution of DRAM address mapping. Overall, there is no solution available to get to know the DRAM address mapping of the Intel server systems that we use.

Chapter 4

Discovery of Memory Performance Problems

Finding performance problems in code can be difficult, even with the help of performance analysis tools. A user usually has a hypothesis of what performance problems are in a code, and tries to verify this hypothesis with data from a performance analysis tool. This can be a difficult task because the interpretation of the data or visualizations that are provided by the tool is necessary. Automated discovery that reports performance problems and their locations simplifies the process. As discussed in Chapter 3, all of the existing automatic approaches focus on one specific performance problem. It makes the usage of such tools less practical for real applications.

We have a combined detection approach that can detect false sharing and main memory bandwidth contention. For the memory bandwidth contention three possible causes can be identified. First, an imbalance in the usage of NUMA resources. Second, an imbalance in the use of memory channels. Third, a low row buffer hit rate. Figure 4.1 shows the detectable performance problems and their relationship. The NUMA imbalance, channel imbalance, and row buffer conflict detection methods can be applied in parallel. There is no dependency between those methods. We chose the order to make it easier to follow the description in this thesis.

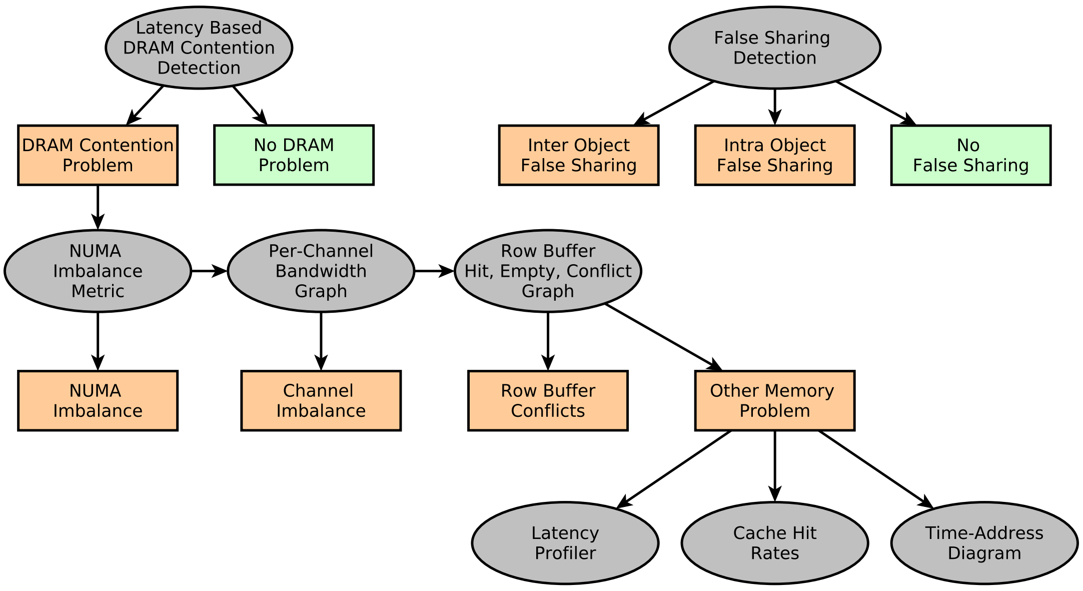
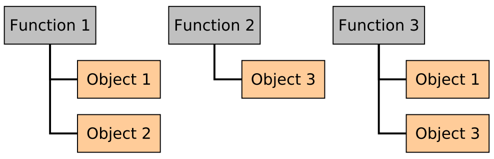


Figure 4.1. Grey ellipses show detection methods. Red boxes indicate performance problems. Bandwidth contention and false sharing are two distinct performance problems. Three causes for bandwidth contention can be differentiated. Two types of false sharing can be differentiated. Other types of memory problems can be discovered using manual analysis methods.

4.1 Candidate Selection

The automatic detection of performance problems consists of two steps. First, it selects function and object combinations as candidates. Second, these candidates are checked for signs of memory performance problems using the algorithms introduced later in this chapter. A candidate is a pair of a function accessing an object. Objects are identified by a common allocation call stack. The reason behind this grouping is that functions group the application code into different parts. Another grouping option would be to use loop nests instead of functions. But loop nests can not be directly identified in the instruction sampling data. Because memory accesses are of interest, accesses to objects are another possibility to distinguish different parts of an application. The combination of a specific function, that access an object should have a constant behavior, which can be analyzed by our detection algorithms. The creation of candidates is visualized in Figure 4.2. The left side (Figure 4.2a) shows the list of functions, each with the objects accessed by the function. The right side (Figure 4.2b) shows the list of candidates that will be checked for performance problems. All function and object pairs are considered as candidates. Except for those functions that contribute less than 1% to the total execution time. Excluded functions can at most bring a one percent speedup if their execution time was fully eliminated, which is not possible in practice. Excluding those functions can speed up the automatic detection process.



(a) Function accessing objects.

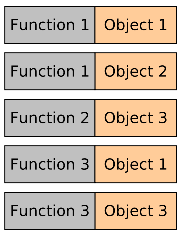


Figure 4.2. An example of candidate creation from a list of functions that access objects.

4.2 False Sharing

False sharing is hard to detect manually because its occurrence depends on the data layout, cache line size, compiler, and memory allocator. Despite numerous previous efforts [13, 55, 66, 67, 68] detecting false sharing is still difficult and previous tools could not identify some cases of false sharing as we show in Chapter 7. Some tools can theoretically report false positives [67]. Others are incompatible with certain programming techniques, such as locks or atomics [17,66]. The previous tools are specialized tools for false sharing detection and cannot find any other performance problems. Our approach can be applied to general instruction sampling data, which can also be used to detect other types of performance problems.

Instruction sampling provides data about the coherency status of accessed cache lines. The hit modified flag (HITM) indicates that the cache line, in which the requested data resides, is shared with another core and has been previously modified. The idea to use the hit modified flag was proposed before [77]. But no concrete algorithm to identify false sharing and differentiate it from true sharing is given in this earlier publication, and objects cannot be pointed out. We add such differentiation and can give a clear answer whether there is false sharing or not without any further manual interpretation. Because of the precision of instruction sampling we can point out the objects that are affected by false sharing and the source code lines where the memory accesses are happening.

4.2.1 Object Identification

Before we explain the method of false sharing identification, we need to clarify how dynamically allocated objects are identified. Objects can be identified by a common allocation call stack or as individual allocations. For example, two threads that execute the same code that contains a memory allocation would lead to two objects being created. When identifying objects by individual allocations, they are two different objects. When we identify them by the common call stack, both are combined into one object. It is important to differentiate those two identification methods. We use the terms common call path and allocation to differentiate the two in the following description of false sharing identification. In the database queries, if we use the call stack based identification, the call_path_id is specified. If we refer to the object allocation id, then the allocation_id is specified.

4.2.2 Confirmation of False Sharing

The HITM flag itself does not indicate false sharing. It only indicates that accessed data is outdated because it has been modified by another core, and there is a modified copy in another cache. For example, true sharing does also trigger the HITM flag to be set on memory access samples.

To make sure that there is indeed false sharing and to differentiate false sharing from true sharing, several conditions are checked. The challenge in specifying the conditions is that they must be as loose as possible, requiring only a minimal amount of samples. So that sparse sampling is enough to detect false sharing. But the conditions must also be tight enough to ensure that only real false sharing is detected and no false positives are raised.

The algorithm is applied to a candidate as described in Section 4.1. The input for the false sharing detection is a function and an object allocation call path. For our false sharing detection, we look at pairs of memory accesses. If any two different memory accesses samples satisfy the following conditions, then those two accesses have caused false sharing.

1. They access the same cache line and there are (other) accesses to that cache line that have the HITM flag set.
2. The accessed object has the call path id of the currently considered as a candidate.
3. The accesses come from the function that is currently considered as a candidate.
4. The thread ids are different.
5. The accessed addresses are different.
6. The timestamp difference is within 5 ms.
7. Both accesses are writes or one of them is a read and the other one a write.

The first condition filters cache lines that have accesses with the HITM flag set. This is the pre-condition of any case of false sharing. It considerably lowers the number of accesses to check because we only need to look at those cache lines where the hardware has detected modifications. The second and third conditions limit the scope to the currently considered candidate. The third condition also excludes data sharing across functions, that often happens in consumer-producer patterns. The fourth option makes sure that accesses come from different threads. One thread could case a HITM access if it moves to another core. Such cases are excluded by this condition. We do not need to pin threads to cores like other previous approaches [67] need to. The fifth condition separates true sharing from false sharing. If the accessed addresses are the same, then there is true sharing. If they are different, then there is false sharing. The sixth condition is there to exclude false sharing that is not performance relevant. The last condition makes sure that there is a read-write, write-read or write-write condition. A read-read does not cause false sharing.

Additionally, we differentiate intra-object and inter-object false sharing. This is a feature that none of the existing tools provide. Intra-object false sharing is false sharing that occurs within one object. That means the culprit is the data layout. For example, in an array of structs. The object id must be the same for both accesses. In contrast, inter-object false sharing occurs between objects allocated with individual malloc calls. Thus the memory allocator (together with the specified data layout) causes false sharing. In this case, the object id must be different.

Because we track the lifetime of objects, reuse of addresses can not incur false positives. Because the analysis is done for each object individually, objects that are allocated at different places in the source code, but happen to share the same cache lines, are not covered by our approach. They are not systematic false sharing.

4.3 DRAM Contention

The number of cores in computers is increasing, but the memory bandwidth has not scaled accordingly. Thus, the memory bandwidth is often limiting the performance of today's systems. Finding memory contention in source code is difficult. Factors such as data locality, cache sizes, prefetching, and many other software and hardware characteristics influence the required DRAM bandwidth. High bandwidth consumption that can be measured using the PMU is not necessarily an indicator of DRAM contention. Also, the attribution to source code locations and objects is hardly possible with those bandwidth counters.

We introduce an approach to use instruction sampling, identify performance degrading bandwidth usage, and attribute it to source code lines and objects. In cases of high bandwidth utilization, our method can distinguish if an application is suffering from bandwidth contention or if it is within the limits of the system.

4.3.1 Latency as Indicator For Bandwidth

Instruction sampling data can be precisely attributed to code and data, but memory bandwidth can not be measured with instruction sampling. Our idea is to use the latency of a DRAM load as an indicator of bandwidth contention. Loading data from a memory can be done with a fixed latency. If other issues, like bandwidth saturation, occur, the load request is delayed, and the total time to complete the load instruction increases. Figure 4.3 shows the increase in latency when increasing the main memory bandwidth. The data was generated with the Intel memory latency checker [122] on the systems which are introduced in Section 4.3.2. It shows that the latency stays low with only a small increase until the bandwidth gets close to the hardware limit. At this point, when the system reaches its throughput limit, there is a sharp increase in latency. This relationship is well known in queuing theory. When the arrival rate (bandwidth requirement of the application) is higher than what the system can process in a certain time (maximum hardware memory bandwidth), the time required for queuing and processing (latency) of the requests will increase. There are buffers in the hardware, that can store the in-progress requests. This experiment was done using a software-based measurement of latency and bandwidth. The transferred amount of data and number of memory accesses is calculated. Only the execution time is measured. Based on those values, the bandwidth and the latency of a single access can be calculated.

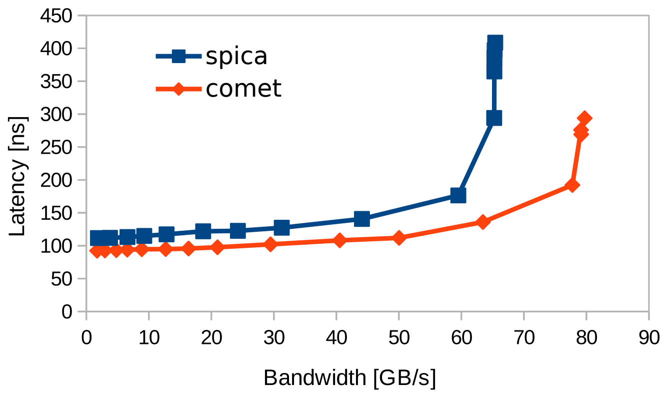


Figure 4.3. The latency change of DRAM accesses with an increased memory bandwidth. The latency and bandwidth are measured in software. Spica and Comet are two different hardware platforms described in Section 4.3.2.

Measurement on Intel CPUs

After having introduced the concept of latency based bandwidth contention detection, we need a way to measure memory access latency on processors. Because we also want to compare our latency based detection method with direct bandwidth measurement, a way to measure the memory bandwidth is also required.

Current Intel processors have a powerful performance monitoring unit (PMU) [52, Chapter 18]. It has support for instruction sampling and performance counters. With performance counters, hundreds of different events can be monitored. Among them are different ways to measure the main memory bandwidth and different ways to measure the memory access latency. Naturally, the question arises how those methods differ from each other, and if one of them is superior compared to the others. This question is not easy to answer because there is little documentation. What those counters really measure and how they are implemented is unknown to the public. Experimental evaluation is required to check if a certain counter reports the intended event accurately.

The latency based concept described above assumes that the latency is the pure DRAM access latency. That means the measured latency is based only on accesses to the DRAM, not access to any other cache levels. And that the latency is measured from the beginning of the request to the memory until the completion of the request. Other parts of the instruction execution, for example, waiting time until an execution port becomes available, would also influence the results. Thus, to apply the latency based detection method, an exact understanding of what is measured by the hardware latency counters is required.

We use experiments with micro-benchmarks to find out what is actually measured by different PMU based methods. We explain and examine the various trade-offs that come with different PMU measurement methods. Our conclusion shows that there is only one way to correctly measure an application's bandwidth and that other methods do not include accesses caused by the hardware prefetchers. For latency measurement, we show that the instruction sampling latency is not the pure DRAM access latency, because it includes delays that come from the in-core processing of the instruction. Nevertheless, our experiments show that it is suitable for the detection of bandwidth boundness and that it has better attribution to program locations compared to bandwidth measurement and can consider the effects of prefetching. We show concrete results for Intel Skylake, Intel Broadwell and Intel Haswell architectures but our method to verify the performance counters can be applied to any type of processor.

4.3.2 Experiment Setup

All of the experiments were executed on machines running Ubuntu 18.04, and the micro-benchmarks were compiled with gcc 7.4. All measurements were done with Linux Perf version 4.17.8. Perf supports multiplexing of events if there are not enough physical counters. We do not use this feature. In our experiments, events are always pinned to a physical counter for the highest accuracy. For experiments that involve instruction sampling, PerfMemPlus was used. All reported numbers are averages from 6 repeated executions. The migration of OpenMP threads has been disabled for all experiments.

Hardware

We conducted the experiments on the last three Intel architectures and machines with different DRAM speed. We used one Haswell, two Broadwell, and one Skylake machine. The two Broadwell systems have a big difference in DRAM speed. The details of the machines are listed in Table 4.1. We used numactl to limit the execution to one of the available nodes and its local memory.

Table 4.1. The hardware used for the evaluation of bandwidth and latency measurement methods.

Name	**Architecture**	**CPU**	**DRAM Speed**
Arcturus	Broadwell	E5-2699v4@2.2Ghz	2400Mhz
Comet	Haswell	E5-2699v3@2.3Ghz	1847Mhz
Rigel	Skylake	Xeon 8176@2.1Ghz	2666Mhz
Spica	Broadwell	E7-8890v4@2.2Ghz	1600Mhz

We want to run our experiments with and without using the hardware prefetchers, to study the influence of the hardware prefetchers. The prefetcher of Intel processors can be turned off and on in software. It can be done on a running system by writing values into machine specific registers [42]. We use the msr-tools to write to the relevant registers using the following commands:

- Disable prefetching: sudo wrmsr –all 0x1a4 15
- Enable prefetching: sudo wrmsr –all 0x1a4 0
- Read the current status: sudo rdmsr 0x1a4

Memory Read Benchmark

We use the memory read benchmark to issue read requests with varying bandwidth and then measure the resulting latency. The memory read benchmark sequentially reads a large array from memory. The hardware prefetchers can predict the access pattern and load data. After issuing four memory read operations, a configurable number of NOPs is inserted. Those NOPs are used for regulating the bandwidth and load on the memory system. The array called *A* consists of records, each with a size of 64 Bytes. Only the first element in a record is accessed. This means that each cache line is only accessed once. The array *A* is defined as volatile to force a read memory access. The code is shown in Figure 4.4. When the hardware prefetchers are turned off, this benchmark allows fine regulation of the memory bandwidth, as shown on the left side of Figure 4.5. Because there is a steep change in bandwidth between 50 and 55 delay cycles, additional data points were added there. When the hardware prefetchers are turned on, no such fine control is possible. Changing the number of threads still allows for a course regulation of memory bandwidth, as shown on the right side of Figure 4.5. We only show the graphs for one system because other systems show similar results. The coefficient of variation of the bandwidth measurement is 0.30% on average and at most 3.97%.

```
#pragma omp parallel
for (long s = 0; s < 200; s++) {
  for (long t = 0; t < 10000000; t++) {
    for (int c = 0; c < 4; c++) {
      A[c][t].next;
    }
    for(unsigned long i = 0; i < delayCount; i++) {
    asm volatile("nop");
    }
  }
}
```

Figure 4.4. The code of the main loop of the memory read benchmark.

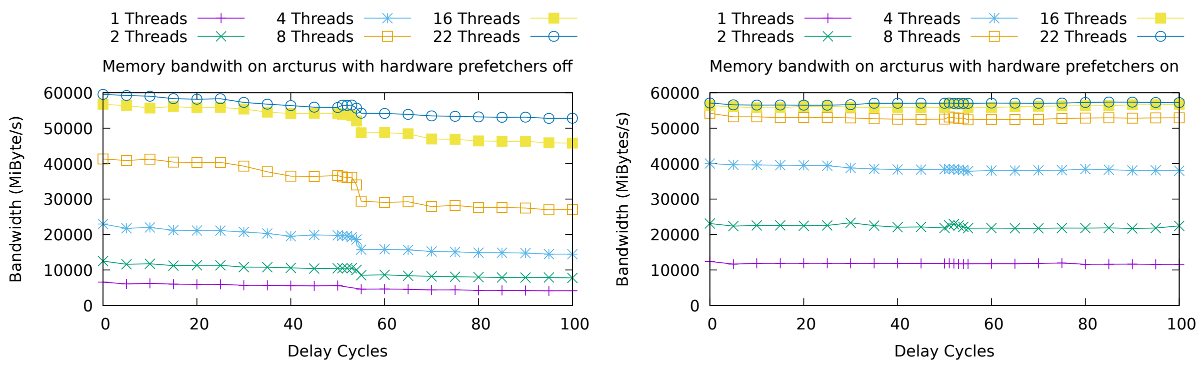


Figure 4.5. The DRAM bandwidth of the memory read benchmark. Course-grain control of the bandwidth is possible by changing the number of threads. Fine-grain control of the bandwidth is possible by adjusting the number of delay cycles.

Stream Triad

We use Stream triad [82] because we can calculate the amount of transferred data. Then we compare it with the measured amount of data transfer to verify the accuracy of bandwidth counters. All arrays of the code in Figure 4.6 have the same size. The array a and b are read. The array c is written. The current Intel architectures use the write allocate scheme, which means that the array c is also read into the caches before it can be written to main memory. We do not use non-temporal stores. Thus, the total read amount of data is three times the array size. It needs to be multiplied by 11, because the benchmark is executed 10 times and for initialization, all arrays are written once. Additionally, the size of the binary is added. We use an array size of 610.4 MB, which results in a total transferred size of 21.974 GB.

```
#pragma omp parallel
  for(long i = 0; i < n; i++) {
    c[i] = b[i] + s * a[i]
  }
```

Figure 4.6. The code of the main loop of Stream Triad.

4.3.3 Direct Bandwidth Measurement

Direct measurement of the memory bandwidth using the PMU is possible. All the methods that we introduce in this section are based on performance counters. Thus attribution to source code and objects is not as good as with the instruction sampling based approaches. Based on the documentation we found the following performance counters that involve DRAM accesses or DRAM bandwidth:

1. mem_load_uops_l3_miss_retired.local_dram
   mem_load_l3_miss_retired.local_dram
2. unc_h_requests.reads
3. uncore_imc_*/cas_count_read/
4. offcore_response.*

The first counter in the list counts the number of uops that load data from the local DRAM. It is not suitable for measuring the DRAM bandwidth. First, a load uop can have different data widths. There is no information about the data size of the load given. Second, loads that are classified as LFB hit, but the data actually comes from the DRAM are not included in this counter.

Counter number 2 and 3 are very similar. Both are in the uncore part of the CPU. They produce similar results and share the same advantages and disadvantages. The difference between them is that the unc_h_requests.reads counter is inside of the home agent and the uncore_imc_*/cas_count_read/ counter is in the memory controller. The offcore_response counters also look like a viable candidate. We focus our detailed evaluation on the memory controller counters and offcore response counters.

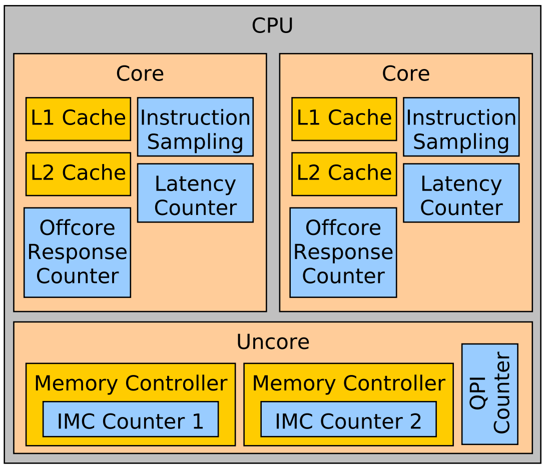


Figure 4.7. The location of different bandwidth and latency counters within the CPU. Uncore counters have more limitations compared to core-local counters.

Memory Controller Counters

The counter inside of the integrated memory controller (IMC) is called uncore_imc_*/cas_count_read. It counts the number of cache lines transferred by the memory controller from the DRAM to the processor. There are similar counters for written cachelines called uncore_imc_*/cas_count_write.

There are multiple memory controllers on one chip. In the event name, instead of the asterisk, the number of the memory controller can be used. When using Perf, an asterisk instead of the number will activate counting on all memory controllers, and the values of all memory controllers are added. To get the number of bytes transferred this value has to be multiplied by 64. Recent versions of Perf already do this for the user and print the data in MiByte.

Because the counters are located in the uncore part of the CPU (Refer to Figure 4.7), they can only count in global mode. It is activated in Perf with the –all-cpus flag. That means that they count everything, including memory traffic caused by other applications and the operating system. This introduces additional sources of noise to the measurement. Because those counters measure the whole system, which could allow gathering information about other running applications, extended privileges are required. Either the perf_event_paranoid flag must be set to -1 or root access is required. This may hinder usage on shared systems. Because all cores of a processor share the same memory controllers, they also count for one whole socket. This makes attribution to code and data even more difficult because the traffic cannot be attributed to a specific core.

The –per-socket flag can be used to gather statistics for every socket individually. For example, the memory bandwidth usage of each socket can be used to diagnose unbalanced usage of the memory in NUMA systems.

Offcore Response Counters

Another option to count the amount of transferred data are the offcore response counters. Those counters are located at the edge between the core and the uncore part of the processor, as shown in Figure 4.7. This means that they can be attributed to specific cores, but still have the attribution problem that is common for all performance counters as explained in Section 2.3. Because the offcore response counters are core local and can be restricted to the profiled application, it is possible to use them even on systems with restricted access. Only the bandwidth of the profiled application will be counted. It leads to lower noise in the measurements.

There are different sub-events for the offcore_response events. It is possible to select either demand accesses, prefetched accesses, or all accesses. Memory reads, writes, code reads, or all accesses can also be configured. The response type can be configured. It can be local DRAM, remote DRAM, or various cache coherence dependent options as well as an option to include all responses. Overall, those offcore events allow a very fine selection of specific events.

The event we used for the experiments on Haswell and Broadwell is offcore_response.all_reads.llc_ miss.local_dram. According to the documentation, it counts all read accesses (including code reads and reads required for later writes) to the local dram no matter if they are demand or prefetch.

On Skylake the event we used is offcore_response.all_data_rd.l3_miss.any_snoop. The more specific event, which according to the documentation "Counts all demand and prefetch data reads that miss the L3 and the data is returned from local dram" called offcore_response.all_data_rd.l3_miss_ local_dram.snoop_miss_or_no_fwd never counts any events in our experiments.

Bandwidth Experiment Results

The experimental results show that the offcore_response counter does not include accesses due to prefetching, even though the documentation says that demand and prefetched accesses are included. On the Skylake system, even disabled hardware prefetchers do not result in correct measurements.

To verify whether the bandwidth measurement is accurate, we use the Stream Triad benchmark as introduced in Section 4.3.2. Figure 4.8 shows the results. First, we can see that the results are similar on Haswell and Broadwell systems.

Memory Controller Counters The IMC counters accurately measure the amount of transferred data. The calculated data volume is a little lower because only the part of the main loop in Stream Triad and the initialization add to the total amount of transferred data. The IMC counter counts every memory transfer that happens while the benchmark is running. The IMC counter has higher noise because it not only counts the applications data transfers but all data transfers that occur while the program is running. This includes data transfers due to the operating system and other applications running at the same time.

Offcore Response Counters The offcore counter is limited to the program under test but still counts other memory accesses that occur in the program and thus show a transferred data volume that is slightly higher than the calculated one. The documentation says that prefetched accesses are included [52, Chapter 18, Page 41] when using this counter. However, our experiments show that the data transferred because of prefetching is not included. When prefetching is turned off, the data is the same as with the other counters and the calculated data amount. In contrast, when prefetching is turned on, the offcore response counters do not report the correct data volume.

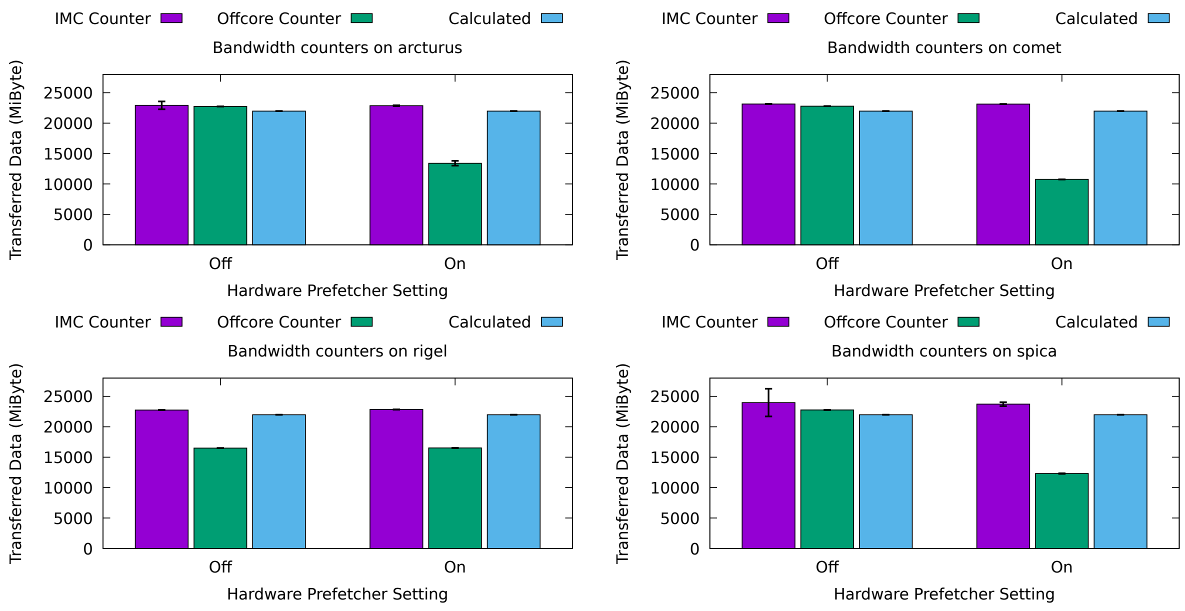


Figure 4.8. The transferred data volume of the Stream Triad benchmark measured with offcore repsonse counters and IMC counters compared to the calculated transferred data.

Exceptions on Skylake On the Skylake machine (Bottom left of Figure 4.8), even if prefetching is turned off, the offcore counter does not report the correct transferred data. But an effect of the prefetcher setting can be seen when looking at other variants of the offcore counter. There are counters for prefetched data (offcore_response.pf_data_rd.l3_miss.any_snoop) and demand data (offcore_response.demand_data_rd.l3_miss.any_snoop). When hardware prefetchers are turned off, the offcore response counter for the total amount reports the same value as the demand counter. The counter value for prefetched data shows values close to zero. In contrast, when hardware prefetchers are turned on. About two-thirds of the data volume is counted by the prefetcher counter and about one third by the demand counter. The sum of those two counters for prefetched data and demand data is always the sum of the counter that includes all accesses.

4.3.4 Latency Measurement

There are two options available for latency measurement. The first one is a metric based on performance counters, and the second one is the latency reported by instruction sampling.

Instruction Sampling

The memory access latency is one of the attributes of a sample. This latency is the time from the start of the execution of an instruction until it reaches the globally observable state [52, Chapter 18 p. 22]. Because the memory level, from which the data was loaded is an attribute of every sample, latency can be calculated for each memory level individually. In these experiments, only those samples which hit in the DRAM and that hit in the TLB are included. Latency of other cache level hits is not included. We calculate the average latency from the filtered samples. The instruction sampling latency has the most precise attribution to code and data. Only data from the profiled application is collected, and no special privileges are required to use this profiling method. We use PerfMemPlus to do the profiling and view the data because it is designed for instruction sampling and provides easy access to the captured data.

Performance Counters

Another way to measure the latency is to use a metric that is based on performance counters. All required counters are part of the core and L1 cache. Same as the offcore response counters, they can be attributed to specific cores and data recording can be limited to the profiled application.

The counter cpu/l1d_pend_miss.pending counts the time spent for loading data into the L1 cache. It sums up the time for parallel accesses. In other words, this counter is increased by the number of currently outstanding L1 data cache misses in every cycle.

In addition the counters cpu/mem_load_uops_retired.l1_miss and cpu/mem_load_uops_retired.hit _lfb are required. The first one counts the number of load instructions that miss the L1 cache. The second one counts the number of load instructions that hit the line fill buffer (LFB).

Based on those counters we define two metrics. The L1 miss latency expresses the average time it takes to fulfill a load request that missed the L1 cache. It does not include LFB hits. The load miss real latency is the average time it takes to fulfill a load request that missed the L1 cache or hit in LFB. The load miss real latency is a metric that is already available in recent Perf versions.

For abbreviation of the long event names the following Greek letters will be used in the metric definition:

$$\begin{aligned}
\alpha &= \text{cpu/l1d\_pend\_miss.pending} \\
\beta_{Haswell,Broadwell} &= \text{cpu/mem\_load\_uops\_retired.l1\_miss} \\
\beta_{Skylake} &= \text{cpu/mem\_load\_retired.l1\_miss} \\
\gamma_{Haswell,Broadwell} &= \text{cpu/mem\_load\_uops\_retired.hit\_lfb} \\
\gamma_{Skylake} &= \text{cpu/mem\_load\_retired.fb\_hit}
\end{aligned}$$

Based on those counters two latency metrics are defined as follows:

$$\text{L1 miss latency} = \frac{\alpha}{\beta}$$

$$\text{load miss real latency} = \frac{\alpha}{\beta + \gamma}$$

The latency is an average for the whole memory hierarchy. It includes the latency of accesses to L2 and L3 caches. In general, it can not be used to measure only the DRAM access latency. Because in our experiments the micro-benchmark only accesses DRAM and no other cache levels, we can still use it to measure the DRAM access latency.

Latency Experiment Results

The main insights from the experiments are: First, memory access latency is indeed a good indicator of bandwidth contention. Second, the effectiveness of prefetching can be seen in the latency. Third, the counter-based metrics only work when including LFB hits. Fourth, the instruction sampling latency consists of in-core instruction processing delays and memory access latency. Despite that, the instruction sampling latency is suitable for diagnosing bandwidth contention.

Hardware Prefetchers Off Figure 4.9 shows the measured values of the three different latency measurement methods with rising bandwidth. Common for all four systems is the rise in latency with higher bandwidth. The absolute values of the load miss real latency and L1 miss latency are higher on the Haswell system (Figure 4.9 top right graph), compared to the other systems.

It is also visible that the instruction sampling latency reaches high values even for low bandwidth situations. We will discuss the reason for that in Section 4.3.4. In contrast, the L1 miss latency and load miss real latency only increase when there is an actual bandwidth limitation. The load miss real latency and L1 miss latency have almost the same values. Because there is no prefetching and only one element per cache line is accessed, there are no LFB hits in these experiments. Looking at the metric definitions in 4.3.4, we can see that in this case, both metrics produce the same value.

In Figure 4.9 three different latency metrics are shown. The coefficient of variation of the measurements of the instruction sampling latency is on average 6.48%, and at most 6.69%. For the L1 miss latency, it is on average 4.15%, and at most 7.81%. For the load miss real latency, it is on average 4.40%, and at most 7.59%

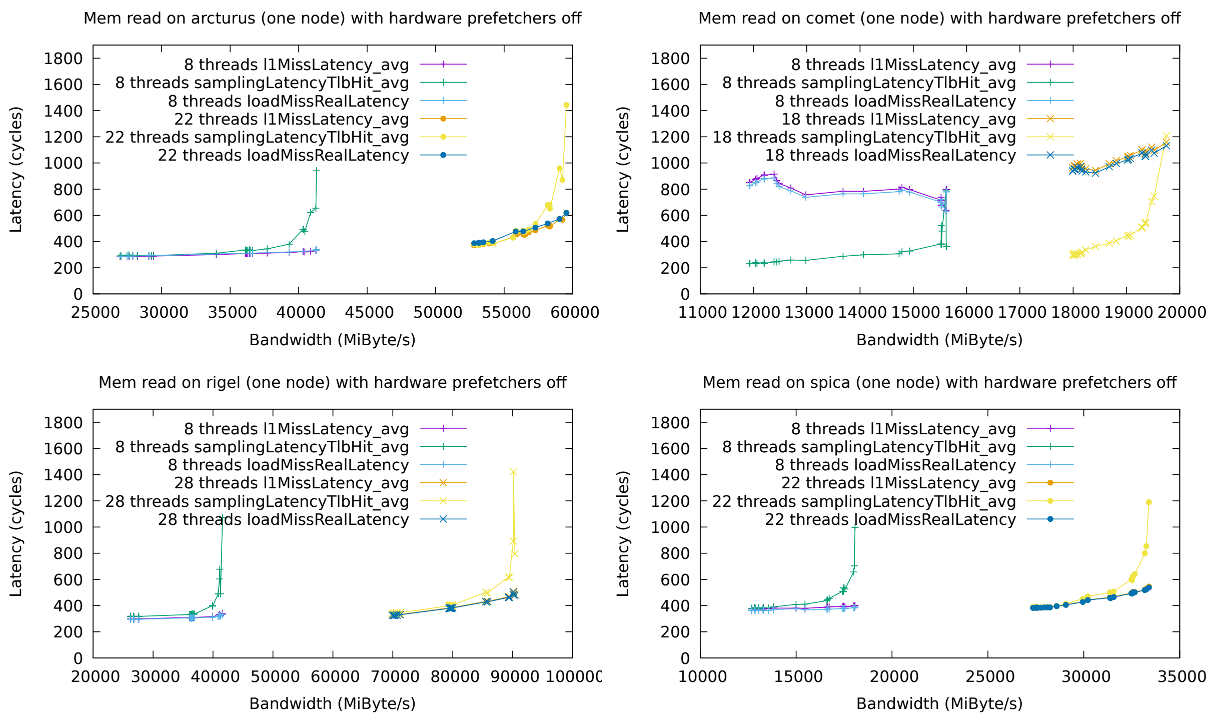


Figure 4.9. The results of different latency measurement methods with increasing DRAM bandwidth when hardware prefetchers are disabled. A selection of threads was picked for this diagram to indicate low and high bandwidth situations. The middle bandwidth section was left out to not obstruct the view. Instruction sampling latency can rise even if the DRAM bandwidth limit is not close.

Hardware Prefetchers On When the hardware prefetchers are turned on, the number of delay cycles plays an important role. Thus, we show separate diagrams for each type of latency. For each type of latency, it is visible that the latency rises with higher bandwidth. Also, all types of latency show a dependency of latency value and the number of delay cycles. This is a phenomenon that does not appear when the hardware prefetchers are turned off. For the sampling latency in Figure 4.10, we can see that latency increase at low bandwidth situations does not appear.

The sampling latency (Figure 4.10) and the load miss real latency (Figure 4.11) show the same trend regarding delay cycles and latency. Even if the bandwidth is almost the same, when the number of delay cycles is low, the latency is higher. Because of the prefetcher, the main memory bandwidth is the same no matter how many delay cycles are inserted. In cases with a low number of delay cycles, the requests for data are issued quickly. The prefetcher does not have enough time to load the required data. Thus, the requests have to wait a long time until the data finally arrives. When there are many delay cycles, the prefetcher has enough time to load the required data. When the load request is finally issued, it will have to wait only a short time.

This is an advantage of the latency measurement compared to the bandwidth measurement. In both situations, many or few delay cycles, the bandwidth is the same. However, the application with few delay cycles has more severe performance problems because there is actual waiting time for the data. In contrast, in the situation with many delay cycles, the prefetcher is simply doing its job, and there is no severe performance problem.

When the prefetcher is turned on, there will be many hits in the LFB and only very few real L1 cache misses. This ratio will change depending on the number of delay cycles. Many delay cycles will result in many LFB hits and few L1 hits. This effect is visible in Figure 4.12. A high number of delay cycles results in a high L1 miss latency because the denominator in the metric definition will decrease. We conclude that the load miss real latency is better than the L1 miss latency to measure memory access latency.

The coefficient of variation of the measurements of the instruction sampling latency shown in Figure 4.10 is on average 53.51% and at most 62.34%. The measurement of the instruction sampling latency is less stable when hardware prefetchers are turned on, compared to the measurements with disabled hardware prefetchers. The coefficient of variation of the measurements of the load miss real latency shown in Figure 4.11 is on average 12.67% an at most 42.56% The maximum coefficient of variation of the measurements of the L1 miss latency shown in Figure 4.12 is on average 13.49% and at most 41.33%.

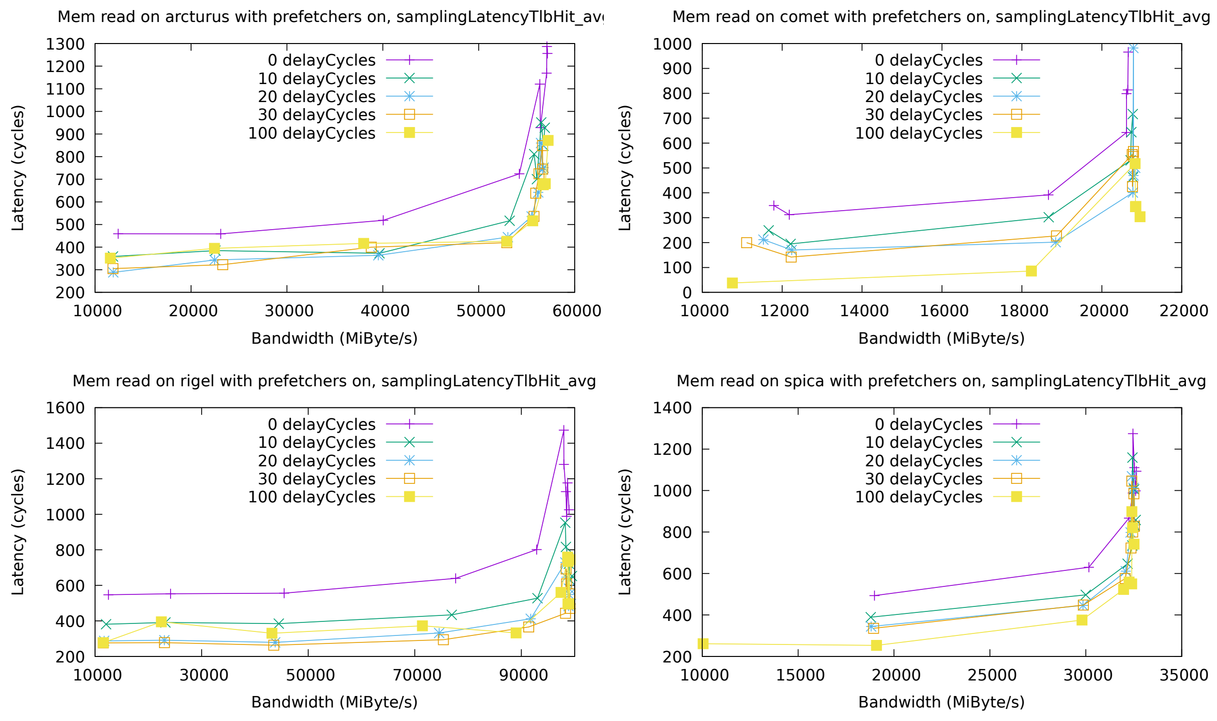


Figure 4.10. When hardware prefetchers are enabled, higher DRAM bandwidth and fewer delay cycles lead to higher instruction sampling latency. The number of threads controls the bandwidth.

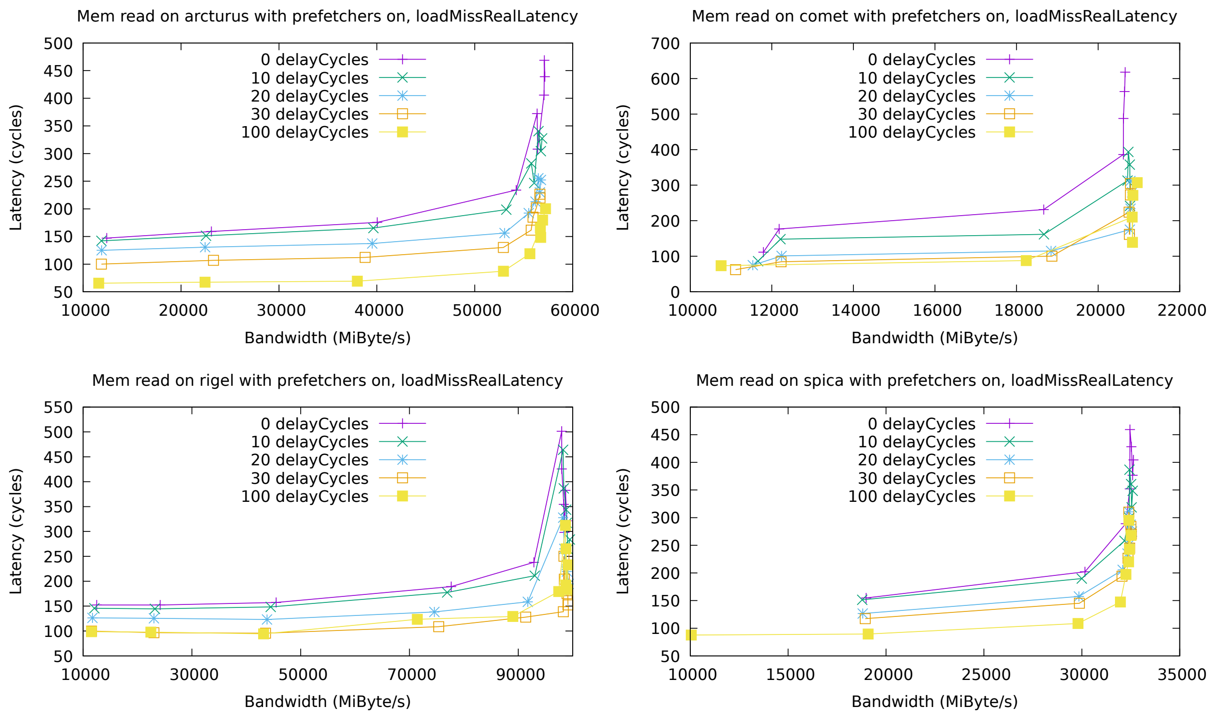


Figure 4.11. The load miss real latency rises with increasing DRAM bandwidth when hardware prefetchers are enabled. Fewer delay cycles result in higher latency. The number of threads controls the bandwidth.

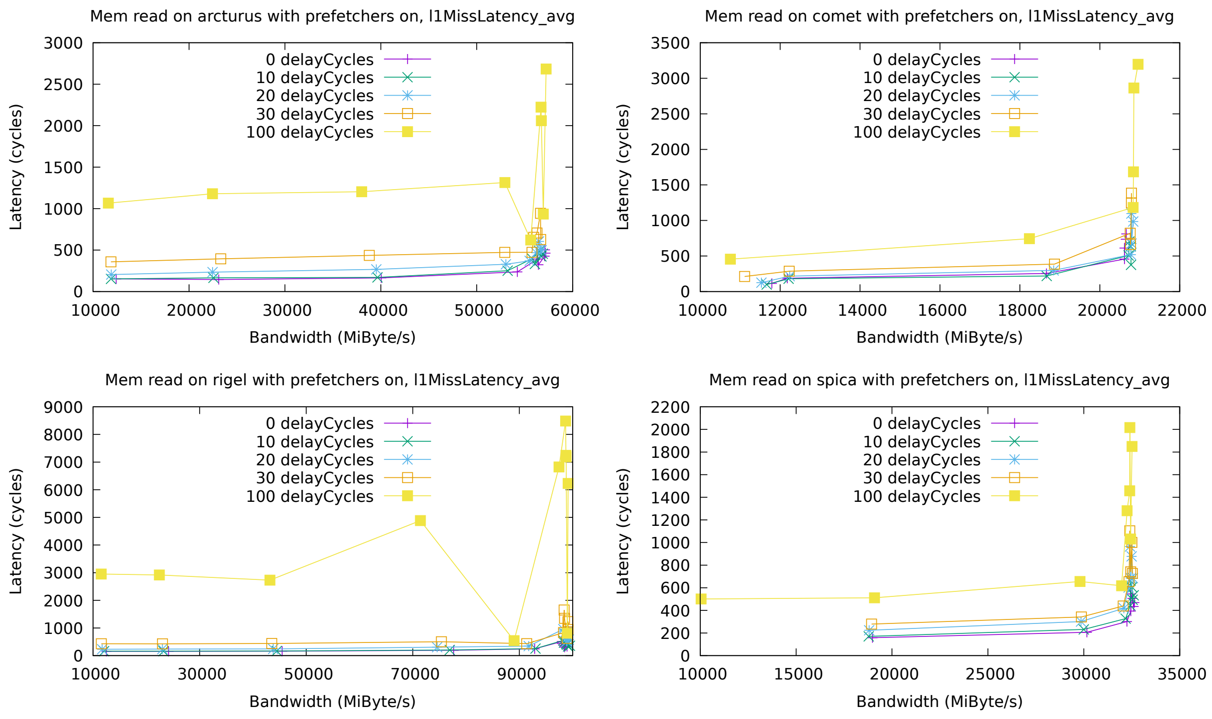


Figure 4.12. The L1 miss latency by DRAM bandwidth when hardware prefetchers are enabled. Fewer delay cycles lead to lower latency. The number of threads controls the bandwidth.

Instruction Sampling Latency Analysis We see in Figure 4.9 that the instruction sampling latency can be high even though the DRAM bandwidth is far from the machine limit. This is a potential disadvantage for the use of the instruction sampling latency for DRAM contention detection. The instruction sampling latency is the time needed for the whole processing of the instruction. It is not only the time it takes to load data. Thus, the instruction sampling latency will increase when there is an in-core limitation. In contrast, the load miss real latency does not increase because it only starts counting after an L1 miss already occurred.

To verify that we looked at counters that express in-core saturation of different resources. Among them is the counter l1d_pend_miss.fb_full. It counts the number of cycles where a demand request was blocked because of unavailable fill buffer slots. Figure 4.13 shows that the number of cycles where a request was blocked due to unavailable fill buffer slots is independent of the memory bandwidth. But it depends on the number of delay cycles. When the hardware prefetchers are turned on (No Figure included), there is the same correlation but the value of l1d_pend_miss.fb_full is never higher than 20.000×10^6. The coefficient of variation of the measurements of the L1D pend miss FB full counter shown in Figure 4.12 is on average 5.94% and at most 11.69%.

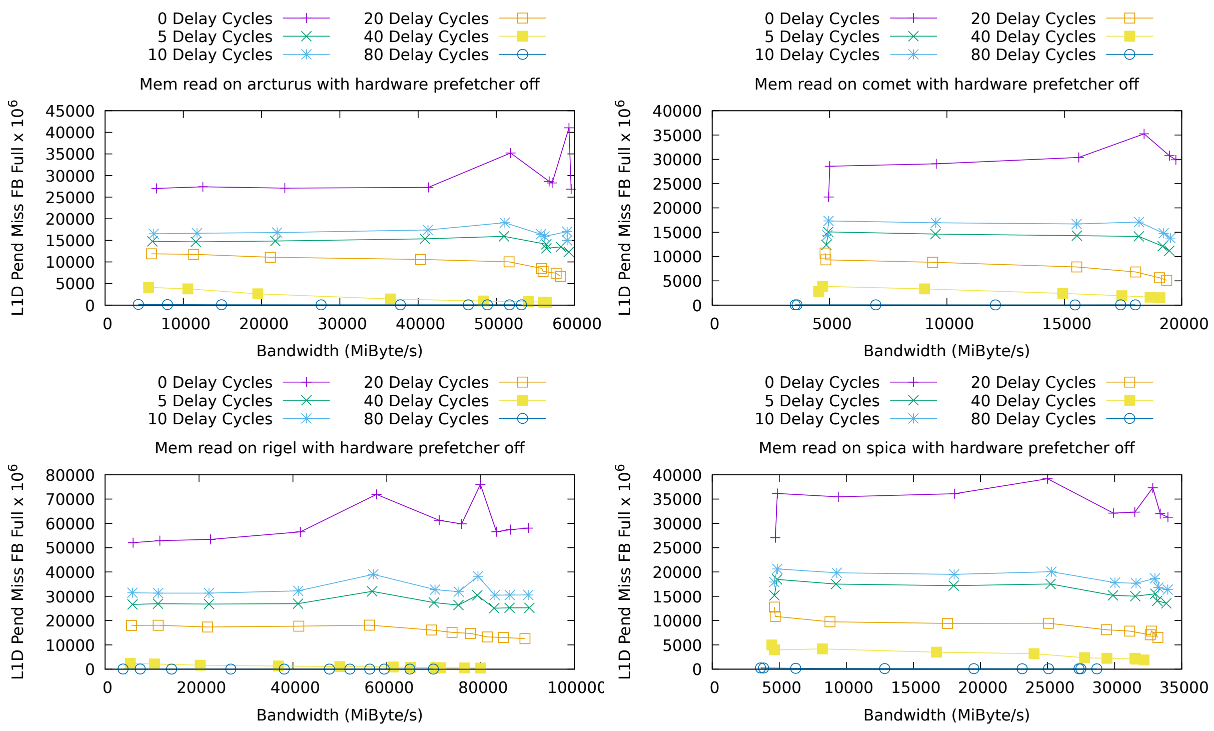


Figure 4.13. The number of cycles where fill buffer slots are unavailable does not depend on DRAM bandwidth but depends on the number of delay cycles.

Variation of the Instruction Sampling Latency The latency of the DRAM access samples is spread over a wide range. Figure 4.14 shows histograms of the instruction sampling latency. The selected benchmark configuration of the mem read benchmark is using eight threads, zero delay cycles, and the hardware prefetchers are turned off. The latency spreads over a wide range with single values of over 3500. When using 20 delay cycles, the latency range becomes narrower, as shown in Figure 4.15. The spread of the latency is similar on all systems. Because the latency can have high variations, it is important to calculate the average based on a large number of samples. Individual samples can have a latency far from the average. Relying on single samples may produce misleading results. A cause for varying latency can be the re-scheduling of requests in the memory controller. To avoid row buffer conflicts, the memory controller may delay requests and prioritize other requests, leading to a different latency of requests. Because some of the requests will be delayed and other requests will be speed up, the average latency of a large number of samples will neutralize the effect of this re-scheduling in the memory controller.

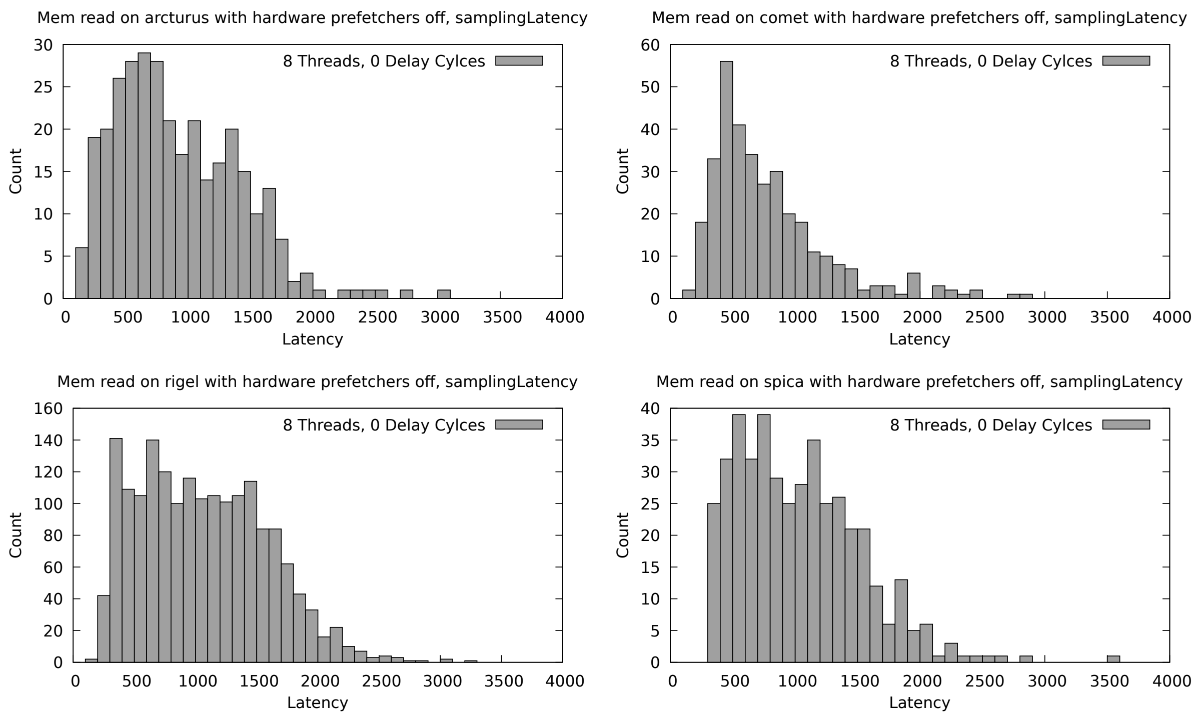


Figure 4.14. Histograms of the instruction sampling latency recorded in one execution of the mem read benchmark with eight threads, zero delay cycles, and hardware prefetchers turned off.

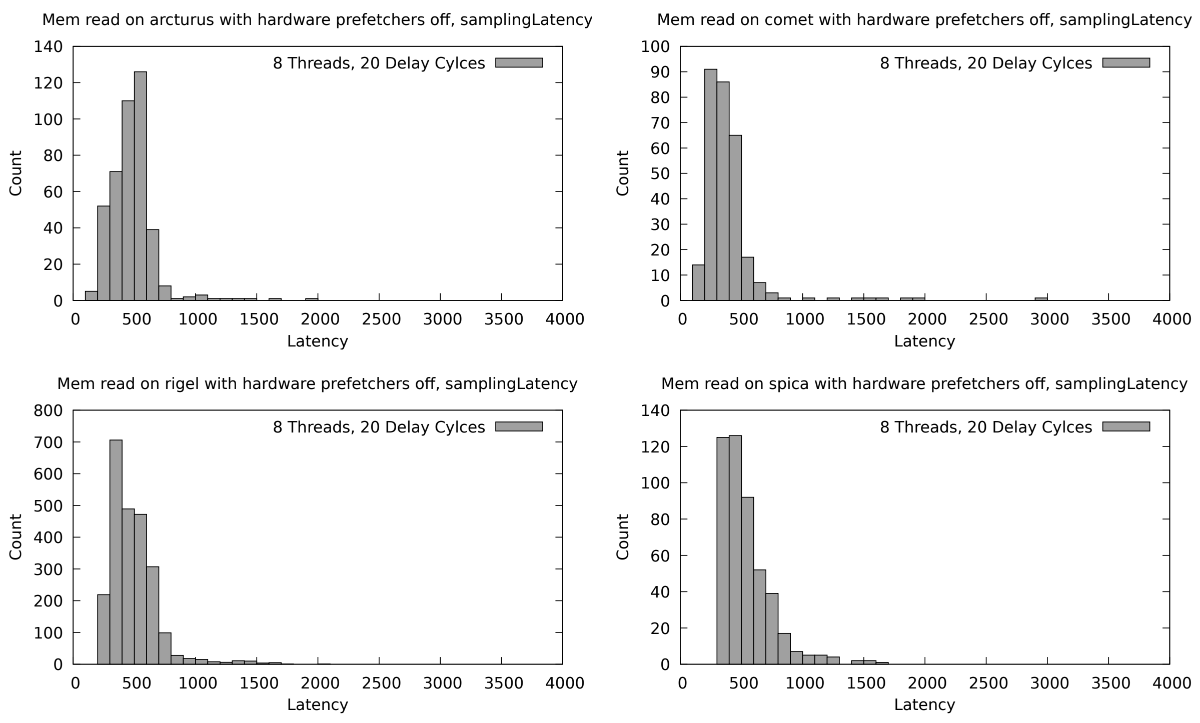


Figure 4.15. Histograms of the instruction sampling latency recorded in one execution of the mem read benchmark with eight threads, 20 delay cycles, and hardware prefetchers turned off.

Differentiation of In-Core and DRAM Limitations To differentiate in-core limitations and DRAM limitations, we propose two solutions. First, use the l1d_pend_miss.fb_full counter like we did in our experiments. When high latency is detected, it can be used to find out if the core itself is limited. Second, count the number of cores that concurrently access the memory. If high latency is observed, but only a few cores access the memory, then the limitation must be in-core because a large number of cores is necessary to saturate the DRAM bandwidth. Because the information from which core an access was triggered is also an attribute of a sample, this method is easy to implement based on instruction sampling data. The exact number of cores that is required to saturate the DRAM bandwidth can be determined by a microbenchmark.

4.3.5 DRAM Contention Detection Method

Based on our findings about memory latency measurement, we use the following method to discover DRAM contention within captured samples. We flag a candidate as suffering from DRAM contention if the DRAM access latency is higher than the base latency of a DRAM access in uncontended state. Based on our results in this section, we conclude that the instruction sampling latency is suitable for this purpose. The detection method is applied to a candidate, that is a pair of a function accessing an object as described in Section 4.1.

Measurement of the Uncontended DRAM Access Latency

The uncontended DRAM access latency is a system characteristic. For measuring the base latency of a DRAM access, we use the ScanRad64IndexUnrollLoop benchmark of pmbw [12]. It is a pointer chasing benchmark. Elements in an array are accessed in random order, and only after a load is completed, the address of the next element is known. Only one access at a time is executed. The processor cannot predict the access pattern. It is the worst case of a local DRAM accesses without contention effects.

We execute this benchmark on a single core with an array size of 2 GB. The benchmark execution is profiled with instruction sampling. We take the latency from the profiled data. This microbenchmark causes TLB misses due to the large array size and random access to array elements. We include this additional delay due to TLB misses in the threshold.

DRAM Contention Detection Conditions

For the practical implementation, not only the latency is important, but other conditions have to be considered as well. The DRAM access latency of a profiled application is calculated as the average latency of all samples that meet the following conditions:

1. The originating function is the current candidate function
2. The accessed address is within the current candidate object
3. The memory access is not locked
4. The memory access hits the local DRAM (or remote DRAM)
5. The memory access hits the TLB

The first two conditions limit the selection to the current candidate. The third condition excludes locked accesses. Those are typically used for atomic accesses and usually have high latency. But those do not have any connection to DRAM contention. The fourth condition limits the selected samples to the local DRAM (or remote DRAM for finding remote DRAM contention) so that we only consider the latency of DRAM accesses and not those of cache accesses. The last condition makes sure that we do not include latency effects that come from TLB misses. The resulting latency is then compared to the uncontended DRAM access latency for the specific system. If the latency is higher than the threshold, the candidate is flagged with DRAM contention.

In addition to those conditions, we make sure that there is an adequate number of samples to draw reliable conclusions. This is required because we often saw a high variance in the latency between individual instructions. First, we require at least 25 samples that meet the above-mentioned conditions. Second, we require a combined DRAM and LFB hit rate of at least 10% in the context of the considered candidate. A better approach would be to statistically verify the reliability of the captured data. But for simplicity, we simply use these two thresholds to make sure that there is a decent amount of samples to draw conclusions from.

4.4 NUMA Imbalance

A common reason for the occurrence of DRAM contention is the imbalanced use of DRAM bandwidth on NUMA systems. Applications that were not designed with NUMA systems in mind, often allocate all memory on only one of the available nodes. Thus, leaving the processor to local DRAM bandwidth of the other nodes unused and causing contention on one node's DRAM. In this situation, the interleaved allocation of the data is a quick solution that can result in a significant performance increase. It would be helpful to know in advance if the DRAM contention is caused by such an imbalanced use of the NUMA resources.

We present a metric that expresses this NUMA imbalance and helps to decide if the interleaved allocation is beneficial for performance. Each sample contains information if local or remote memory was accessed. Together with the originating node of the request, which is known through the CPU id in each sample, we can calculate the NUMA imbalance. The novel NUMA imbalance metric expresses the degree of imbalance of the memory usage in a NUMA system. It is calculated as follows. For each node x, out of all nodes N, the ratio of local memory accesses is calculated first. Then the maximum difference of the local ratios is calculated between all nodes of the system. The resulting ratio expresses the NUMA imbalance within a range of 0 to 1. Where 0 means there is no imbalance, and 1 means there is a high imbalance.

$$LocalRatio_x = \frac{NumLocalAccess_x}{NumLocalAccess_x + NumRemoteAcess_x}$$

$$NumaImbalance = \max_{\forall x \in N}(LocalRatio_x) - \min_{\forall x \in N}(LocalRatio_x)$$

Together with the relative latency, we can determine if there is a performance problem and give optimization advice. It is summarized in Table 4.2. If the relative latency is low, there is no bandwidth related problem. No matter how big the imbalance is, only a minimal performance benefit can be expected from interleaved allocation. If the relative latency is high, a low NUMA imbalance suggests that the bandwidth limitation does not come from the bad use of NUMA resources. If the relative latency and the NUMA imbalance are high, a better allocation is expected to bring a significant performance increase.

Table 4.2. The relative latency and NUMA imbalance guide the optimization process.

Relative Latency	**NUMA Imbalance**	**Performance Problem**
Low	Any	Not bandwidth limited.
High	Low	Bandwidth limited but not NUMA related.
High	High	Bandwidth limited due to inefficient NUMA usage.

Because this metric is based on instruction sampling, it can be calculated for specific function, objects and source code lines. Thus, individual objects, that suffer from the imbalance can be pointed out and differentiated from other objects in the same application that do not suffer from imbalance.

4.5 Channel Imbalance

A DRAM interface can have multiple channels that provide parallelism for memory accesses. If there is an uneven usage of the bandwidth capacity of the channels, the performance will suffer. We present a method to discover such situations of channel imbalance.

In Intel processors, every channel has its own counter for the amount of transferred data. More specifically, it counts the number of transferred cache lines. This counter is referred to as uncore_imc_*/cas_count_read and uncore_imc_*/cas_count_write. The asterisk stands for the channel number. We record the counter value every 500 milliseconds and store the results in an SQLite database. With these numbers, we can create a timeline of the bandwidth usage on each channel. Benchmarks may experience short bursts of high bandwidth, which would disappear in an average over the whole application. With a timeline, they can be seen.

4.6 Row Buffer Miss

Row buffer misses can occur in DRAM devices and hurt performance. Intel refers to the row as page, so we use the term page from now on. We introduce a method to measure the page state and to create a timeline of it. According to Intel documentation [48], there are performance counters from which page misses can be derived. There are three states for a row buffer access that can be distinguished using those counters:

- Empty: No row is currently loaded in the buffer. The requested row needs to be loaded before it can be accesses. This state has medium access latency.
- Conflict: A different row then the requested one is currently in the buffer. The current row must be written back and the requested one must be loaded into the buffer. This results in high access latency.
- Hit: The requested row is in the buffer. This results in the shortest access latency.

According to the documentation [48], the required raw performance events are:

- ACT_COUNT: Counts the number of DRAM activations.
- CAS_COUNT.RD: Counts the number of DRAM reads.
- CAS_COUNT.WR: Counts the number of DRAM writes.
- PRE_COUNT.PAGE_MISS: Counts the number of DRAM precharge events due to page miss.

Based on those raw event counters, the hit rate for each of the row buffer states is calculated as follows:

- PCT_REQUESTS_PAGE_EMPTY = (ACT_COUNT - PRE_COUNT.PAGE_MISS) / (CAS_COUNT.RD + CAS_COUNT.WR)
- PCT_REQUESTS_PAGE_MISS = PRE_COUNT.PAGE_MISS / (CASE_COUNT.RD + CAS_COUNT.WR)
- PCT_REQUESTS_PAGE_HIT = 1 - PCT_REQUESTS_PAGE_EMPTY - PCT_REQUESTS_PAGE_MISS

We record the counter value every 500 milliseconds and store the data in an SQLite database. From this database, we create a visualization with a timeline of the page hit, page empty, and page conflict state.

Chapter 5

Tool Implementation and Features

Performance monitoring hardware changes between processor generations. Previous tools [61, 83, 98], which implement a custom hardware interface, require code changes for new hardware. The data that is collected through instruction sampling can be huge. To process the large data efficiently, a scalable storage format is required. Previously, the data was stored in binary formats [31, 73, 104, 120, 128], which makes it hard to explore the data and difficult to modify existing tools.

The following key ideas in our tool design address these issues. The central component in PerfMemPlus is Linux Perf. Perf is available as part of the Linux kernel and can be run on a variety of Linux based operating systems without modifications, and it comes with regular updates for new hardware. We add a few other software components around Perf to make its use easier and tailored to the analysis of memory accesses. Figure 5.1 shows the software components.

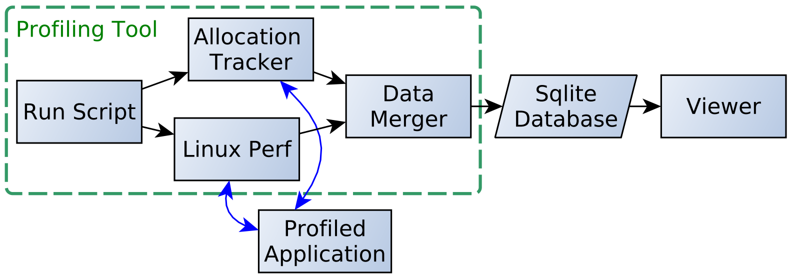


Figure 5.1. The components of PerfMemPlus.

Perf cannot resolve dynamically allocated objects, so we add an allocation tracker to provide this capability and merge the captured data based on timestamps. Perf stores the recorded sampling data in a binary, Perf specific format. Through a scripting interface, we export the data into an SQLite database. The advantage of SQLite is that it is an easily usable data format and separates the recording tool from the analysis tool. Because the instruction sampling data on its own does not provide insights to software developers, we supply a GUI viewer tool with PerfMemPlus. The viewer executes the automatic detection of performance problems as described in Chapter 4 and displays the findings. The viewer also supports the manual exploration of the data. It has a unique approach to guide the user through the data step by step without overwhelming the user with too much information at a time. The following sections describe the details of our tool design and the challenges we had to solve for implementing this tool. It also explains the analysis features of the viewer. PerfMemPlus is open source and the code is available at https://github.com/helchr/perfMemPlus.

5.1 Profiling Tool

The profiling tool of PerfMemPlus records the profiling data and prepares it for analysis. It consists of different components. This section explains the individual components and their interaction.

5.1.1 Linux Perf

Because Linux Perf is developed by the Linux kernel development community, we expect that this tool will be continuously maintained. We can update the underlying Perf without modifications to PerfMemPlus as long as the interface remains stable. We do not use the perf_event_open syscall interface. Instead, we use the command line interface provided by the Perf. This makes the implementation simpler and better isolates the individual components. We do not modify Perf itself because we would have to maintain a forked version of the original Perf. Instruction sampling has the advantage that it can analyze an application without modification of the source code of the application. Only debug information needs to be available in the binary, and fully optimized binaries are supported. Perf already provides all those capabilities.

5.1.2 Run Script

The run script starts the profiling of the application under test and executes the data preparation afterward. To keep the usage easy there are only a few input parameters:

- Sampling period (-c): The rate at which memory read accesses are sampled. A lower value leads to more samples being taken. It results in higher overhead.
- Minimum allocation size (-a): Memory allocations below the specified size are ignored. It can help to reduce the overhead in case there are many small allocations. The ignored allocations are reported as one anonymous object together with stack and static data.
- Output file (-o): The file to save the resulting SQLite database.

The run script configures perf for instruction sampling and starts the profiling process. It uses the options that we have found to work well in our experiments.

The following events are used:

```
cpu/mem−loads,ldlat=1,period=$samplingPeriod/P
cpu/mem−stores,period=$storeSamplingPeriod/P
cpu/cpu−cycles,period=$cycleSamplingPeriod/P
cpu/instructions,period=$cycleSamplingPeriod/P
```

The first event is memory load instructions that have a latency equal to or greater than 1 cycle. The sampling period is selected by the user. For the remaining three events, the cycle sampling period is used. It is the configured sampling period times 1000. It is higher because details about stores are not required, and cycles and instructions occur at a much higher frequency than memory load instructions. Thus, sampling instructions and cycles at the same rate as memory load instructions would cause a high overhead. The remaining three events are memory stores, cycles, and instructions. The cycles are used to calculate the contribution of each function to the total execution time. The instructions, together with the cycles, can be used to calculate the instructions per cycles (IPC). The capital P in the event definition activates the highest precision mode. In the case of those events, it activates the instruction sampling. The run script calls the perf record command with the following parameters:

- –sample-cpu: Enables recording the CPU core number on which a sample was taken.

- -d: Enables recording the data address.
- -W: Enables recording the latency of instructions.
- -e: Specifies the events introduced above.
- -g: Enables capturing call stacks for each sample.
- -k: Specifies the clock source. Details are described in Section 5.1.4.
- -o: Specifies the output file.

The run script also sets the minimum size of allocations to ignore as an environment variable to transfer it to the allocation tracker. The allocation tracker can not take normal input parameters because it loaded as a library at runtime. The allocation tracker reads the environment variable and applies it for profiling.

5.1.3 Allocation Tracker

Together with Perf, we use an allocation tracker. The allocation tracker is based on the one used in the Memprof [7] tool. The memory allocation functions like malloc and free are replaced with the ones defined in the allocation tracker. Those new functions record a call stack, allocated address range, and a timestamp before calling the original memory allocation functions. After the application has finished, the stored data is printed into text files. Data allocated on the stack or statically allocated data can not be resolved using this method. Accesses to this kind of data will show up grouped into one anonymous object.

The original Memprof version does never free memory. This way addresses can not be reused. This makes it easier to process the recorded data. But it alters the default behavior of the profiled applications. In our implementation, we allow the deallocation of memory and handle re-used addresses.

The allocation tracker is using the LD_PRELOAD feature, which is available on Linux systems to replace the existing allocation functions. A shortcoming of this approach is that it can not be used if another library is also replacing the memory allocation functions. For example, some MPI implementations replace the default memory allocators.

Overhead Reduction

We have done several modifications to the original memprof version of the allocation tracker. We have added a feature that allows ignoring small allocations. In some situations, when applications issue many small allocations, excluding those allocations reduces the overhead dramatically. Samples are still collected, making it possible to still find performance problems. Just the attribution to those specific objects is no longer possible.

Compared with the original Memprof version, the amount of printed text is reduced. There was redundant information in the original format, which we removed. This helps to decrease the overhead of tracking and to increase parsing speeds in the data merger.

Benchmarks are often executed as part of scripts, which may call Unix tools like cat, mkdir, dirname, and so on. We check the name of the invoking process and if one of those tools is found, we exclude them from the allocation tracking. This is another measure to reduce the overhead. We have encountered cases where several hundreds of MB of allocation tracker data was accumulated because of such Unix tools.

5.1.4 Perf Export and Data Merger

Perf stores the recorded sampling data in a binary, Perf specific format. Perf provides a scripting interface. This interface allows accessing the Perf data. We use this interface and a python script to export the data to SQLite. The script is based on an example already provided with Perf [65]. We decode all bitfield datatypes to support SQL queries. By using cTypes package, this can be achieved with high performance. Using this database, the sampling data is much easier to process compared to the Perf binary format.

A few changes in the original export script were necessary to improve the performance of the SQLite export. Journaling and synchronous file operations are turned off. Samples, that come from perf are first stored in a buffer instead of inserting them directly into the database. Up to 5000 elements are stored in the buffer. When the buffer is full, all previously cached samples are inserted into the database in one transaction.

5.1.5 Correlation of Samples and Allocation Data

We use two independent tools to profile the execution of an application. Perf does the instruction sampling, collecting data about the memory access, including the accessed address. The allocation tracker records data about the dynamic memory allocations. After the profiling is finished, we merge the data captured by the two tools.

Dynamic memory allocations are only valid for a certain time period. Thus, to make a lookup from a given accessed data address to an allocation call stack, the timestamp of the sample and the interval in which the allocation was valid has to be considered. Both tools individually record a timestamp for samples and allocations. In order to have comparable timestamps from both sources, both must rely on the same clock source. Perf provides a parameter (-k) to specify the clock source. We use the CLOCK_MONOTONIC clock source, which returns a timestamp counted in nanoseconds from the startup of the system.

A point to look out for when using this counter is the synchronization of the hardware clock source between multiple cores and sockets of the same system. The CLOCK_MONOTONIC is internally based on the Time Stamp Counter (TSC), which is present in all modern x86 processors. The TSC is not affected by power management, which may change the operating frequency of the processor. It is also synchronized at startup with all cores across all sockets. Some older processors might not have this feature, but this can be verified by checking the presence of the constant_tsc and nonstop_tsc flags.

By correlating the process id and timestamp, we can then find the corresponding data objects for a given address. This lookup is done entirely using SQL queries. SQLite already provides a method for efficiently selecting an address and timestamp within ranges. Thus our implementation is simpler than existing approaches [61] that rely on balanced trees to implement the range search. This update process involves many database queries and updates. High performance is achieved by the following implementation details. Synchronous file operations and journaling are turned off. Inserts and updates are buffered and inserted in one transaction when the buffer is full. Indexes and temporary tables are created to enable quicker lookup of often queried data.

5.1.6 Resolving Instruction Pointers

An important feature of a profiling tool is to resolve addresses to source code locations. For resolving binary names and function names, we rely on Linux Perf. It uses the debug info in applications to resolve and demangle function names. The instruction pointer values are also exported. The allocation tracker only exports addresses. Thus, we still need to resolve the addresses from the allocation tracker and the data from Perf in case we want to know source files and source code lines. This translation is done using addr2line from GNU Binutils. To avoid the overhead of starting a new process for every address resolution, a cache of processes is held. For each binary one active process is kept open. Addresses are then sent to this process, and addr2line returns the function name, source file, and line number. The path to the profiled binaries is taken from Perf. The path is also stored in the SQL database.

5.1.7 Attribution of Samples with Counter Values

We introduce a generic method to combine instruction samples with performance counters. It is useful because it allows exploring the possibilities of performance counters together with instruction sampling. This method works as follows.

In an exported perf database are entries for performance counters. An entry includes a timestamp, at which the counter was read, and how many events have been counted since the last time the counter was read. We define a window of counter readings. For example, a window of 100 counter reads. For each window, the counter values are summed up and then divided by the time span of the window. At each recorded data point, Perf reports the increase of the counter for that specific interval. For each window, the instruction samples that fall within the time window, are extended with the average counter increase in this interval.

For core-local counters, Perf reports the results for each core and thread individually. Perf takes care to count correctly on context switches or thread migrations. When attributing the samples with counters, the core and thread information must match the counter readings. Care must be taken for other counters, which count globally for the whole socket. Then the values reported by perf still contain a thread and core. However, this is the core and thread which has read the counter. The counter value itself still applies for all cores of that socket and all threads running on this socket during this time.

HPC applications often use many threads and cores. Applying the above-mentioned method for each possible thread and core combination would create a large overhead in data post-processing. Thus, we only take into account thread and core combination which have enough counter readings to form at least one interval. This check can be done with a simple and fast SQL query.

5.1.8 NUMA Configuration

Especially for NUMA related problems, it is useful to know the CPU core to CPU socket mapping. After the profiling, during the processing of the captured data, the hardware layout is read using the numactl –hardware command. The information is then added to the database. This information allows to do queries for specific nodes such as the example in Figure 5.14 without looking up external information about the NUMA configuration.

5.2 Database Format

The SQLite database connects the profiling tool and the viewer. Because of the SQLite-based data format, it is possible to keep the profiling tool and use another custom tool for visualization of the data and vice versa. The relational data model and SQL are suitable for instruction sampling data. Aggregations to specific functions or objects, sorting by certain attributes, and selecting ranges can be done using SQL queries. Researchers can easily access the data and extract performance relevant information. Unlike binary formats, where parsing data structures, iterating through them, and manually aggregating them to useful views is necessary. The implementation of the viewer component of PerfMemPlus serves as an example. It is essentially a set of predefined SQL queries that have proven to show useful views for performance analysis. SQLite as a data format gives scalability with little implementation effort. Big trace files can be stored on disk, and indexes enable quick lookups to find the required data. Any other relational database would also satisfy these requirements. But we chose the file-based SQLite database because it does not require the installation of databases, and file-based operations are familiar to most users.

The database is mostly based on the format specified by Perf. Because there is no documentation about this format, we explain it here. We also describe our modifications to the data format. The original SQLite format that comes with Perf is designed to hold samples related to memory accesses and samples related to branch instructions. Thus some of the fields in the database have names that rather fit branch instructions, and some are even unused when exporting memory access samples from Perf. We omit those unused fields in this description. The metadata table contains the command line of the program, which was profiled, the sampling period, and the minimum allocation limit of the allocation tracker.

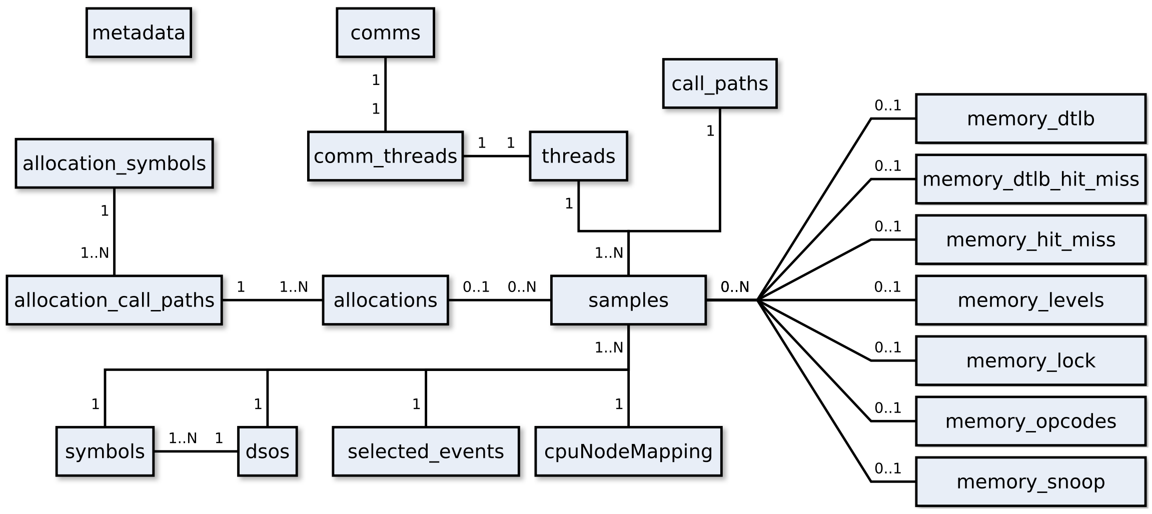


Figure 5.2. The entity relationship diagram of the instruction sampling database.

As shown in Figure 5.2, the samples table is the central one. It contains all the recorded samples. Many attributes are not stored in the samples table directly. Instead, the data is stored in tables connected by foreign keys. The tables on the right side of Figure 5.2 that start with memory are a decoded version of the data_src field in the samples table.

There are the following attributes for each sample:

- id: A unique id for each sample.
- evsel_id: The type of event that was recorded with this sample
- thread_id: Thread id. The OS pid and tid has to be taken from the linked table
- comm_id: The command name of the application from which this sample was taken.
- dso_id: The name and path of the executable.
- symbol_id: The function name and further information about the functions such as address range and dso.

- symbol_offset: The offset within the symbol where the instruction pointer was at the time the sample was taken.
- ip: Instruction pointer.
- time: Timestamp where the sample was taken. In our configuration, it is the time in nanoseconds since the start of the system. Samples in the database are not guaranteed to be ordered by time.
- cpu: CPU on which the sample was collected.
- to_ip: Accessed data address. The name comes from the branch sample export and for branch instructions, it shows the target of a taken branch.
- period: The number of events since the last sample was taken on the core and within the thread.
- weight: Latency of the instruction.
- data_src: A bit-field that encodes various information about the memory access. It is decoded in the following attributes.
- memory_opcode: Type of memory access. For example, read or write access.
- memory_hit_mist: Cache hit or miss.
- memory_level: Memory hierarchy level where data was found. It needs to be combined with the cache hit or miss field.
- memory_snoop: Memory snoop and coherency protocol status.
- memory_locked: Locked memory transaction information.
- memory_dltb_hit_miss: DTLB hit or miss.
- memory_dltb: DTLB hierarchy level where lookup was found. It needs to be combined with the hit dltb hit or miss attribute.
- call_path_id: A link to the first entry in the call path.
- allocation_id: Allocated memory including address range, time interval and call stack of the allocation.

5.3 Data Analysis and Viewer Tool

The viewer can open profiling results and display them. It supports automatic and manual performance analysis in different views.

5.3.1 Auto Analysis

The automatic analysis creates a list of candidates, applies the false sharing detection, applies the DRAM contention detection, and finally reports the results.

False Sharing Detection

The conditions that are checked to identify false sharing are described in Section 4.2. We implement this condition check as a three-stage process. First, we generate a temporary table. This table is a filtered selection of all samples. Only the load and store samples are included. The attributes are limited to the ones needed for false sharing detection. All further queries are done against this temporary table. By using the temporary table, we are can speed up the detection process.

Second, we find the cache lines that we need to consider. Those are all the cache lines, that have accesses with the HITM flag set. Those cache lines can be obtained with the SQL statement in Figure 5.3.

```
select distinct cl from samplesForFs -- a tempory table to speed up quries
where allocation_id in (select id from allocations where call_path_id = ?)
and memory_snoop =
(select id from memory_snoop where name = "Snoop_Hit_Modified")
and symbol_id = ?");
```

Figure 5.3. The SQL statement to get cache lines that have accesses with the HITM flag set.

Third, we check if accesses to those cache lines meet the conditions. The query in Figure 5.4 returns all memory accesses and the attributes that are required to check the conditions. All returned memory access samples are checked against each other to see if the conditions are met by any pair of samples.

```
select t_ms, to_ip, thread_id, memory_opcode, allocation_id, ip from samplesForFs
where cl = ? and allocation_id in (select id from allocations where call_path_id = ?)
and symbol_id = ?
```

Figure 5.4. The SQL query to obtain access samples that access a specific cache line.

Main Memory Bandwidth Contention Detection

First, the average latency of memory accesses that satisfy the conditions specified in Section 4.3.5 is obtained. If the latency is higher than the threshold, that is defined in a configuration file, the considered candidate is flagged with DRAM contention. The DRAM access latency is obtained with the query in Figure 5.5.

Second, the conditions for an adequate sample size are checked. The number of samples is determined using the query in Figure 5.6. If there are not enough samples, another flag will be set as a warning for the user.

```
select avg(weight) from samples where symbol_id = ? and
allocation_id in (select id from allocations where call_path_id = ?) and
evsel_id = (select id from selected_events where name like "cpu/mem-loads%") and
memory_level = (select id from memory_levels where name = "Local_DRAM") and
memory_dtlb_hit_miss = (select id from memory_dtlb_hit_miss where name = "Hit") and
memory_lock != (select id from memory_lock where name = "Locked")
```

Figure 5.5. The SQL query to get the average latency of accesses to the local DRAM.

Detection Results Report

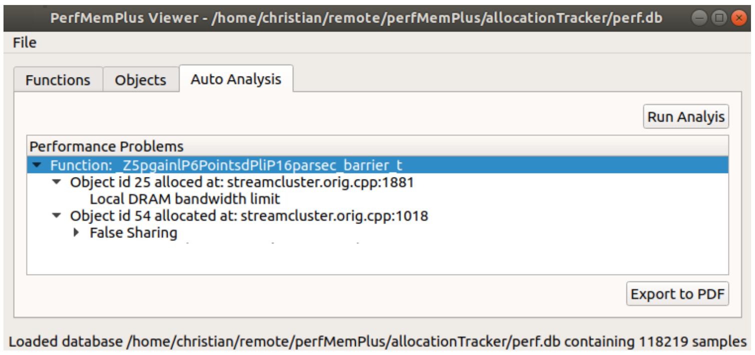
The auto analysis view has a button in the upper right to start the analysis process. Once the results have been obtained, they are displayed in the white area below.

Performance problems and their locations are displayed as a list of trees. The uppermost element in a tree is the function name. The child of the function is a list of objects that are accessed by this function.

```sql
select count(*) from samples where symbol_id = ? and
allocation_id in (select id from allocations where call_path_id = ?) and
evsel_id = (select id from selected_events where name like "cpu/mem-loads%") and
memory_level = (select id from memory_levels where name = "Local␣DRAM") and
memory_lock != (select id from memory_lock where name = "Locked")
```

Figure 5.6. The SQL query to get the number of samples that access the local DRAM.

An object is listed with its ID and with the file and source code line where it was allocated. Those two tree levels identify the location of a performance problem. With this information, the user can easily look up the location in the source code or go to the manual analysis features to look at more details of those functions and objects. Finally, the last item in the tree is a list of performance problems. In the example in Figure 5.7 we can see one function with two objects, each with one performance problem. The performance problems are local DRAM limitations when the function accesses the first object with ID 25 and false sharing when the same function accesses the object with ID 54.


PerfMemPlus Viewer - /home/christian/remote/perfMemPlus/allocationTracker/perf.db
File
Functions
Objects
Auto Analysis
Run Analyis
Export to PDF

Performance Problems
Function: _Z5pgainlP6PointsdPliP16parsec_barrier_t
Object id 25 alloced at: streamcluster.orig.cpp:1881
Local DRAM bandwidth limit
Object id 54 allocated at: streamcluster.orig.cpp:1018
False Sharing

Loaded database /home/christian/remote/perfMemPlus/allocationTracker/perf.db containing 118219 samples

Figure 5.7. The auto analysis view of PerfMemPlus. In this example, it shows a case of false sharing and local DRAM bandwidth limitation.

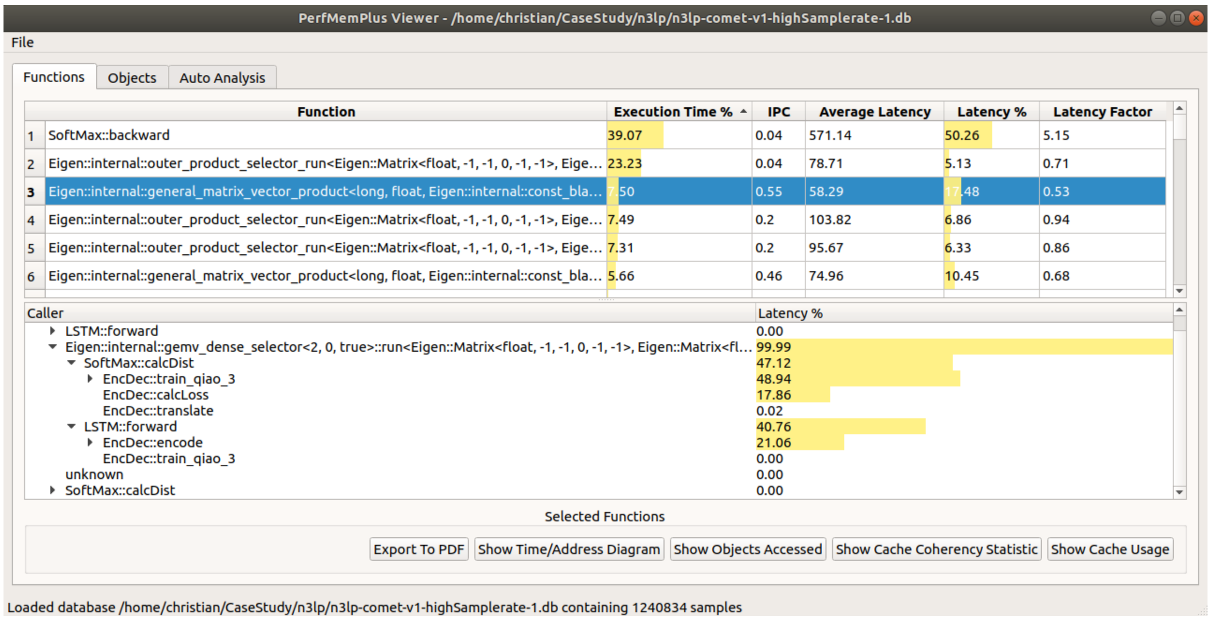
5.3.2 Manual Analysis Features

Because automatic discovery is only supported for two specific performance problems, manual exploration of sampling data is also supported. There are views of cache hit rates and latencies in different cache levels, as well as a visualization of access patterns and data sharing between threads. The main idea for navigation through the data is to use functions, objects, and their connection. The data can be explored from both sides. There are a function profile and an object profile in the main window. After a function, object, or combination of both is selected, details of the selection can be shown in separate windows.

Function Profile

A screenshot of the function profile is shown in Figure 5.8. The upper half shows the function profile. It consists of the names of functions and metrics. In this manual exploration view, the user can first identify the functions that contribute most to the execution time, like it is done with a traditional profiler. Next, the user can rely on the memory metrics to identify if there memory-related performance problems. Those memory metrics are the average memory access latency, the latency contribution in percent, and the latency of the function in relation to the latency of the whole application (Latency Factor). The average memory access latency is a good indicator of performance problems. Because inefficient use of the memory system (cache miss, remote memory access, TLB miss) will increase the latency. The latency contribution in percent and latency factor can identify the heaviest memory accesses in an application.

The lower half shows the call stack for one of the selected functions, together with the latency contribution coming from each function. It is a latency based profile, to know from where the slow memory accesses are issued. This is particularly useful for cases where library functions are reported in the function profile and the user wants to know from where in the programmer's own code they are called. On the bottom are buttons that execute an action for the selected function. They will open another window with more details.


PerfMemPlus Viewer - /home/christian/CaseStudy/n3lp/n3lp-comet-v1-highSamplerate-1.db
File
Functions
Objects
Auto Analysis

	Function	Execution Time %	IPC	Average Latency	Latency %	Latency Factor
1	SoftMax::backward	39.07	0.04	571.14	50.26	5.15
2	Eigen::internal::outer_product_selector_run<Eigen::Matrix<float, -1, -1, 0, -1, -1>, Eige…	23.23	0.04	78.71	5.13	0.71
3	Eigen::internal::general_matrix_vector_product<long, float, Eigen::internal::const_bla…	7.50	0.55	58.29	17.48	0.53
4	Eigen::internal::outer_product_selector_run<Eigen::Matrix<float, -1, -1, 0, -1, -1>, Eige…	7.49	0.2	103.82	6.86	0.94
5	Eigen::internal::outer_product_selector_run<Eigen::Matrix<float, -1, -1, 0, -1, -1>, Eige…	7.31	0.2	95.67	6.33	0.86
6	Eigen::internal::general_matrix_vector_product<long, float, Eigen::internal::const_bla…	5.66	0.46	74.96	10.45	0.68

```
Caller                                                                                       Latency %
  ▸ LSTM::forward                                                                            0.00
  ▾ Eigen::internal::gemv_dense_selector<2, 0, true>::run<Eigen::Matrix<float, -1, -1, 0, -1, -1>, Eigen::Matrix<fl…  99.99
      ▾ SoftMax::calcDist                                                                    47.12
          ▸ EncDec::train_qiao_3                                                             48.94
            EncDec::calcLoss                                                                 17.86
            EncDec::translate                                                                0.02
      ▾ LSTM::forward                                                                        40.76
          ▸ EncDec::encode                                                                   21.06
            EncDec::train_qiao_3                                                             0.00
    unknown                                                                                  0.00
  ▸ SoftMax::calcDist                                                                        0.00
```

Loaded database /home/christian/CaseStudy/n3lp/n3lp-comet-v1-highSamplerate-1.db containing 1240834 samples

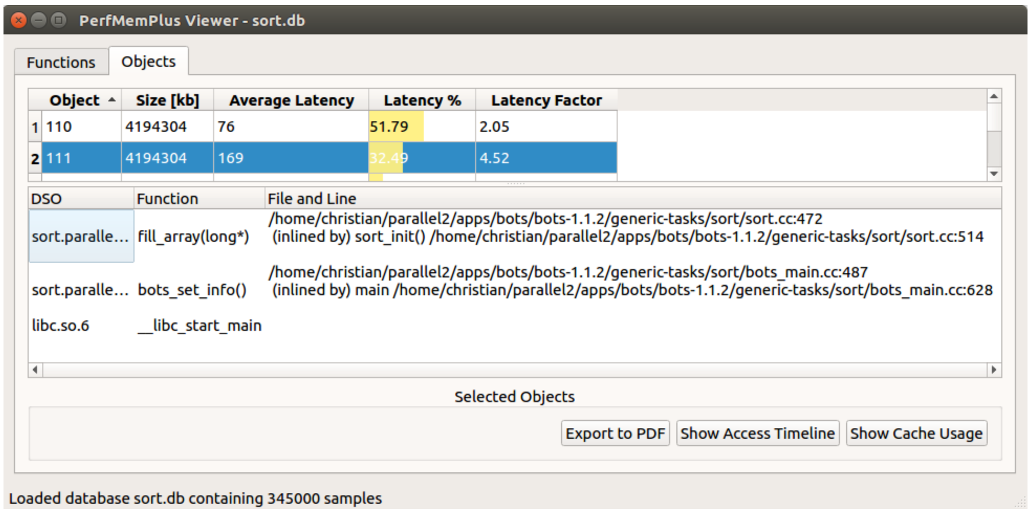
Figure 5.8. The function profile of PerfMemPlus. It shows traditional metrics like execution time together with memory-related metrics. On the bottom is a call stack that shows latency contribution.

Object Profile

The object profile, that is shown in Figure 5.9, is another way to start a performance analysis of an unknown program. It lists all of the dynamically allocated objects in the program. Static and stack objects are put together into one anonymous object. The table on the top shows the objects arranged in rows. The columns show the attributes and metrics of the objects. The size of the object that is determined through the allocation call. The average latency of all read accesses to this object and how much the accesses to this object contribute to the overall latency of the application is also shown. Finally the latency factor. It shows the relation of the average latency of accesses to this object compared to the average memory access latency of the whole program. For the currently selected object, the allocation call stack is shown in the bottom half of the window. In includes the object file, function name, and exact location in the source code. Inlined functions can be resolved as well.

Accessed Objects

After one, or multiple functions have been selected, the objects that are accessed by those functions can be shown. The view (Figure 5.10) is similar to the object profile. The difference is that the object profile shows the objects in the context of the whole application, whereas this window shows it in the context of the selected function. The objects are shown as a latency profile, to make it possible to select the object accesses that contribute most to the latency. It allows getting an overview of unfamiliar code. A


PerfMemPlus Viewer - sort.db

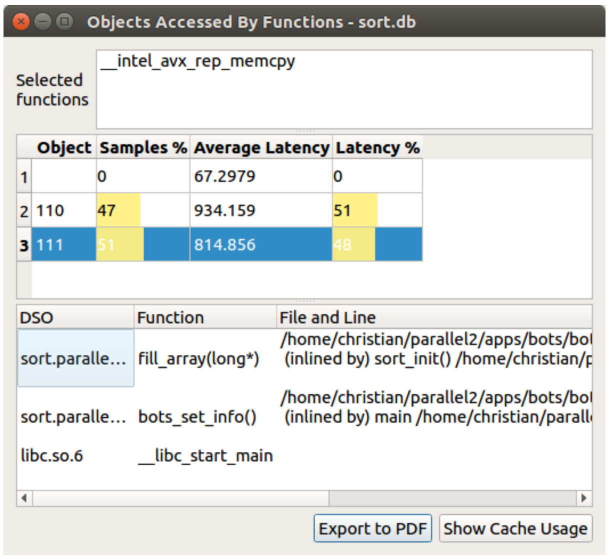
	Object	Size [kb]	Average Latency	Latency %	Latency Factor
1	110	4194304	76	51.79	2.05
2	111	4194304	169	32.49	4.52

DSO	Function	File and Line
sort.paralle…	fill_array(long*)	/home/christian/parallel2/apps/bots/bots-1.1.2/generic-tasks/sort/sort.cc:472 (inlined by) sort_init() /home/christian/parallel2/apps/bots/bots-1.1.2/generic-tasks/sort/sort.cc:514
sort.paralle…	bots_set_info()	/home/christian/parallel2/apps/bots/bots-1.1.2/generic-tasks/sort/bots_main.cc:487 (inlined by) main /home/christian/parallel2/apps/bots/bots-1.1.2/generic-tasks/sort/bots_main.cc:628
libc.so.6	__libc_start_main	

Loaded database sort.db containing 345000 samples

Figure 5.9. The object profile of PerfMemPlus. It lists the dynamically allocated objects of the application along with their latency metrics. The call stack on the bottom shows where the objects are allocated.

user can comprehend which functions access which objects. It is part of a two-stage selection process of interesting functions and objects, to limit the focus on a particular function accessing a specific object.


Objects Accessed By Functions - sort.db

Selected functions: __intel_avx_rep_memcpy

	Object	Samples %	Average Latency	Latency %
1		0	67.2979	0
2	110	47	934.159	51
3	111	51	814.856	48

DSO	Function	File and Line
sort.paralle…	fill_array(long*)	/home/christian/parallel2/apps/bots/bot (inlined by) sort_init() /home/christian/p
sort.paralle…	bots_set_info()	/home/christian/parallel2/apps/bots/bot (inlined by) main /home/christian/parall
libc.so.6	__libc_start_main	

Figure 5.10. The accessed objects window. It shows an object profile of all objects that are accessed by a previously selected function.

Time Address Diagram

The time address diagram can be used for recognizing memory access patterns and data sharing across threads. It is shown in Figure 5.11 and Figure 5.12. The horizontal axis represents the execution time of the application. The vertical axis is the address range that is accessed. Each point in the diagram represents a memory access sample. Different colors represent different threads. At the top right either all threads, that access the selected object or a single thread can be selected. Data sharing between threads and memory access patterns are hard to see from the source code. This visualization

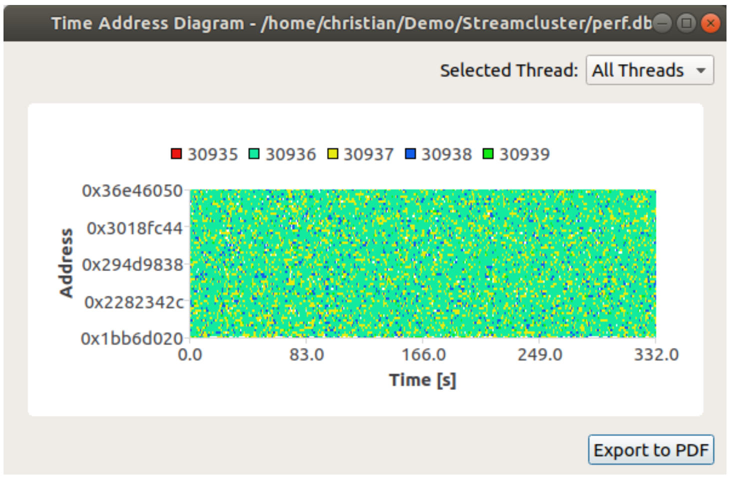


Figure 5.11. The time address diagram of PerfMemPlus. This case shows inefficient data sharing between threads.

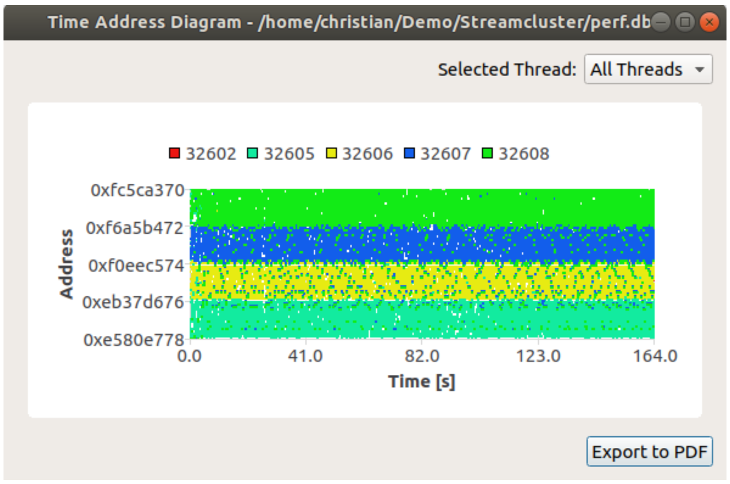


Figure 5.12. The time-address diagram of PerfMemPlus. This case shows a large array that is split and assigned to individual threads.

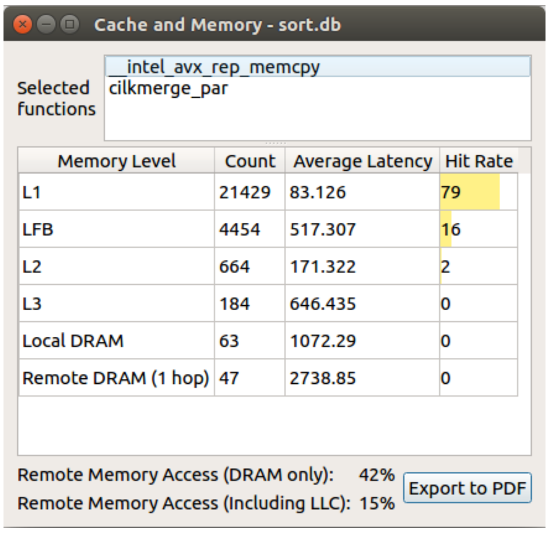
helps to understand the data accesses and spot optimization opportunities. The example in Figure 5.11 shows an application with intensive data sharing between threads. In contrast, Figure 5.12 shows the same application with an optimized data access pattern. Each thread stays within the same address range.

Because there can be a large number of points that need to be drawn, it is important that the drawing is done fast. We rely on the QT drawing library. It supports drawing with OpenGL support if the drawn items are simple, like dots and lines. Here we only need to draw dots. Drawing with OpenGL support speeds up the drawing significantly. Without OpenGL support, the drawing speed is often too slow. Especially in cases with many samples, it can become unusable.

Cache and Memory Details

This window shows cache access details for previously selected functions and objects. A screenshot is shown in Figure 5.13. On the top, there are the selected functions for which the cache access details are shown in this window. There could also be objects or a combination of objects and functions in this window.

In the table below are the memory levels arranged in the rows. In the columns, from left to right are the number of samples, the average latency, and the hit rate relative to all samples. A bar graph is overplayed with the number to quickly see the hit rate in each cache level. On the bottom, the percentage of remote accesses is shown. First, as the ratio between remote and local memory access. Second, as the ratio of local L3 hits plus local DRAM hits against the number of remote L3 hits and remote DRAM hits. With this view, it is possible to diagnose locality issues, remote memory access issues and high latencies in specific cache levels.



__intel_avx_rep_memcpy
cilkmerge_par

Memory Level	Count	Average Latency	Hit Rate
L1	21429	83.126	79
LFB	4454	517.307	16
L2	664	171.322	2
L3	184	646.435	0
Local DRAM	63	1072.29	0
Remote DRAM (1 hop)	47	2738.85	0

Remote Memory Access (DRAM only): 42%
Remote Memory Access (Including LLC): 15%

Figure 5.13. The cache and memory window of PerfMemPlus. It shows the memory hit rates and latencies for a specific function and object selection.

Imbalanced Memory Usage In NUMA Systems

The instruction sampling data contains information, that allows differentiating if data was loaded from a remote DRAM or the local DRAM. We can use this information to find unbalanced use of the memories. This unbalanced use occurs if an object is allocated on one node but is later accessed from other nodes. In this situation, the interleaved allocation of the object can often help to increase the performance, because the memory bandwidth of both nodes can be utilized. The SQL query in Figure 5.14 shows how to obtain the required information. The result is a table that shows for each node how many local and remote memory accesses were recorded.

```sql
select (select node from cpuNodeMapping where cpu = s.cpu) as node,
(select name from memory_levels where id = s.memory_level) as memory,
count(*)
from samples s where
memory_level in (16, 32) --16 is local dram, 32 is remote dram
and allocation_id = ?
and symbol_id = ?
group by node,memory_level
```

Figure 5.14. The SQL query to show the number of accesses from each node to its local DRAM and remote memories.

For example, in a two-socket system, if there is an imbalance, one of the nodes will access mostly the local DRAM, and the other node will mostly access the remote DRAM. In an evenly balanced situation, both nodes access the local DRAM and remote DRAM equally. We demonstrate the usage of this feature in the case study of canneal in Section 7.9.1.

Objects That Share Cache Lines

For the confirmation of inter-object false sharing, it can be useful to confirm the allocation of several objects in the same cache line. Because all dynamic memory allocations are tracked, such an allocation can be confirmed using the profiling data and the query in Figure 5.15.

```sql
select call_path_id from (
select *,address_start/64 as cla, address_end/64 as cle from allocations where call_path_id = ?) a
inner join (
select *,address_start/64 as cla, address_end/64 as cle from allocations where call_path_id = ?) b
where (a.cle = b.cla or a.cla = b.cle) --Beginning and end are in the same cache line
and a.id != b.id --Exclude the same object
and (a.time_start between b.time_start and b.time_end --Object is alive at the same time,
or b.time_start between a.time_start and b.time_end) --otherwise it could be a reused address
```

Figure 5.15. The SQL query to verify if objects are allocated in the same cache line.

The query simply checks if the end of one allocation is in the same cache line as the beginning of another allocation. Additionally, it makes sure that those objects are alive in the same time period. Otherwise, they could just be objects allocated at recycled nearby addresses. The allocation call path of detected objects is returned to find them in the source code.

Chapter 6

Reverse Engineering of Intel DRAM Addressing Using Performance Counters

The memory subsystem of a modern computer is complex. The memory is split into different channels to provide higher bandwidth. Organization of DRAM chips in bank groups and banks provide the opportunity for pipelining requests. This has led to increased throughput of DRAM systems over the years without a significant performance increase in the single DRAM cell [33]. The memory controller must interface with those different DRAM components and address them individually. The memory controller receives requests to load data at specific physical addresses. From the physical address, the DRAM controller must determine the channel, rank, bank, row, and column in which the data is stored. The definition of how addresses are translated is called DRAM address mapping.

If the DRAM address mapping is known, it enables a wide range of applications. In hardware architecture and system software research, approaches for better usage of memory channels and banks are being explored. For example, application-aware memory channel partitioning [89], adapted page sizes for better usage of row buffers [114], efficient use of new hybrid memory technology [130], effects of unreliable memory [2], or DRAM layout aware memory allocators [14, 93, 132]. For the evaluation of such new concepts, simulated hardware is often used. If the address mapping is known, such a system can be implemented and evaluated using real processors and applications. The few cases where evaluations are done using real hardware rely on specific processor models where the address mapping is documented or require manual reverse engineering effort. For example, Chandru and Mueller [14] use a Tilera processor, Pan et al. [93] use an AMD Opteron 6128 processor, and PALLOC [132] is implemented on a Xeon W3530 and Freescale P4080. The above-mentioned concepts also showed performance gains which we cannot utilize on other machines without knowing the address mapping.

Researchers in IT security are also interested in the DRAM address mapping to evaluate their concepts. Covert communication channels across CPUs were demonstrated by Pessl et al. [96]. A variation of Rowhammer [58] attacks was introduced by Gruss et al. [34]. And Song et al. show a method for hiding rootkits [110].

The address translation is done in hardware, it is different for every system, and it is mostly undocumented except for a few specific processor models. For example, the documentation of the outdated Intel Xeon 5500 contains a description of the address mapping [46]. However, for newer generations of Intel processors, this information is not published. Up to a certain extent, the mapping is configurable in hardware. At the startup of the system, the BIOS reads the DIMM configuration and programs the configuration registers in the processor. After the initialization, the configuration cannot be changed. The mapping can also be influenced by the BIOS settings. For example, the activation of on-chip

NUMA domains changes the addressing [29]. Thus the address mapping depends on many factors and may be different for every system. A general addressing function, for example for a specific processor generation, does not exist.

We introduce an automatic and reliable method for reverse engineering the DRAM addressing on Intel processors. Our tool can automatically find the addressing functions of memory channels, ranks, bank groups, and banks. It is available online at https://github.com/helchr/reMap. The main idea is to directly and reliably measure the number of accesses to each component (e.g., bank), unlike existing approaches that leverage unreliable numbers such as access latency or the number of bit-flips by rowhammer. With a smart selection of probed addresses, we can determine the addressing functions in a short time. Because we use boolean algebra to resolve the addressing functions, we can differentiate measurement errors from insufficient sampling. In the case of asymmetric DIMM population, we gather additional information from configuration registers. Our tool supports server-class processors with a large amount of memory and also supports asymmetric DIMM population. We demonstrate that we can reverse engineer the address mapping on an Intel Haswell, two Broadwell, and a Skylake system. They are equipped with up to 2TB of RAM and include a system with asymmetric memory channel population. Based on benchmarks that access certain components of the DRAM using the reverse engineered address mapping, we confirm that our method can find the correct address mapping on recent Intel server-class processors.

6.1 Methodology

Our reverse engineering approach has two steps. In the first step, pairs of the physical address and accessed component are collected. This is done by picking an address from a pool and then finding the component that this address accesses. The component is identified using performance counters. After this process, there is a list of physical addresses and the component that the address accessed. In the second step, address mapping functions are calculated from this list of samples. The steps are summarized in Algorithm 1. The remainder of this section explains the individual steps.

Algorithm 1 Pseudocode of the reverse engineering method.

```
Allocate pool of memory
while not enough addresses do
    Pick a virtual address from the pool
    Get the physical address
    for all components do
        Set up measurement for component
        Repeatedly access the address
        if counter value > number of accesses then
            Record pair of physical address and component
        end if
    end for
end while
Calculate mapping function from samples
```

6.1.1 Memory Allocation

The DRAM address mapping is based on the physical address. If we can control individual bits of the physical address, we can use a structured address selection method, such as the one described in Section 6.1.2. Only the bits that express offset within a page frame directly translate to bits of the physical address. The bits for the frame number are not under our control. Thus we increase the page size to 2MB, which leads to 21 bits of the physical address being under the direct control of our tool.

6.1.2 Address Selection

Theoretically, it is possible to use random addresses from a large pool to gather samples. However it would require the collection of many samples to get enough coverage to reconstruct the addressing function. We want to find out for every bit of the physical address if it influences the accessed DRAM component. Thus addresses that differ in only one bit would allow us to directly judge the influence of this one bit on the result. To generate such addresses we use the following mechanism. First, we take a random address from the allocated memory pool. The next address is generated by changing the least significant bit. The following addresses are generated by reversing the last bit change and then changing the next more significant bit. In other words, a shifted bitmask with a single one is xored with the initial address. Once we run into the next page frame or out of the boundaries of the address pool, we choose a new random address and start again with modifying the least significant bit.

The memory controller takes physical addresses as the input of the mapping function. Thus we need to gather physical addresses samples. Through the /proc/self/pagemap interface, we can translate the virtual addresses to physical addresses.

6.1.3 Performance Counters

For each physical address, we need to know which component is accessed. We use performance counters for each component to measure if they are accessed. Each channel has its own Performance Monitoring Unit (PMU) with its own counters. By selecting the right PMU we count the number of transfers on this channel. For each rank, there is a separate performance event with a separate umask for each bank or bank group. The event definitions can be found in the Intel uncore performance monitoring guide [50]. Each measurement checks one specific channel, rank, and bank. We cycle through all possible components until we found one where the counter value is equal or higher than the number of programmed DRAM accesses. The performance counters that we use require root access or the perf event paranoid flag to be set accordingly. The use of performance counters is a significant difference from a previous timing based approach [96]. The measurement is more reliable and practically eliminates attribution errors. A disadvantage of this method is that only CPUs which have the appropriate performance counters are supported.

6.1.4 Enforcing DRAM Accesses

For the measurement, it is required to repeatedly access a certain address, and every access must cause a load from DRAM that we can count. Registers and caches exist to avoid such redundant loads from DRAM. Thus we use the code in Figure 6.1 with cache line flush and fence instructions that enforce a load from DRAM with every access to the same address.

The performance counters we use count for the whole system, not just for our application. Thus co-running applications or transfers by the OS cause noise. We need to set the NUM_ACCESSES variable high enough so that those other accesses do not disturb our measurements. If the number of accesses is too high, it causes unnecessary delays. In our experience, on a system that is not running other applications than the default OS services, 2000 or more accesses allow accurate measurements. On a system that executes other tasks in the background, a higher value may be required.

```
volatile uint64_t *p = (volatile uint64_t *) addr;
for (unsigned int i = 0; i < NUM_ACCESS; i++)
{
  _mm_clflush((void*)p);
  _mm_lfence();
  *p;
  _mm_lfence();
}
```

Figure 6.1. The C code that enforces memory loads from DRAM when repeatedly accessing the same address.

6.1.5 Computing Addressing Functions

The list of samples that contain physical address and accessed component is not useful on its own. We need to extract an addressing function from those samples. We introduce a novel method to resolve the bits of the physical address that are used for addressing components. It reports exact results for used, unused, and unconfirmed bits, as well as other types of errors.

Constructing an Equation System

All previous work [39, 79, 96, 107] indicates that the mapping either uses a single bit or a xor combination of multiple bits of the physical address to calculate the component index. Because of the limitation to xor functions, this problem can be formulated as a boolean equation system consisting of two operations (*xor*, *and*). The idea for the construction of the equation system is as follows. The input is a list of samples as shown in Figure 6.2.

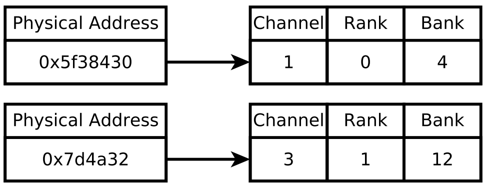


Figure 6.2. A list of physical addresses and the accessed component.

We split all the collected samples into a one-bit component address. E. g. If there are four memory channels, two bits are needed to address those four channels. We duplicate the list of physical addresses and build two lists of samples, one for the first bit of the channel address and one for the second bit of the channel address. Each of the bits of the physical address could be used for the calculation of the component address bit. Thus we add a switch (the boolean *and* operation) to every bit. This switch can turn the usage of this bit on or off. If there are multiple bits used, we know that they are combined using the xor operation. Thus we add a xor between every bit. The resulting structure is visualized in Figure 6.3.

The formalization of this concept as an equation system is shown in Equation 6.1.

$$\begin{aligned}
&x_{0,0} * b_0 \oplus x_{0,1} * b_1 \oplus \cdots \oplus x_{0,n} * b_n = c_0 \\
&x_{1,0} * b_0 \oplus x_{1,1} * b_1 \oplus \cdots \oplus x_{1,n} * b_n = c_1 \\
&\vdots \\
&x_{m,0} * b_0 \oplus x_{m,1} * b_1 \oplus \cdots \oplus x_{m,n} * b_n = c_m
\end{aligned} \tag{6.1}$$

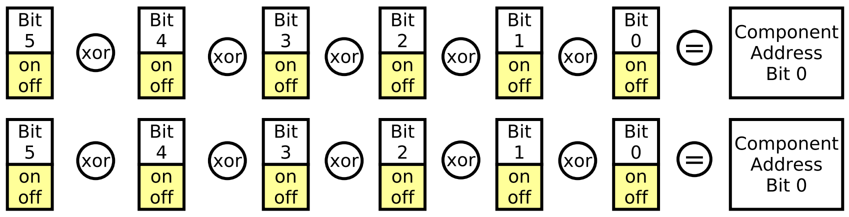


Figure 6.3. The list of samples converted into equations with bit switches.

In Equation 6.1 the $*$ symbol denotes the *and* operation and $\oplus$ symbol is the *xor* operation. There are n unknown parameters b that express if a bit is used or not. And there are m physical address samples taken with their individual address bits x. The c on the right-hand side represents the one-bit component address.

Solving the Equation System

The equation system can be solved in the F_2 space, where *xor* is an addition and *and* is a multiplication, with any equation system solver. We use Gaussian elimination. The usage of established linear equation system mathematics brings the advantage of a clear differentiation of the results.

The equations system either has a solution, is partially solvable, or it has no valid solution. If it is not solvable, there are contradicting equations. This can happen in case of wrong measurements or if a more complex address mapping, such as one with multiple regions, is not correctly considered. Theoretically, it could happen that wrong measurements lead to an equation system that is solvable and produces wrong results. However, this would require a systematic error in the measurement. For example, if a performance counter always reports accesses to a different component than the one specified in the measurement setup. Such an error is unlikely to occur, and we never observed such a case in our experiments.

If there is a solution or a partial solution, for every bit of the physical address there are three possible states. A bit can be used for calculating the index, a bit can be unused, or it is unknown if a bit is used. The unknown state happens if there is a partial solution with dependent equations, and there are no samples that cover this specific bit.

This accurate reporting is an advantage over the brute force solver by Pessl et al. [96] because it can identify wrong measurements and differentiate it from unknown bits caused by too few samples.

6.1.6 Region Based Mapping

In the case of asymmetric DIMM population or if the number of DIMMs in a channel is not a power of two, the hardware uses more complex region based address mapping. For example, a two-bit wide channel address, calculated using the xor combination of bits, targets four different channels. A space of three different channels can not be expressed. Thus a region based mapping is used in hardware. The regions are address ranges. For each region, a different addressing function can be used. The regions help to get a balanced distribution of requests over all of the three channels. The regions are defined in registers in the memory controller and set up during system startup. Those registers are mostly undocumented, but Hillenbrand [39] provides the locations and decoding for Intel Haswell and Broadwell systems.

The address sample collection works the same, no matter if regions are used or not. For the calculation of the addressing functions, additional steps are necessary. First, we read the region limit addresses from the registers. Then we group all captured addresses into their respective region. Finally, for each of the regions, the addressing functions can be computed as described in Section 6.1.5.

As already reported previously [39], on some systems, Linux is not able to access the PCI extended configuration space. The channel region definitions are within an address range that is always accessible. But for the ranks, the registers may not be accessible. We experienced this issue on an Intel Haswell system that is equipped with 6 ranks per channel. A workaround is to use a modified Linux kernel [39]. On our two Broadwell systems, complete configuration space was accessible.

Hillenbrand [39] only describes the registers of Broadwell and Haswell based systems. For Skylake and its successors, the register layout changed and is poorly documented. We cannot read the registers to find the regions for channels or banks. On those newer generation systems, our approach only works for a balanced power of two DIMM population.

6.2 Results

We reverse engineered the address mapping on four different systems and confirm that the obtained addressing functions are correct.

6.2.1 Hardware

Table 6.1 lists the basic facts of the server systems. In the following, the servers will be referenced by their name, which is in the leftmost column of Table 7.1.

Table 6.1. The hardware we used for testing the reverse engineering method.

Name	Architecture	CPUs	Board	DRAM Speed	DIMMs	Number of Channels	Number of Ranks
Arcturus	Broadwell	2x E5-2699v4	Supermicro X10DGQ	2400Mhz	Micron 36ASF4G72LZ-2G3B1	4	4
Comet	Haswell	2x E5-2699v3	Dell 0CNCJW	1867Mhz	SK Hynix HMA84GL7MMR4N-TF	3 and 4	6
Rigel	Skylake	2x Xeon 8176	Supermicro X11DPG-QT	2667Mhz	Samsung M386A8K40BM2	4	4
Spica	Broadwell	4x E7-8890v4	Supermicro X10QBL-4	1600Mhz	Samsung M386A8K40BM1	4	8

All of the systems are NUMA aware but do not use on-chip NUMA. This means that the OS sees the different processors with their own explicitly addressable memory but cannot see the different memory controllers within one processor. All of the systems are equipped with DDR4 RAM. DDR4 RAM always has four bank groups, each with four banks. Every system uses only a single type of DIMM. For Arcturus, Rigel, and Spica, the configuration is the same on all sockets. On Comet, the memory setup differs for socket 0 and socket 1. On socket 0, only three out of four channels are active due to a hardware defect. On socket 1, all of the four channels are active. Rigel is a Skylake system, which supports up to six memory channels. Our system is equipped with DIMMs on four of the six channels. The rightmost column ranks in Table 7.1 is the number of ranks per channel.

All of the experiments were executed on machines running Ubuntu 18.04, and the benchmarks were compiled with gcc 7.4. We configured our tool to use a 20GB address space that is allocated using 2MB pages. We execute 2000 accesses per test and capture a total of 400 address samples.

6.2.2 Addressing Functions

We use a 0 based numbering for address bits. I. e. the bit number 0 is the least significant bit of an address followed by bit 1 and so on. The bank addressing can either be interpreted as 16 different banks, which need 4 bits for addressing. Or it can be seen as four bank groups, each with four banks. Thus the bank group addressing bits are the same as two of the bank addressing bits. There are separate performance counters for the 16 banks and the four bank groups. The reverse engineering is done separately and we report the results for all 16 banks and the four bank groups individually.

Table 6.2 shows the addressing function of Arcturus. Channels and banks use xor hashing. The pattern of the mapping is similar to what is reported by Pessl et al. [96]. However, the individual bits used for addressing are different. It highlights the need to study the mapping for every system individually.

Because Comet is equipped with 6 ranks per channel, a region based mapping is used for the ranks. The rank configuration registers are not accessible using a standard Linux kernel. Thus we cannot resolve the rank addressing and subsequently cannot resolve the bank addressing. On socket 0, there is a region based mapping due to the use of 3 channels. It is shown in Table 6.4. In the first address region, we did not record any memory accesses. We suspect that it is a reserved hardware area that is not mapped to the DRAM. The second region uses interleaving between the two controllers. Within the first memory controller, the channels are interleaved. The second controller needs no further interleaving because only one channel is available. The third region only uses the first controller and interleaves it's two channels. The used bits are different from the second region. To the best of our knowledge, this is the first time a region based mapping was successfully reverse engineered.

Spica and Rigel do not use xor hashing. Instead, only single bits are used as shown in Table 6.5 and Table 6.6. With such a configuration, a performance decreasing imbalanced use of channels, ranks, or banks can easily occur if strided memory accesses happen to all fall into the same channel, rank, or bank.

If we equip Rigel with enough DIMMs for all six channels, a region based mapping will be used. Because we do not know the configuration registers for this architecture, reverse engineering is not possible.

Table 6.2. DRAM address mapping of Arcturus.

Component	Index Bit	Physical Address Bits
Channel	0	$8 \oplus 12 \oplus 14 \oplus 16 \oplus 18 \oplus 20 \oplus 22 \oplus 24 \oplus 26$
	1	$7 \oplus 17$
Rank	0	15
	1	16
Bank	0	$6 \oplus 24$
	1	$21 \oplus 25$
	2	$22 \oplus 26$
	3	$23 \oplus 27$
Bank Group	0	$6 \oplus 24$
	1	$21 \oplus 25$

Table 6.3. DRAM address mapping of Comet socket 1.

Component	Index Bit	Physical Address Bits
Channel	0	$8 \oplus 12 \oplus 14 \oplus 16 \oplus 18 \oplus 20 \oplus 22 \oplus 24 \oplus 26$
	1	$7 \oplus 17$

Table 6.4. DRAM channel address mapping of Comet socket 0

Address Region	Component	Physical Address Bits
0 to 1984M	-	-
1094M to 198592M	Controllers	$7 \oplus 17$
	Channels in Controller 0	$8 \oplus 12 \oplus 14 \oplus 16 \oplus 18 \oplus 20 \oplus 22 \oplus 24 \oplus 26$
198592M to 296896M	Channels in Controller 0	$7 \oplus 12 \oplus 14 \oplus 16 \oplus 18 \oplus 20$

Table 6.5. DRAM address mapping of Rigel.

Component	Index Bit	Physical Address Bits
Channel	0	8
	1	9
Rank	0	15
	1	16
Bank	0	6
	1	21
	2	22
	3	23
Bank Group	0	6
	1	21

Table 6.6. DRAM address mapping of Spica.

Component	Index Bit	Physical Address Bits
Channel	0	6
	1	7
Rank	0	8
	1	9
	2	10
Bank	0	11
	1	12
	2	13
	3	14
Bank Group	0	11
	1	12

6.2.3 Speed of Reverse Engineering

In addition to the reliability of the measurements, timing based approaches also have the disadvantage of long processing time. We compared the time required for reverse engineering of our approach and the timing based approach of Pessl et al. [96], even though it did not report the correct result. We did the experiment on Spica. It has a large DRAM size of 512GB per socket, and it has 512 addressable sets (16 banks $\times$ 8 ranks $\times$ 4 channels). Our tool finds the correct addressing for all sets in ten out of ten tests in an average time of 1:04 minutes. In contrast, the timing based approach needs over 51 hours for the complete reverse engineering process. We assume the optimal case when the mapping is found in the first try, which often does not work due to inaccuracies in the timing measurement.

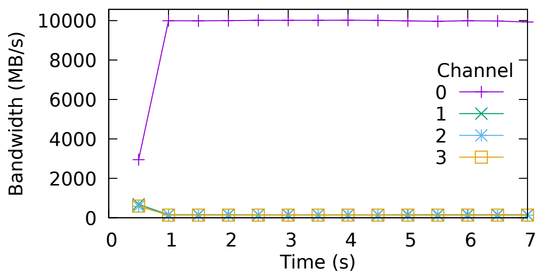
6.3 Usage of Addressing Functions

To confirm that the recovered address mappings are correct, we implement micro benchmarks that reproduce known performance effects of bank and channel usage.

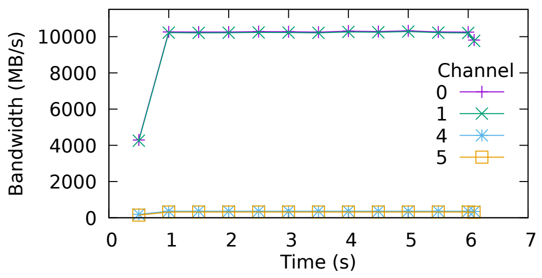
Based on addressing functions shown in Section 6.2.2, we implement a benchmark that can access specific memory components. The benchmark first allocates a large array. Then it calculates the array indexes that are on the specified channels, ranks, and banks. Finally, it accesses the calculated array indexes in a parallel for loop. The number of data accesses stays constant, regardless of the configured channels, ranks, banks, and threads.

6.3.1 Channels

Figure 6.4a shows the measured bandwidth on the four different channels of Spica. In this experiment, the benchmark accesses only one memory channel. This can be clearly seen in the diagram. We measure a bandwidth of about 10GB/s on one of the channels but almost no activity on the other channels. Figure 6.4b shows the bandwidth on Comet with two out of four channels in use. Figure 6.5 shows the speedup over the sequential version when only one, two, three, or all four memory channels are used. As expected, the speedup is limited by the number of available channels. This experiment was executed on Spica. These experiments demonstrate that our determined addressing functions for the channels are correct.



(a) Measured on Spica when memory accesses are limited to one specific channel.



(b) Measured on Comet when memory accesses are limited to two channels.

Figure 6.4. Time resolved bandwidth graph.

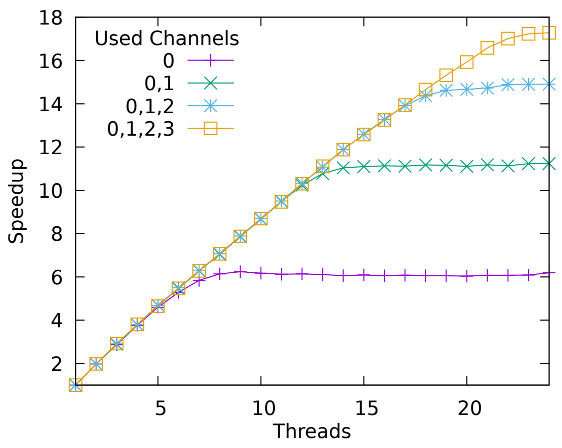
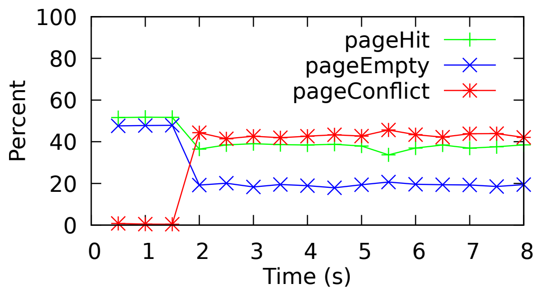


Figure 6.5. Parallel speedup on Spica when the usage of memory channels is restricted.

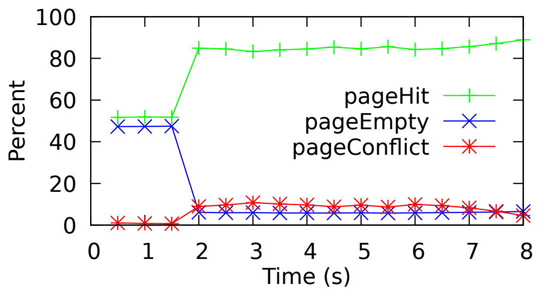
6.3.2 Banks

It is known that co-running applications or multi-threaded programs, where concurrent threads access different address regions, interfere with each other, and cause increased row buffer conflicts [14,40,86]. We implemented a micro-benchmark that reproduces this phenomenon and an optimized version that uses a fixed thread-to-bank assignment. The benchmark reads an array using 16 threads. We limit the access on one channel and one rank so that there are 16 banks available to use. The original version simply accesses the array indexes in ascending order using a loop that is parallelized with OpenMP. Thus each thread accesses different array locations. In the optimized version, we use the same parallel for loop, but each thread accesses only a specific bank. E. g. thread 1 only accesses data stored on bank 1, while at the same time thread 2 only accesses bank 2. We measure the row buffer access status (hit, empty, conflict) as described in the Intel documentation [48].

Figure 6.6a shows the page hit, page empty, and page conflict ratios measured over time of the original version. We see that, in the original version, after the initialization phase, the page conflicts increase and the page hits decrease. Over 40% of the row accesses result in a page conflict. In contrast, in the optimized version, we can eliminate most of the conflicts and get a high page hit ratio by mapping each thread to a designated bank, as shown in Figure 6.6b.



(a) Using a standard multi-threaded array access.



(b) Using a fixed thread-to-bank assignment.

Figure 6.6. Row buffer hit, empty, and conflict ratio measured over time on Arcturus.

Chapter 7

Evaluation

This evaluation covers many different aspects. First, we evaluate the false sharing detection method. Second, we present the results of the main memory bandwidth contention detection evaluation. It also includes the evaluation of the NUMA imbalance, channel imbalance, and row buffer hit rate. Then we discuss the profiling overhead of our approach and the processing time of the profiling data. Finally, we present case studies of PARSEC benchmarks and machine learning applications.

7.1 Experiment Environment

The hardware and OS configuration is common for all of the experimental evaluations. All of the experiments were executed on machines running Ubuntu 18.04 and the benchmarks were compiled with gcc 7.4.

7.1.1 Hardware

We used four different systems for the evaluation. The characteristics are listed in Table 7.1. In the following experiments, the processors will be referenced by their name that is printed in the first column of Table 7.1. Hyper-threading is enabled on all machines. The machine called Rigel has a lower L3 cache size than the other systems. But this L3 cache is exclusive. Data that is stored in the L2 cache is not stored in the L3 cache. The L3 cache of the other systems in inclusive, which means that data that is stored in the L2 cache is also stored in the L3 cache.

Table 7.1. The hardware used for the evaluation of DRAM contention detection, false sharing detection, profiling overhead, and in the case studies.

Name	Architecture	CPUs	Total Phys. Cores	L3 Cache	DRAM Speed
Arcturus	Broadwell	2x E5-2699v4@2.2Ghz	44	55MB	2400Mhz
Comet	Haswell	2x E5-2699v3@2.3Ghz	36	45MB	1847Mhz
Rigel	Skylake	2x Xeon 8176@2.1Ghz	56	38.5MB	2666Mhz
Spica	Broadwell	2x E7-8890v4@2.2Ghz	48	60MB	1600Mhz

Memory Setup of Spica

The mcmtr [49] register is set to 0x00007d0d which means that closed-page management policy is used. Also xor mapping of bank addresses is disabled. All four nodes are equipped with the same amount and type of memory. Spica has four memory channels. The equipped DIMMs are Samsung M386A8K40BM1-CPB [105,106]. This kind of DIMM has a total capacity (not including EEC checksum) of 64GB. A DIMM has four ranks with a total of 36 DRAM chips. For each rank there are 9 DRAM Chips. One is used for the ECC checksum and 8 are used for data. Each rank has a capacity of 16GB. Each chip has a capacity of 2GB and a data bus width of 8 bit. There are 16 banks in each chip. The banks are arranged in 4 bank groups each with 4 banks. There are 8 chips for each rank. There are 131072 (2^{17}) rows and 1024 columns. A row spreading across all eight chips has a size of 8192 bytes. A row in a single chip has a size of 1024 bytes. There are 1024 columns in each row. Every column has a capacity of 1 byte or 8 bits.

Memory Setup of Arcturus

The mcmtr register is set to 0x00314f0c. This means open-page policy and enabled xor bank mapping. It uses Micron 36ASF4G72LZ-2G3B1 Dual-rank DIMMs with a capacity of 32GB. A DIMM has two ranks with a total of 36 DRAM chips. For each rank there are 18 DRAM Chips. Two are used for the ECC checksum and 16 are used for data. Each rank has a capacity of 16GB. Each chip has a capacity of 1GB and a data bus width of 4 bit. There are 16 chips for each rank. There are 16 banks in each chip. The banks are arranged in 4 bank groups each with 4 banks. There are 131072 (2^{17}) rows and 1024 columns. A row spreading across all eight chips has a size of 8192 bytes. A row in a single chip has a size of 1024 bytes. There are 1024 columns in each row. Every column has a capacity of 4 bits.

Memory Setup of Comet

On Comet the mcmtr register is set to 0x00014f04. This means open-page management policy and a replacement of ChnAdd[6] with SysAdd[6]. The CPU of Comet supports four channels. However, socket 0 is only equipped with DIMMs that populate three channels. Socket 1 is equipped with enough DIMMs for all four channels. We can only recover the channel address mapping for socket 1. Because Comet has 6 ranks per channel we can not apply our reverse engineering approach only for the channel mapping but not for the ranks and banks.

Memory Setup of Rigel

Because Rigel has 6 memory channels we can not apply our reverse engineering tool. We can not include Rigel in experiments that need the address mapping.

7.1.2 Benchmarks

For each of the evaluated performance problem detection method we use specific micro-benchmarks. In addition we also use a set of common benchmarks. Those represent more realistic use cases. We apply every detection method to those common benchmarks. These common benchmarks are:

- Small kernels. These contain the Stream [82] benchmarks split into their four components; Add, copy, scale, and triad. We use a data size of around 600MB per array. The operations in the benchmark are repeated 30 times. The small kernels also contain a matrix multiplication benchmark using double precision floating point numbers (DGEMM). This benchmark comes from OpenBLAS [133]. Each matrix has 6000 rows and 6000 columns. We refere to this benchmark as mMat.

- The PARSEC [11] benchmarks. Those represent a wide spectrum of realistic applications. We use the pthreads version of the benchmarks except for Freqmine which comes only with an OpenMp version.

7.2 False Sharing Detection

With this evaluation, we want to verify if our approach can reliably detect false sharing, if it raises false positives, and how the sampling period affects the detection results.

To achieve this, we need a set of benchmarks and the ground truth if those benchmarks suffer from false sharing or not. We use two types of benchmarks. First, simple micro-benchmarks where the occurrence of false sharing is obvious and visible in the source code. Second, realistic applications that are too complex to manually check for the occurrence of false sharing. All the benchmarks have been analyzed in previous studies. We know if benchmarks suffer from false sharing or not from the previous publications. If our approach points out false sharing, we manually check for the occurrence of false sharing to verify whether the reported instance is correct.

We execute all benchmarks multiple times with different sample rates and record in how many cases false sharing is correctly identified. This way, we can examine the dependence of detection results on the sampling rate. Each benchmark configuration was executed 10 times.

7.2.1 Benchmarks

We used benchmarks from PARSEC [11], Phoenix [60] and artificial benchmarks [57]. The benchmarks were selected because they have been analyzed in several previous studies, and we can compare our detection accuracy to the previous approaches. They include micro-benchmarks and realistic applications.

Artificial Benchmarks

A part of the evaluation is done with artificial benchmarks that produce false sharing. They are small micro-benchmarks that serve no other purpose then deliberately creating false sharing. For most of the benchmarks, we rely on the source code that is published along with Huron [57]. All benchmarks are designed to run with 4 threads.

Boost Spinlock is a spinlock implementation taken from the boost C++ library with a specially designed program that triggers false sharing in this spinlock implementation. There is a pool of locks. Each lock struct consists of several fields with primitive data types. All lock structs of the pool are allocated in a contiguous array. Thus, multiple structs share one cache line. There is a similar array with pthread mutexes. The benchmark is implemented so that each thread accesses only one of the structs and only one of the mutexes. This leads to false sharing in the spinlock. There is no false sharing between mutexes because there is enough padding. There is no true sharing in the mutexes because each thread only accesses its own mutex.

Lockless is an artificial benchmark that serves no purpose except to create false sharing. Multiple threads update an array of integers. Each thread accesses an exclusive range of integers. Thus, there is only false sharing and no true sharing.

Locked is the same as lockless but there are mutex lock and unlock operations around the write access. There is an array of mutexes, with one mutex for each thread. There is no true sharing in the mutexes.

Ref Count is adapted from Java's reference counting implementation. True sharing occurs in a statically allocated mutex and a statically allocated shared variable. Because they are statically allocated PerfMemPlus can not detect true sharing. The main data structure consists of a small array of integers with one integer for each thread. There is also a large 2d array that contains pointers. Each pointer value is changed once. This step is protected by a mutex. In the second step, each thread repeatedly updates its own value of the integer array. This step is not protected by the mutex.

True Sharing is an artificial micro-benchmark to generate true sharing. It is not provided by Khan et al. [57]. It is our own implementation. There is an array of integers. Each thread updates each value of the array making every individual integer a truly shared object. The code of our implementation is shown in Figure 7.1.

```
struct myInt
{
  int value;
};
auto* array = new myInt[16];
std::mutex m;
size_t repeat = 20*1000*1000;
#pragma omp parallel num_threads(4)
for (size_t r = 0; r < repeat; r++)
{
  for (size_t i = 0; i < 16; i++)
  {
    m.lock();
    array[i].value += 1;
    m.unlock();
  }
}
```

Figure 7.1. The code of the true sharing benchmark. Values in an array are modified by multiple threads.

Phoenix Benchmarks

Two of the benchmarks from Phoenix [60] are known to suffer from false sharing. They are small benchmarks, but they serve a real purpose and are not just micro-benchmarks. We run them using 4 threads. The other benchmarks form Pheonix, that do not suffer from false sharing are not included in our evaluation.

Histogram calculates a histogram of a picture. The whole picture is loaded into memory and split into regions processed by each thread. If and how much false sharing occurs depends on the input of the program. Because false sharing occurs only in one of the three color channels' histogram. The test is run with two different inputs. The default input where false sharing seldom occurs and a specially crafted input (HistogramFs) where a lot of false sharing occurs.

Linear Regression calculates a linear regression on a set of input points. It uses the same array of structs pattern as the other benchmarks. The struct is 60 Bytes large. Within this struct, there are five 64 bit integers that are updated by each thread.

PARSEC Benchmarks

The PARSEC benchmarks are larger applications and one of them has known false sharing. We execute all benchmarks with the native input set and 4 threads. We use the pthreads version of the benchmarks. Because freqmine does not have a pthreads version we use the OpenMP version. We measure the execution time using the region of interest (ROI) as it is defined in PARSEC.

Some of the PARSEC benchmarks issue many memory allocations and de-allocations. This can cause a long delay in the post-processing of the profiling data. Thus, we set a minimum allocation size. Only larger allocations will be recorded. The limits are shown in Table 7.2. We applied a simple heuristic to find the limit. We start with a 10 Bytes limit. If the resulting allocation data has a size larger than 200 MB, we increase the limit to 1 KB. If it is still larger than 200 MB, we increase the size to 100 KB. At 100 KB, all recorded allocation data was below 200 MB.

Table 7.2. The minimum allocation size setting for profiling some of the PARSEC benchmarks. All other benchmarks are profiled with a minimum allocation size of 10 Bytes.

Benchmark	**Minimum Allocation Size**
Bodytrack	1 KB
Dedup	100 KB
Swaptions	100 KB
Vips	100 KB

The minimum allocation size does not influence the false sharing detection accuracy but influences the overhead. The false sharing detection can still be executed even if an object is not tracked. All addresses, for which no object can be attributed, are attributed to a common anonymous object. The false sharing detection is then executed using this anonymous object. Thus false sharing can be detected, even if the object is anonymous. But in the results, the actual object can not be resolved. In case there is false sharing, it will be reported in an anonymous object. If that is the case, it is recommended to lower the minimum allocation size limit to find the true object that suffers from false sharing.

Streamcluster is an application that solves the online clustering problem. More details about this benchmark are available in [11]. It is parallelized using pthreads. Previous work states that the benchmark suffers from false sharing [55, 67]. The object work_mem suffers from false sharing. It is a continuous array with an element size smaller than the cache line size. Each element in the array belongs to one thread and is exclusively accessed by one thread. There is padding implemented, but it is assuming a cache line size of 32 bytes. But the real cache line size is 64 bytes so that two elements still end up in the same cache line.

Other PARSEC benchmarks are not known to suffer from false sharing. Thus we do not further introduce them in detail in this section. In previous studies, it has been found that Bodytrack suffers from true sharing [74] because there is a producer-consumer pattern [94]. It is important not to confuse this pattern with false sharing. Other benchmarks also suffer from true sharing. For example, true sharing occurs in mutexes in many benchmarks.

We encountered bugs in raytrace and x264 that originate in the memory management. After the region of interest (ROI) of the benchmarks is completed the applications crash. This also terminates the allocation tracker and no objects can be resolved. This has the same effect as setting a high value for the min allocation size. False sharing detection can still be executed, despite the bug in the programs.

Performance Implications of False Sharing

Figure 7.2 shows the speedup that can be obtained when fixing false sharing in the benchmarks. In most of the benchmarks, false sharing is a severe performance problem. But in freqmine and streamcluster, there is only a small speedup of at most 6.5%. The significance of false sharing has also been confirmed in previous studies [57, 68].

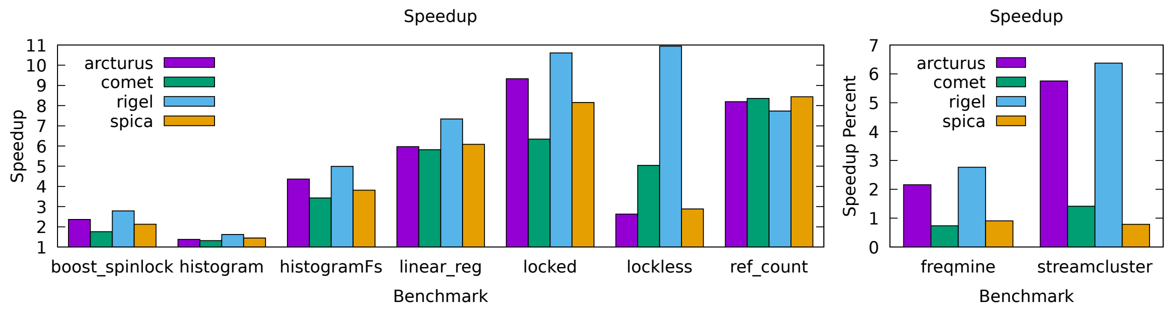


Figure 7.2. The speedup when fixing false sharing in various benchmarks. The speedup of the PARSEC benchmarks on the right side is much lower than that of the other benchmarks. The speedup scale on the left is the relative speedup compared to the initial version. The speedup scale on the right shows how many percent the execution time improved.

The coefficient of variation is on average 3.18% and at most 3.91%, for the PARSEC benchmarks. It is on average 56.18% and up to 59.17%, for the artificial false sharing benchmarks and Phoenix benchmarks. Especially the original versions, that are affected by false sharing have high variations of the execution time between iterations.

7.2.2 False Sharing Detection Accuracy

We conducted 13200 individual experiments. Table 7.3 lists the overall results of the false sharing detection. False sharing has been detected in 66% of all cases. This result does not consider the sampling period. With an appropriate sampling period, the detection results are much better, as shown in the latter part of this section. No false positives were reported. All 9240 cases without false sharing were correctly classified. The cases of true sharing were also correctly classified and did not raise false positives.

Table 7.3. The overall detection accuracy of false sharing. This evaluation does not consider the sampling period. There are no false positives.

		Detection	
		False Sharing	No False Sharing
Actual	False Sharing	1356	2604
	No False Sharing	0	9240

The detailed results show that the detection accuracy worsens with higher sampling periods in all of the experiments. Figure 7.3 shows the results combined for all systems and all benchmarks. Up to a sampling rate of 8000, the detection rate stays stable on high levels of over 90%. At a sampling period of 16,000 or higher, the detection rate rapidly decreases.

Out of all benchmarks, ref count (Figure 7.7) shows the worst detection accuracy. A manual analysis of the captured samples reveals that in many cases there are no HITM accesses captured, even with low sampling periods, such as one out of 500 samples. If the false sharing detection is executed on all cache lines, not just on those with the HITM flag set, then false sharing is detected. The benchmarks

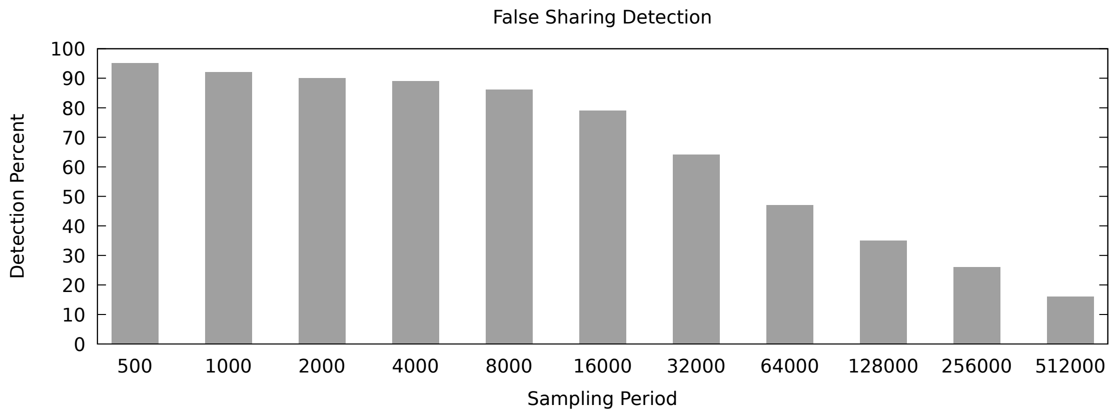


Figure 7.3. The overall accuracy of the false sharing detection depends on the sapling period.

histogram with false sharing input and linear regression show perfect detection results up to a sampling period of 256000. It can be seen in Figure 7.9 and Figure 7.10 respectively. As expected, detection results are worse with the original input as shown in Figure 7.8. In freqmine (Figure 7.11), inter-object false sharing is reliably detected up to a sampling period of 64000. Freqmine is the only benchmark that suffers from inter-object false sharing. In all other benchmarks, the type of false sharing is intra-object. In streamcluster (Figure 7.12), false sharing is found reliably up until a sampling period of 8000. In the other PARSEC benchmarks, no case of false sharing was found. Overall, a sampling period of 8000 is enough to detect most cases of false sharing. The detection results differ only slightly between the four systems. It indicates that the method works the same on all tested architectures.

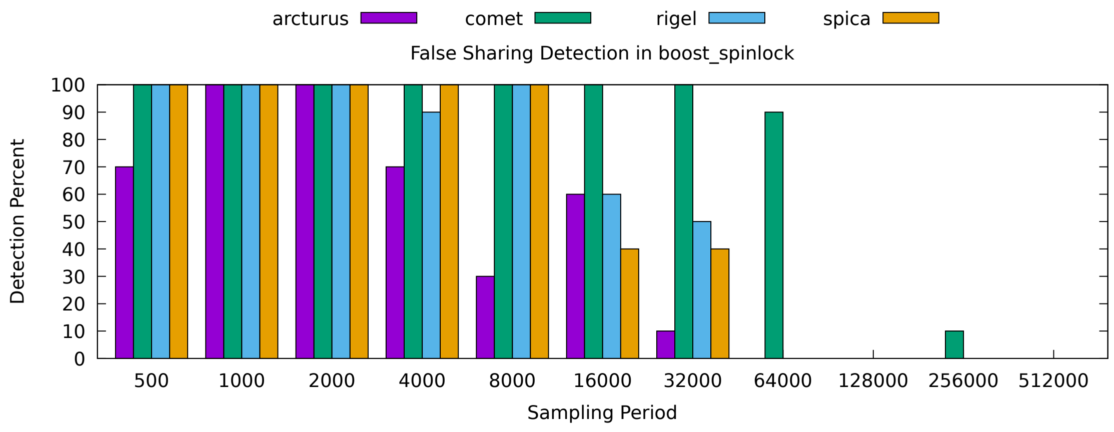


Figure 7.4. The false sharing detection results in the boost spinlock benchmark.

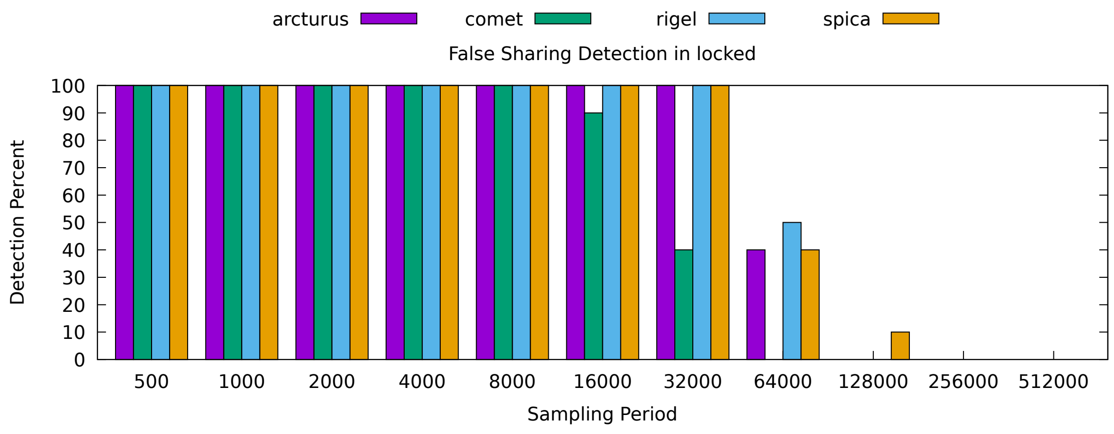


Figure 7.5. The false sharing detection results in the locked benchmark.

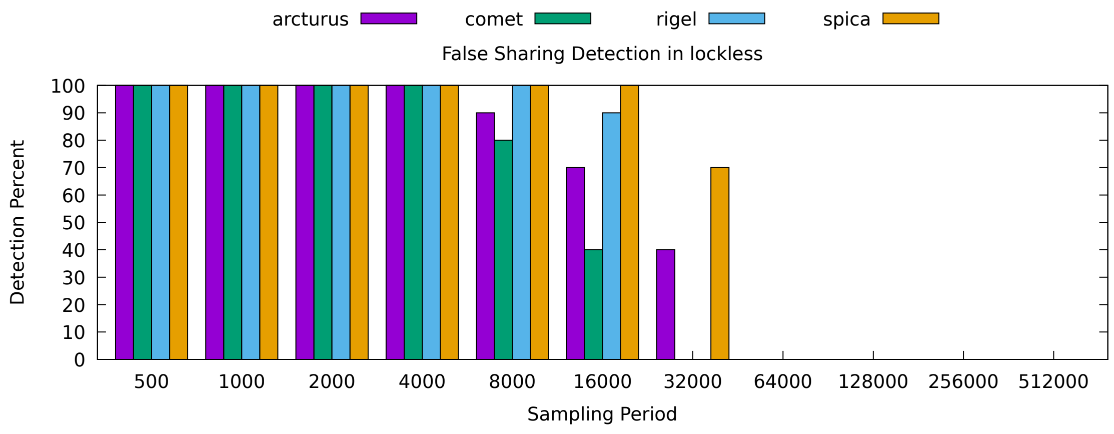


Figure 7.6. The false sharing detection results in the lockless benchmark.

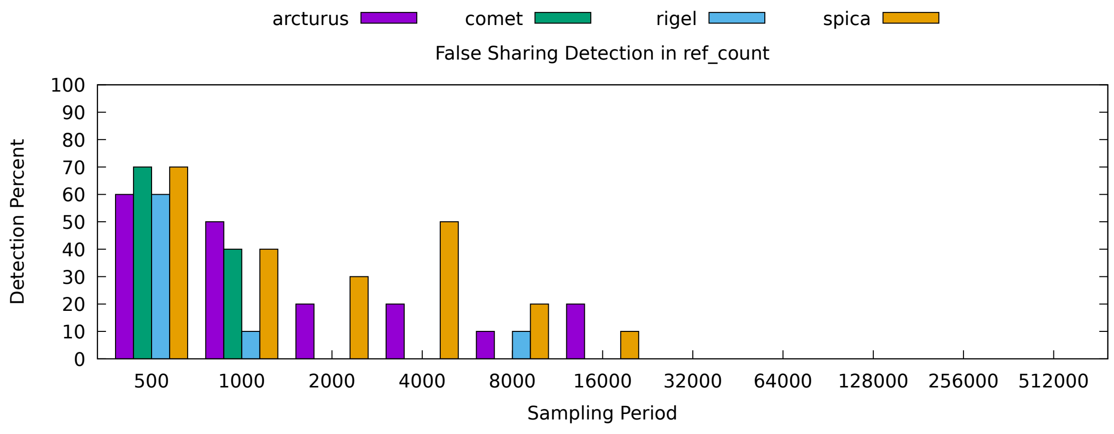


Figure 7.7. The false sharing detection results in the ref count benchmark.

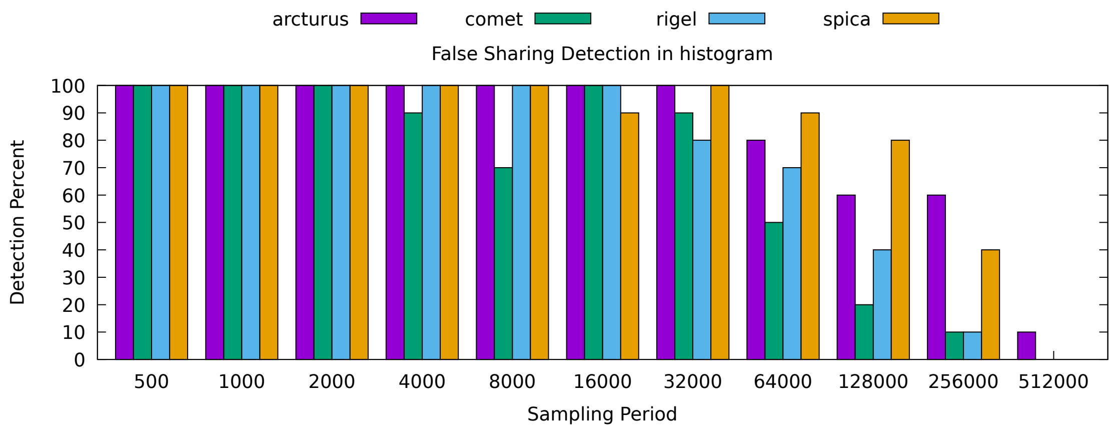


Figure 7.8. The false sharing detection results in the histogram benchmark with the default input.

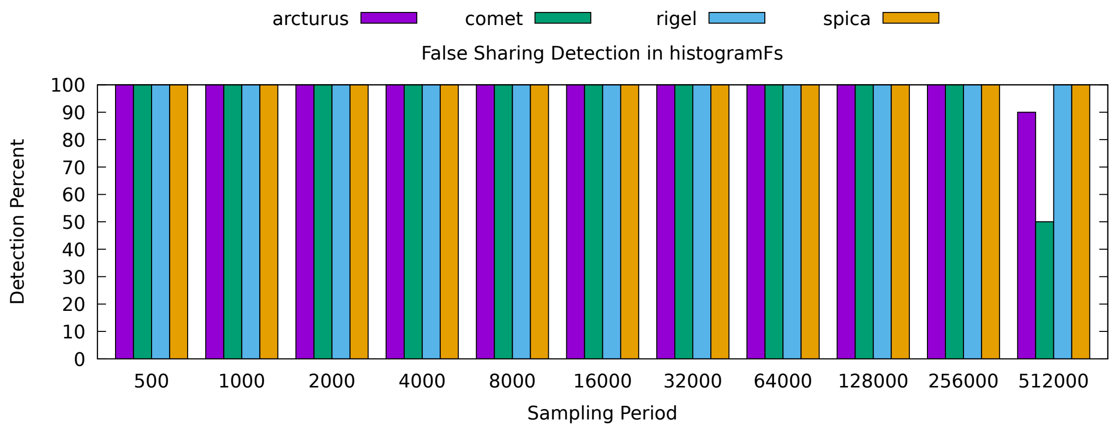


Figure 7.9. The false sharing detection results in the histogram benchmark with the input that is designed to cause false sharing.

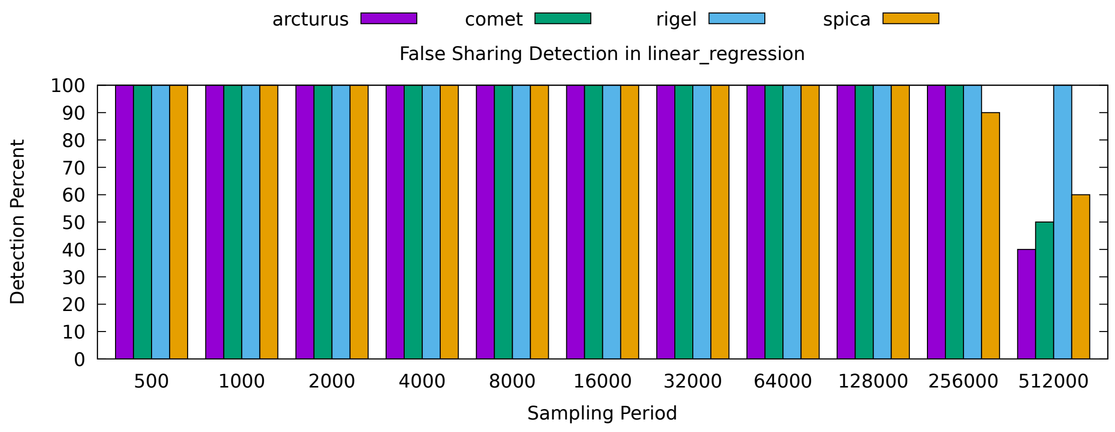


Figure 7.10. The false sharing detection results in the linear regression bechmark.

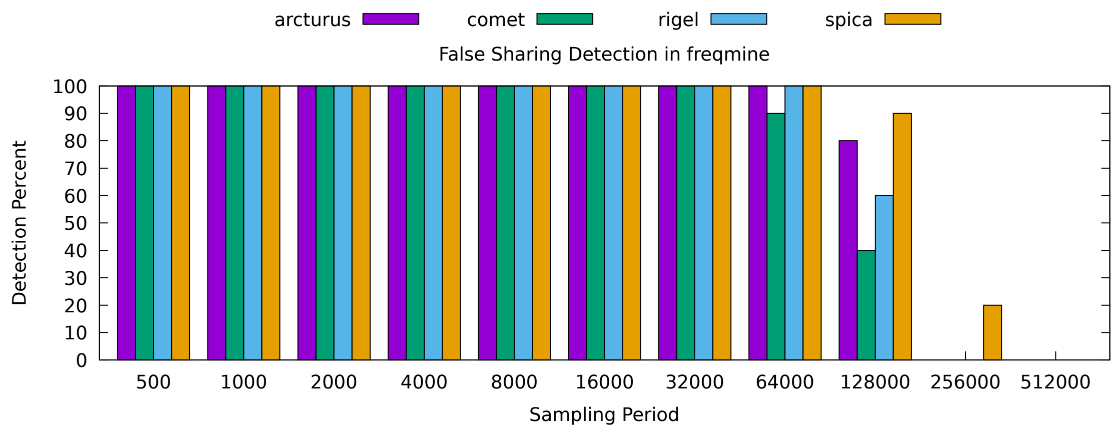


Figure 7.11. The false sharing detection results in the freqmine benchmark.

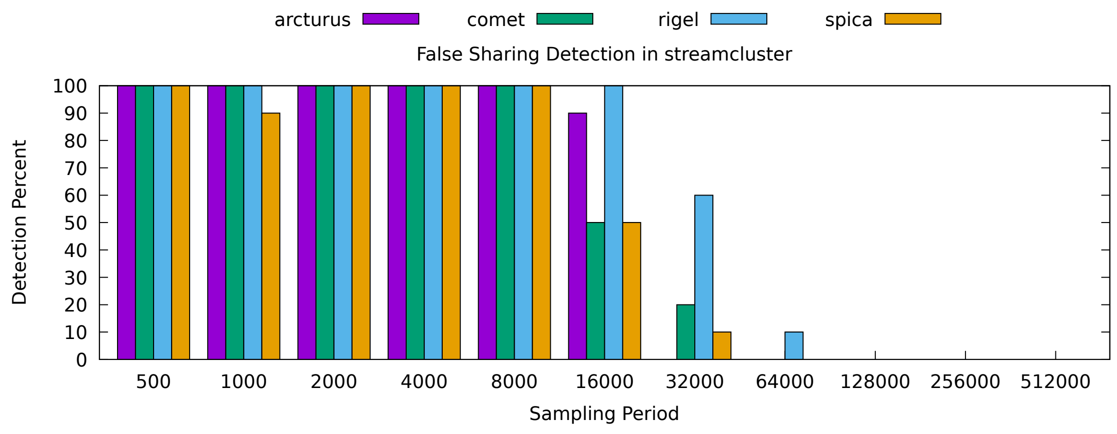


Figure 7.12. The false sharing detection results in the streamcluster benchmark.

7.2.3 Comparison with Existing Methods

The benchmarks were analyzed by several different tools in previously published papers. Based on those publications, we compare our results with existing approaches. Table 7.4 summarizes the detection results of our approach and previous approaches. PerfMemPlus detects all previously known cases of false sharing. It also detects false sharing in freqmine. This case has not been found by any of the other tools. However, it could be that due to a different memory allocator, false sharing did not occur in the previously published experiments. We could not verify this based on the results given in the papers. One may also argue that false sharing in streamcluster and freqmine only results in a small performance penalty, and thus it is tolerable if tools do not detect it. We, as well as most of the previous papers, regard the false sharing in streamcluster as a valid case. In freqmine, the performance impact of false sharing is slightly lower. Our approach did not raise false positives. But some of the previously published tools did raise false positives. In summary, the detection results of Feather and Huron are the best out of the previous tools. PerfMemPlus has the same detection accuracy but additionally finds false sharing in freqmine.

Table 7.4. The false sharing detection results of prior work and PerfMemPlus. TN = True Negative, TP = True Positive, FN = False Negative, FP = False Positive. Wrong classifications are marked in red. An empty cell indicates that the benchmark was not analyzed in the publication.

Benchmark	**Predator** [68]	**Feather** [13]	**Huron** [57]	**ML** [55]	**TMI** [17]	**Sheriff** [66]	**PerfMemPlus**
Blackscholes	TN	TN	TN	TN	TN	TN	TN
Bodytrack	TN	TN	TN	TN	TN		TN
Canneal	TN	TN	TN	TN	TN	FP	TN
Dedup	TN	TN	TN	TN	TN	TN	TN
Facesim	TN	TN	TN	TN	TN		TN
Ferret	TN	TN	TN	TN	TN	TN	TN
Fluidanimate	TN	TN	TN	TN	TN	FP	TN
Freqmine	FN	FN	FN	FN	FN	FN	TP
Raytrace	TN	TN	TN	TN	TN		TN
Streamcluster	TP	TP	FN	TP	FN	TP	TP
Swaptions	TN	TN	TN	TN	TN		TN
Vip	TN	TN	TN	TN	TN		TN
x264	TN	TN	TN	TN	TN		TN
Boost spinlock			TP		TP	FN	TP
Histogram	FN	TP	TP	FN	TP	FN	TP
HistogramFs			TP		TP		TP
Linear regression	TP	TP	TP	TP	TP	TP	TP
Locked			TP		TP	FN	TP
Lockless			TP		TP	TP	TP
Ref count			TP		TP	TP	TP

The different approaches of false sharing detection have requirements for their usage. Huron [57] and Predator [68] need an LLVM compiler. Sheriff [66] can not handle inline assembly or atomic instructions [74]. Feather [13] requires a custom Linux kernel to reduce the overhead. It needs support for instructions sampling and debug registers. The machine learning approach by Jayasena [55] cannot point out objects and code locations with false sharing and requires hardware performance monitoring. TMI and PerfMemPlus only require support for hardware instruction sampling.

7.3 DRAM Bandwidth Contention Detection

In this section, we evaluate the main memory bandwidth contention detection feature of PerfMemPlus. The hardware is the same as the one used for the evaluation of the false sharing detection. It is described in Table 7.1. First, we evaluate the DRAM contention detection with micro-benchmarks, that allow fine control over the amount of contention and thus a detailed analysis. Second, we test our approach and with benchmarks from PARSEC to verify if it also works with realistic applications.

7.3.1 Micro-Benchmarks

Because the artificial benchmarks have a known behavior and the amount of DRAM contention is known, we can do a detailed evaluation of our DRAM contention detection approach. We show that the DRAM latency is a superior metric compared to DRAM bandwidth. And that the detection of DRAM contention based on a fixed hardware-specific threshold latency detects all cases of severe contention.

Benchmarks

We evaluate the DRAM contention detection feature using artificial micro-benchmarks to check the sensitivity of the detection. The idea is to use benchmarks, which are known to suffer from an adjustable degree of DRAM contention. We use benchmarks that Xu et al. [128] introduced and made available online [127]. They consist of three individual benchmarks. All of them use three different vectors and execute operations on them in parallel. Xu et al. designed them to create different memory access patterns. Because of those different memory access patterns, they are also suitable for us. The three benchmarks are:

- countv: It counts how often a certain value appears in the vectors.
- sumv: It sums up the values in each individual vector.
- dotv: It calculates the dot product of two vectors and stores it in the third vector.

The benchmarks are parallelized with OpenMP parallel for loops. These operations are repeated many times to increase the execution time and amount of memory accesses. There are two configuration options for those benchmarks. First, the number of threads. Second, the array size. We chose two different array sizes. First, an array size of 1,000,000 that results in a total data size of 23 MB. Second, an array size of 10,000,000 that results in a total data size of 229 MB. The small size fits into the L3 cache of all the systems listed in Table 7.1. The large array size fits into none of the L3 caches. By comparing the performance of both versions, we know the performance influence of the DRAM accesses.

The second varied parameter is the number of threads. These benchmarks are embarrassingly parallel, but memory-intensive applications. We expect near-linear scaling with the small array size but limited scalability with the large array size due to the DRAM bandwidth limitation. Our experimental results, which are shown in Figure 7.13, confirm this.

Because we want to study the influence of local DRAM contention, we execute the benchmarks only on one of the two available processors per system and enforce memory allocation on the local DRAM. We study remote DRAM contention in separate experiments. All benchmarks were executed 10 times. The provided performance results are the average of those 10 repetitions. For detection results, we show in how many of the 10 cases DRAM contention was found. The coefficient of variation of the execution time is on average 8.05% and at most 13.89%.

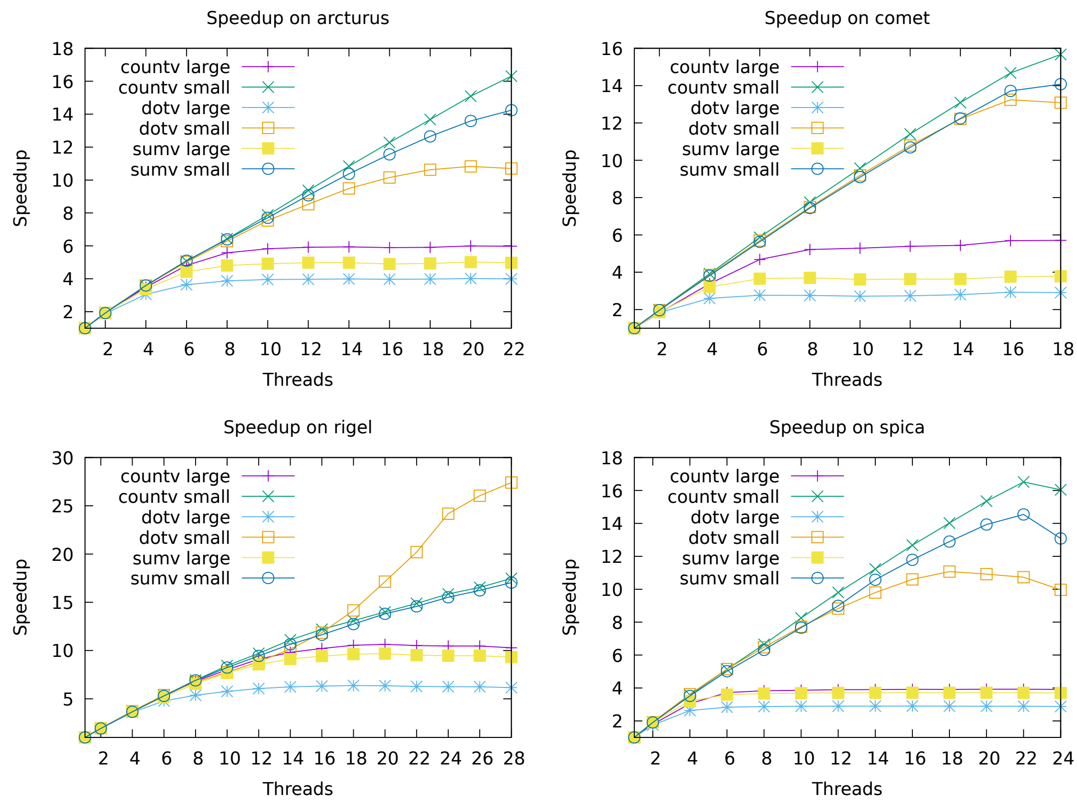


Figure 7.13. The speedup of the micro-benchmarks used for DRAM contention detection evaluation. Benchmarks with a small data size show better parallel scaling than benchmarks with large data size.

Speedup Loss Metric

First, we need to establish a metric for the severity of DRAM contention in the benchmarks. This metric is the speedup loss. It is defined as the speedup of the large array version divided by the speedup of the small array version. It expresses how many times more speedup could be obtained if there would be no DRAM limitation. For example, The speedup loss of sumv on Comet at 8 threads is 2. At this configuration, the large array version reaches a speedup over the sequential version of around 4. The small array version reaches a speedup of 8 over the sequential version. Figure 7.14 shows the speedup loss over the number of threads. There is a big range of losses between different benchmarks and systems. This allows us to evaluate the DRAM contention detection in a wide range of scenarios. In general, the speedup loss increases with the number of threads and the speedup loss of the dotv benchmark is the highest. On Comet, there is a large difference in the speedup loss of the benchmarks. In contrast, on Arcturus and Spica, the speedup loss of all benchmarks differs only slightly. On Rigel, countv and sumv show a similar speedup loss, but the speedup loss of countv is much greater.

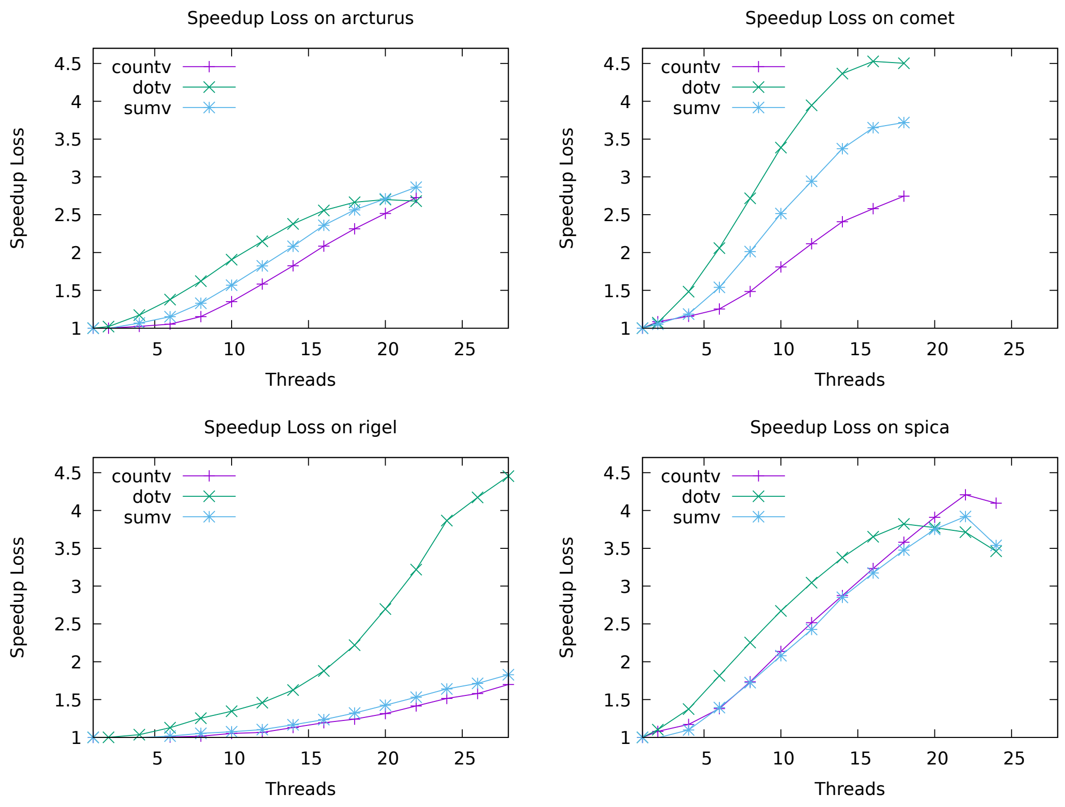


Figure 7.14. The speedup loss of the artificial benchmarks expresses the severity of DRAM contention. It rises with the number of threads and differs between benchmarks and systems.

Summary of Contention Detection

Figure 7.15 shows a summary of DRAM contention detection in all benchmarks and on all machines. On the horizontal axis are intervals of speedup loss. It starts with a speedup loss of less than 1.25. The following intervals are groups of speedup loss with a range of 0.25. The vertical axis shows in how many cases, out of all experiments that fall into the group, a DRAM contention was detected. It is a summery over all systems and all benchmarks. From this figure, we can see that cases with a small speedup loss are only occasionally detected. When the speedup loss gets higher, a DRAM contention is always detected. For example, if the speedup loss is at least 2, and the sampling period is 500 or 1000, then in over 75% of the cases the DRAM contention is detected. At a speedup loss of 2.5 or higher, it is detected in over 95% of the cases.

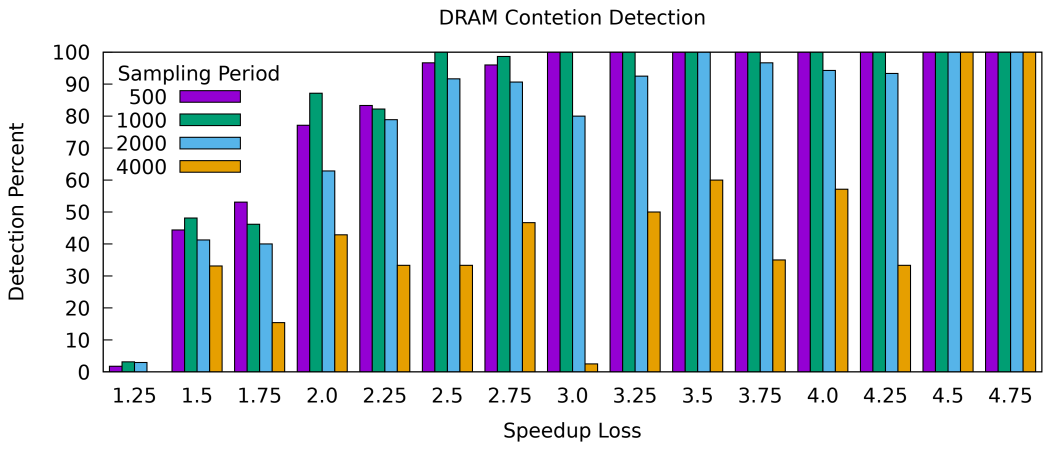


Figure 7.15. The Percentage of DRAM contention detection for intervals of speedup loss. The values on the horizontal axis denote the upper bound of the interval.

Influence of the Sampling Period

As shown in Figure 7.15, sampling periods from 500 to 2000 produce reliable results. There is a slight decrease in detection results with a sampling period of 2000. Higher sampling periods, like 4000 shown in Figure 7.15 do no longer result in reliable detection of DRAM contention. Figure 7.16 shows in more detail how the DRAM contention detection depends on the sampling rate. The experiments for this diagram were executed on Arcturus. The maximum number of threads was picked. At 22 threads, all three benchmarks show a speedup loss between 2.5 and 3. We expect that for all benchmarks a DRAM contention is detected. But this is only true up to a sampling period of 1000. At a sampling period of 2000, already some cases in countv and sumv are missed. At a sampling period of 4000 or higher, only in dotv DRAM contention is detected. At a sampling period of 16,000 or higher, also the detection results of dotv deteriorate. Based on those findings, we chose a sampling period of 1000 for the next evaluation.

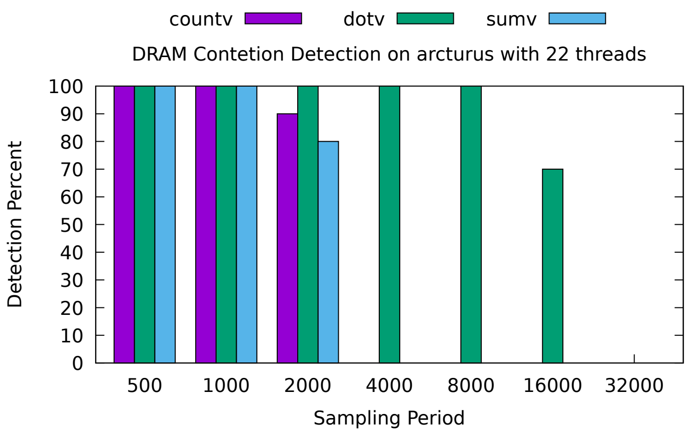


Figure 7.16. The detection accuracy of DRAM contention depends on the sampling period. A sampling period of 500 or 1000 allows reliable detection in these benchmarks.

Detection Result Details for Systems and Benchmarks

In Figure 7.17 the detection results are plotted over the number of threads. With a higher number of threads, the DRAM contention gets more severe, as shown in Figure 7.14. The detection results in Figure 7.17 should be compared to the performance loss plotted in Figure 7.14. When comparing both diagrams, we can see that when the contention gets worse, it is more likely to be detected. But there are differences between the systems and benchmarks. On Rigel, even though the speedup loss of countv and sumv is almost the same, only in sumv DRAM contention is detected. In countv, it is never detected. On Spica, the speedup loss of countv and sumv is also very similar. At 8 threads, contention in sumv is detected with 100%, but for countv, it takes 14 threads to reach a detection of 100%. The cases of missed DRAM contention detection always occur at low speedup losses. Severe DRAM contention with a speedup loss of more than 3 times, is always detected.

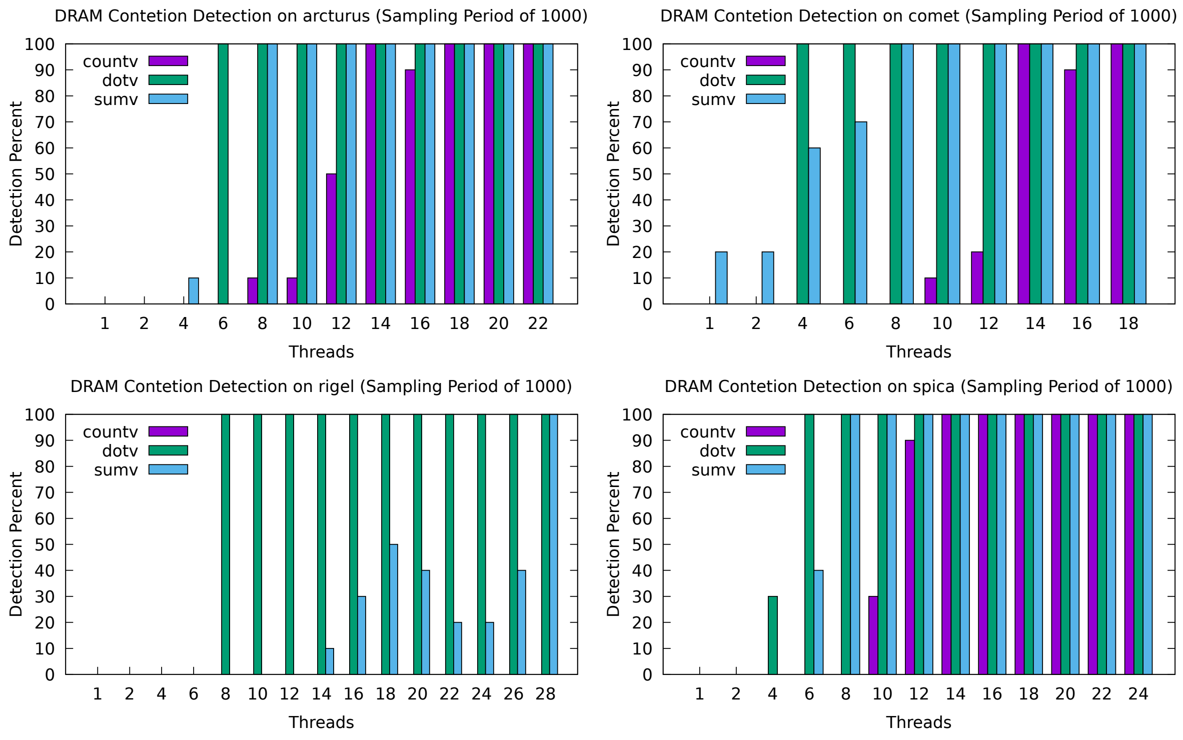


Figure 7.17. The higher the number of threads is, the more severe the DRAM contention is, and the more likely it is to be detected. A 100% detection rate means that contention was detected in 10 out of 10 experiments

Relationship of Latency and Speedup Loss

The DRAM access latency, on which the DRAM contention detection is based, is shown in Figure 7.18. With the number of threads, and thus the speedup loss, the latency rises. A correlation diagram of the speedup loss and the latency is shown in Figure 7.19. The latency rises with the speedup loss. Ideally, the contention detection metric should be proportional to the speedup loss, and the same for every benchmark. There is a small difference between the different benchmarks. On Rigel, the latency graph of dotv remains rather flat after a speedup loss of more than two. Also, the point where the limit is crossed is different for benchmarks and systems. However, severe DRAM contention of more than 2.25x speedup loss always results in an average latency above the limit. The latency also differs between the benchmarks. Dotv always has the highest latency, followed by sumv and countv. When we look at the actual scalability (Figure 7.13) of the large array version, and not the speedup loss, we see that dotv has the worst scalability, followed by sumv and finally countv. The benchmarks with low scalability have a high latency. This bad scalability may come from additional factors that also delay the instruction processing, which then increases the latency. We already know from our prior experiments that the instruction sampling latency consists of in-core processing delays as well as memory access delays.

The coefficient of variation of the latency measurement is on average 35.01% and at most 53.24%. It indicates that the latency can vary between experiments. A large number of samples or the repetition of experiments is required to obtain reliable results.

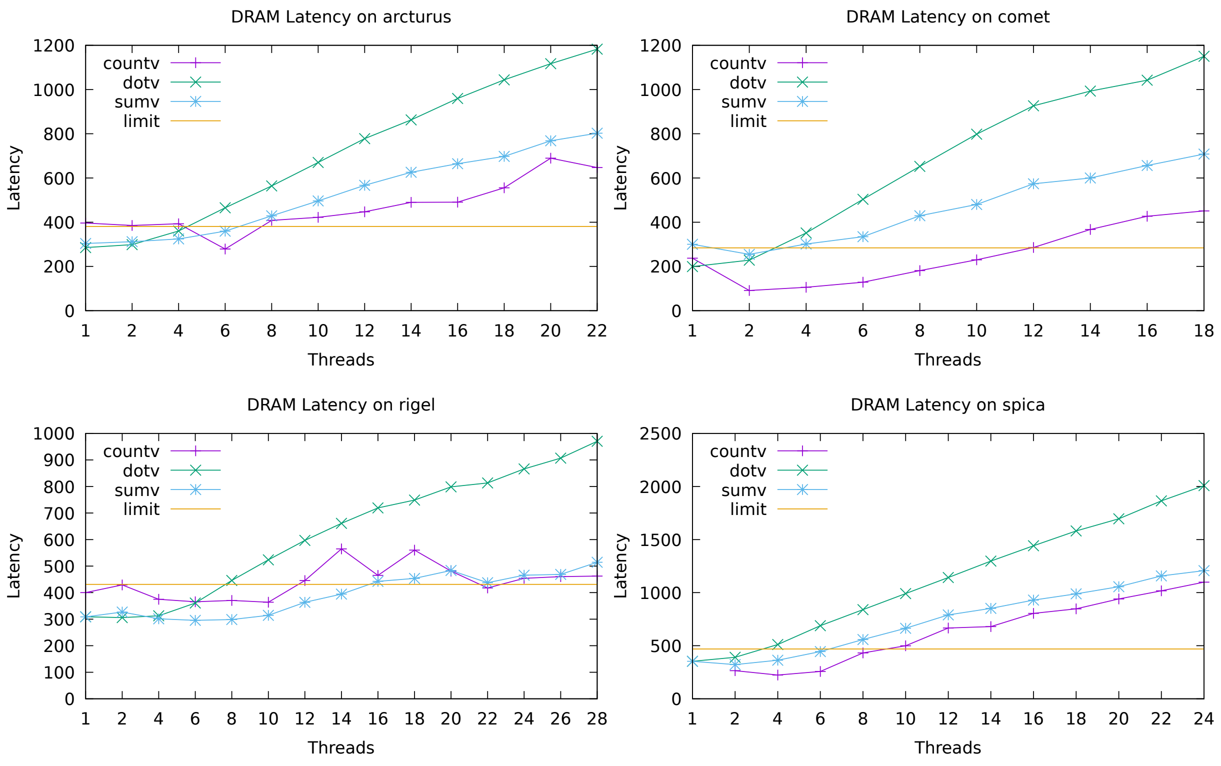


Figure 7.18. The more severe the performance problem is, the higher the DRAM access latency rises. The latency is higher in the benchmarks that suffer more from DRAM contention.

Comparison to DRAM Bandwidth

The measured DRAM bandwidth, shown in Figure 7.20 does not increase above the hardware limit. The DRAM bandwidth can not be used to judge the severity of the contention. In contrast, the DRAM access latency rises higher, even if we already hit the bandwidth limit of the system. Thus, the latency can be an indicator of the severity of the contention. The correlation of speedup loss and latency in Figure 7.19 shows this.

A specific example of this is dotv on Comet. The scalability loss differs depending on the number of threads. It is about 2 with 6 threads and about 4.5 with 18 threads (Figure 7.14). Not matter if we run the application with 6 threads or 18 threads, there is no difference in the bandwidth (Figure 7.20). In contrast, the latency differs between 500 cycles and 1100 cycles (Figure 7.18).

Just by looking at the DRAM bandwidth, one can not judge if an application is suffering from limited DRAM bandwidth. The difference between the benchmarks is also not visible when looking at the DRAM bandwidth. For example, on Comet, with 6 threads the bandwidth of all benchmarks is similar and close to the maximum. Just by looking at the DRAM bandwidth, one could conclude that all benchmarks suffer from DRAM contention. However, at this point, the speedup loss ranges between 1.2 for countv and 2.1 for dotv. The DRAM access latency shows this difference. It ranges from about 160 for countv, to over 500 for dotv. By using the latency, we can see that countv is not impacted by the constraint DRAM bandwidth, but dotv is suffering from the limited DRAM bandwidth.

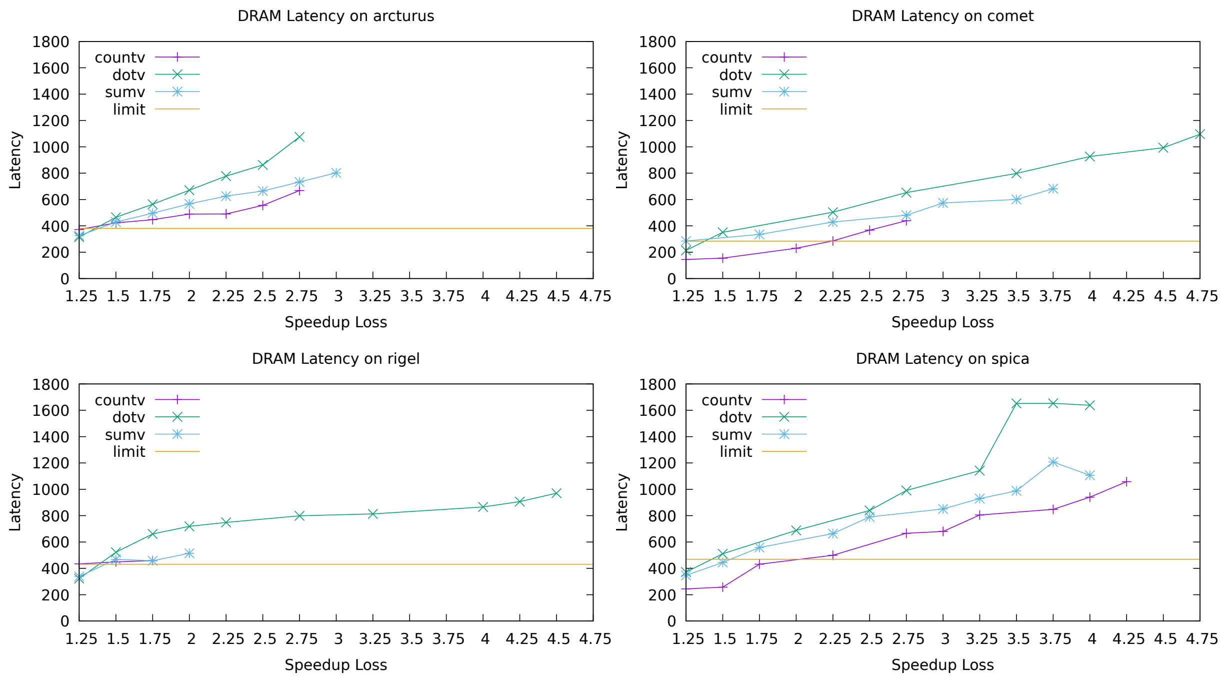


Figure 7.19. The average DRAM access latency for intervals of speedup loss. The upper bound of the speedup loss intervals is denoted on the horizontal axis.

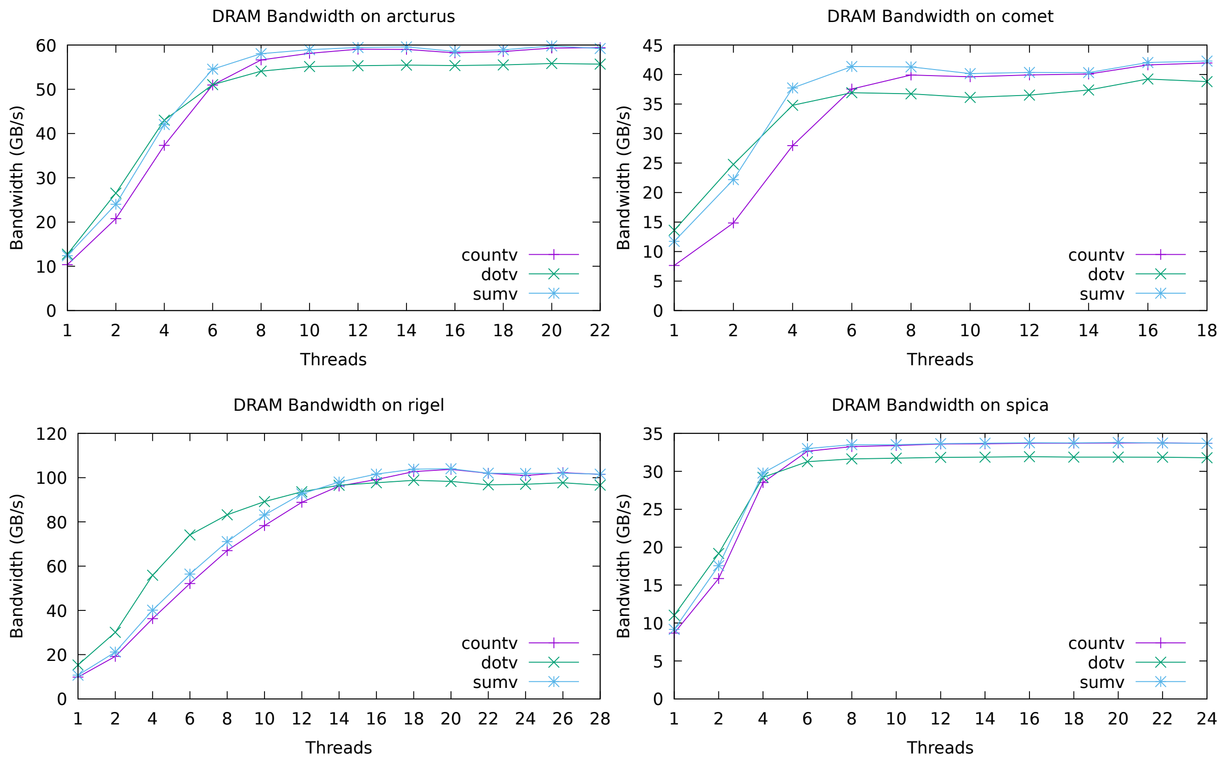


Figure 7.20. The bandwidth reaches a upper limit and can not be used to judge the severity of the contention problem. The displayed bandwidth is the combined read and write bandwidth measured with IMC counters.

False positives

False positive detection of DRAM contention occurs in these experiments. A very low speedup loss, for example, less than 10%, is a case where DRAM contention has only a minimal influence on the performance. If such cases, with very low speedup loss, are reported as DRAM contention it is a false positive. There is no exact definition of such a speedup loss threshold. We use 10% as an example here. Table 7.5 shows all of those cases in experiments with a sampling period of 500 or 1000. Higher sampling periods are not included because we showed before that results may get unreliable.

Table 7.5. A summary of the cases where DRAM contention was detected despite the speedup loss being less than 10%.

System	Benchmark	Threads	Speedup Loss	Detection Count	Average Latency
Arcturus	sumv	4	1.07	2 out of 20	373
Comet	sumv	1	1.0	5 out of 20	333
Comet	sumv	2	1.05	6 out of 20	331

All of the cases occur in the sumv benchmark with a low thread count of up to 4. A thread count of 4 is not enough to saturate the memory bandwidth of current systems, as we have shown in Section 4.3.4. A differentiation of DRAM limitations and core-local limitations as described in Section 4.3.4 would solve those false positive detections.

Remote DRAM Bandwidth Contention

So far we evaluated the discovery of local DRAM contention. But the method also works on the remote DRAM, as we show with the following experiments. For these experiments, we allocated all threads on one node and all the data on the other node. This way all DRAM accesses are remote DRAM accesses. The latency limit for remote DRAM contention is higher than for the local DRAM contention. It is also determined using a pointer chasing benchmark. Because the remote DRAM bandwidth is more limited compared to the local DRAM bandwidth, the speedup over the serial version is lower. On Arcturus, Comet, and Spica, the speedup over the sequential version is limited to about 3x. On Rigel, a sightly higher speedup of 6x is achieved. Because the performance of the version with a data size that fits into the L3 cache does not change, the speedup loss reaches higher values of up to 7x.

Figure 7.21 shows the detection results. The trend is similar to the local DRAM contention. Higher values of the speedup loss are always detected, and low values are only partially detected. In the local DRAM case, a speedup loss of greater than 2.25 leads to certain detection of the problem. In the remote DRAM case, a speedup loss of greater than 3.5 leads to certain detection. A higher sampling period has fewer effects on the detection accuracy compared to the local DRAM.

A look at the remote DRAM latency confirms the similarity to the local DRAM results. Figure 7.22 shows the average latency at intervals of speedup loss. Even though the measured latency values and the hardware thresholds differ, the general trend is the same as for the local DRAM latency shown in Figure 7.19.

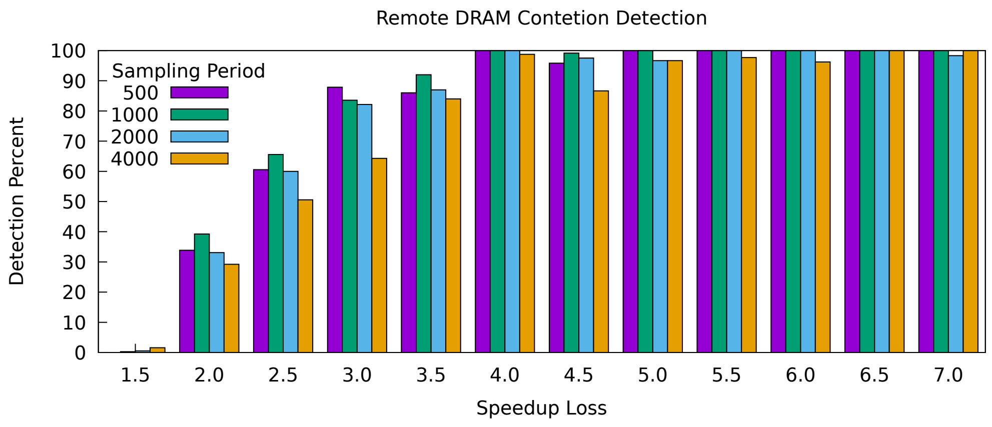


Figure 7.21. The Percentage of remote DRAM contention detection for intervals of speedup loss. The values on the horizontal axis denote the upper bound of the interval.

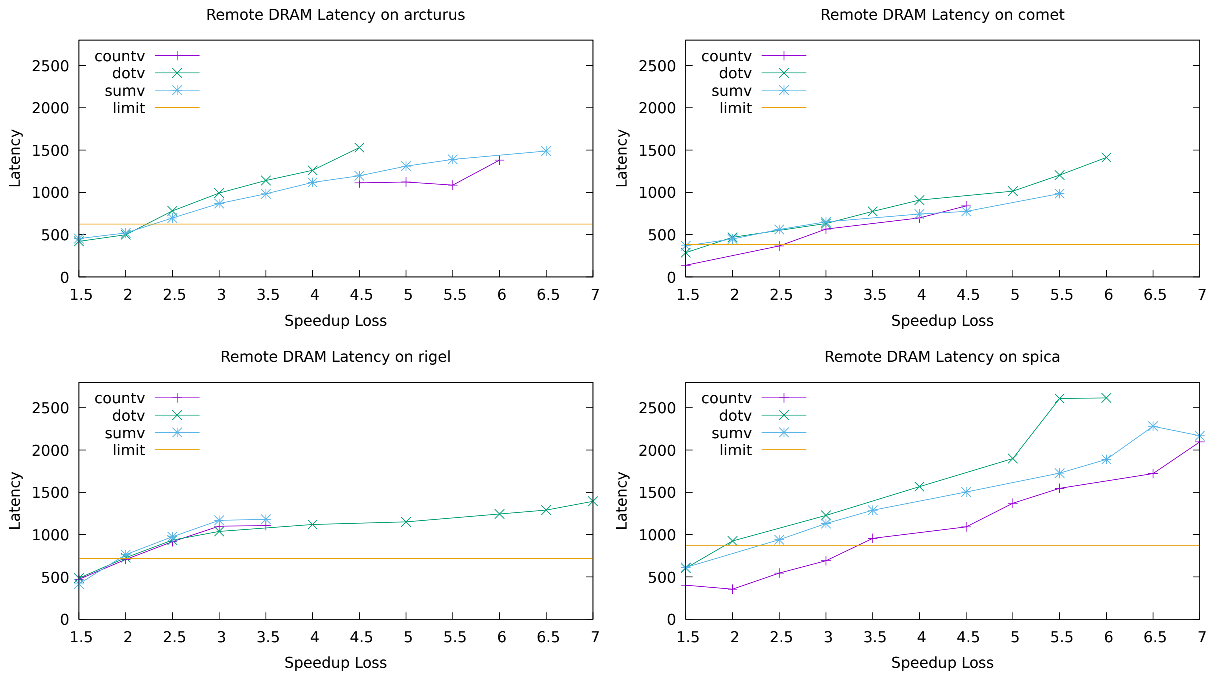


Figure 7.22. The average remote DRAM access latency for intervals of speedup loss. The upper bound of the speedup loss intervals is denoted on the horizontal axis.

7.3.2 Small Kernels

All of the Stream benchmarks suffer from DRAM contention. It is reliably detected at sampling periods ranging from 500 to 32000 as shown in Figure 7.23 to Figure 7.26. Only the copy benchmark shows slightly lower detection rates at high sampling periods as depicted in Figure 7.24. All of the Stream benchmarks have a DRAM latency much higher than the system latency as shown in Figure 7.27. As expected, this indicates the severe bandwidth boundness of those benchmarks. Because of high cache hit rates mMat does not have enough DRAM access samples for detection and for calculation of the latency.

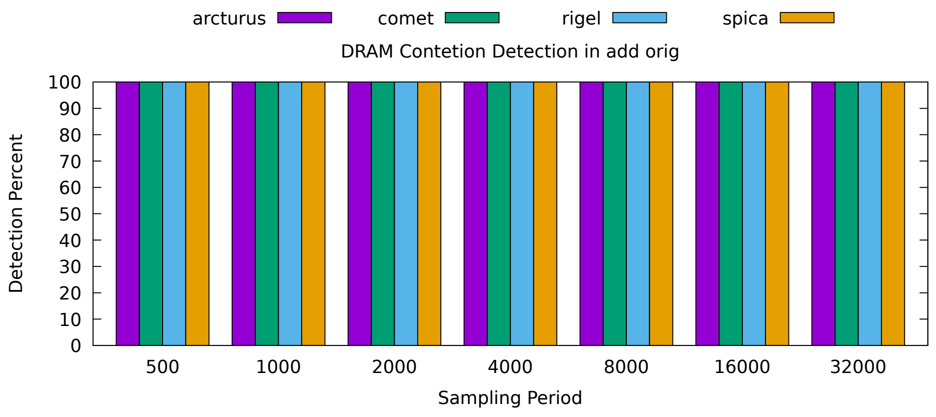


Figure 7.23. The DRAM bandwidth contention detection results of Stream add.

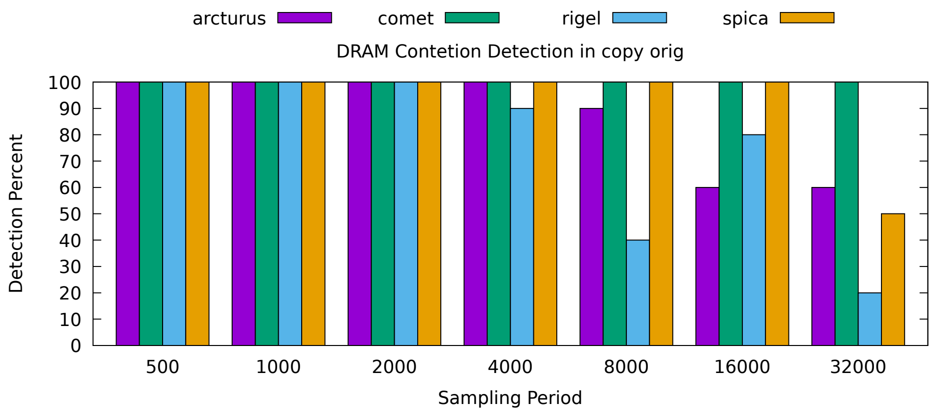


Figure 7.24. The DRAM bandwidth contention detection results of Stream copy.

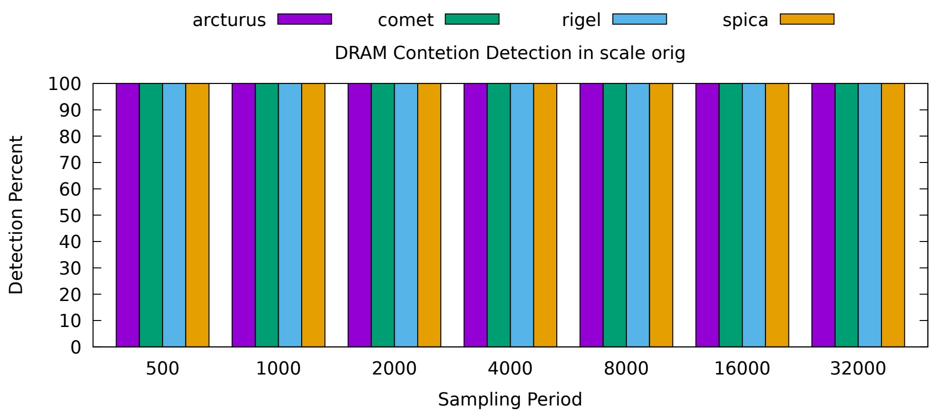


Figure 7.25. The DRAM bandwidth contention detection results of Stream scale.

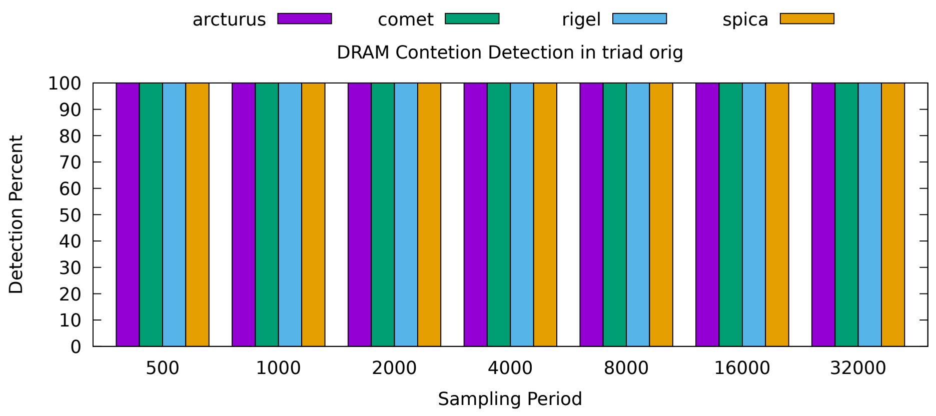


Figure 7.26. The DRAM bandwidth contention detection results of Stream triad.

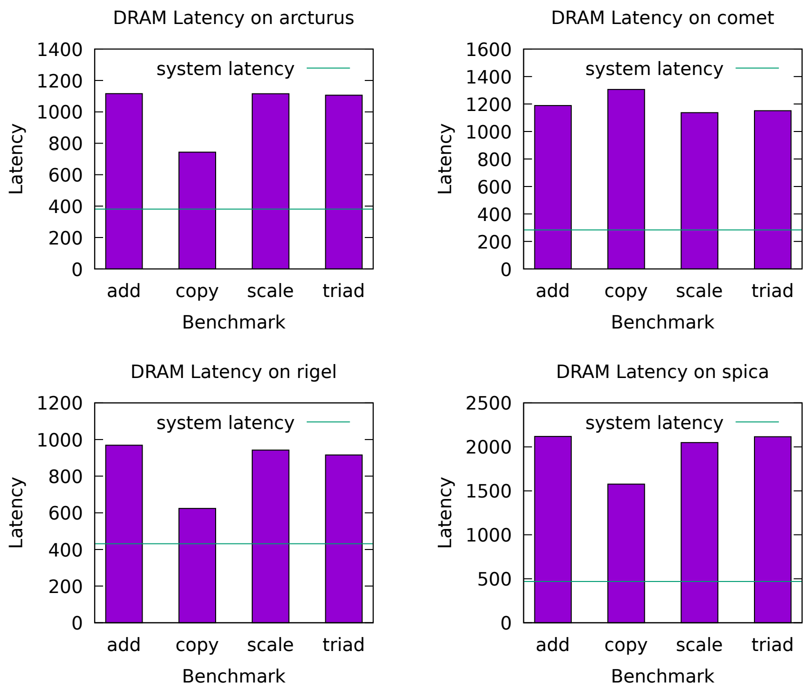


Figure 7.27. The average local DRAM access latency of the Stream benchmarks.

7.3.3 PARSEC Benchmarks

The method of evaluation is different from the artificial benchmarks because no clear metric for the performance loss caused by DRAM contention exists. Still, the results show that our DRAM contention detection approach also works for realistic applications.

Benchmarks

We evaluated the automatic discovery using all of the 13 PARSEC [10] benchmarks. All PARSEC benchmarks were executed with the native input set and the speedup results are based on the time required for the region of interest (ROI). We use the maximum number of physical cores available on the systems. Those numbers are shown in Table 7.1. Except for Facesim and Fluidaniamate, where we used 32 threads due to limitations in the benchmark. In this paper, we present the results for the pthreads version of all benchmarks except for Freqmine, which only has an OpenMP version. PerfMemPlus supports other parallelization methods such as Intel TBB, but we omit the results in this paper because it does not provide any relevant insights.

The PARSEC benchmarks were analyzed in several previous studies [9, 10, 23, 26, 61, 67, 75, 102, 111] so that we conclude the ground truth of the performance problems from the existing literature. Only two benchmarks have known memory-related performance issues.

Canneal suffers from memory issues [26] that cause slowdown and it has the second-highest bandwidth requirement [10] of all PARSEC benchmarks.

Streamcluster is sensitive to DRAM speed [9] and has the highest main memory bandwidth requirement of all PARSEC benchmarks [10]. There are also NUMA issues due to the allocation of the main array to one node and accesses from both nodes [61]. Data locality is worsened by shuffling pointers to data between algorithm iterations [75].

All benchmarks were executed 10 times. The provided performance results are the average of those 10 repetitions. For detection results, we show in how many of the 10 cases DRAM contention was found.

Cases of DRAM Contention Detection

PerfMemPlus detected DRAM contention in some of the PARSEC benchmarks. In canneal, one function that accesses two different objects was pointed out. The reported function swap_cost contributes more than 75% to the total execution time. In streamcluster, PerfMemPlus reported one function accessing one object. The reported function, called pgain contributes more than 80% to the total execution time. DRAM contention was also reported in one case in the benchmark bodytrack, running on Rigel. However, this case occurred in the tar program which is executed to prepare the required data to run this benchmark. In fluidanimate and facesim, instances of DRAM contention are reported, when profiling is done with low sampling periods. However, they occur in functions that contribute less than 5% to the total execution time. When using sampling periods higher than 4000, there are not enough DRAM access samples in the concerned functions. Because of those small contributions of the functions to the overall execution time, we did not consider to optimize those applications. In fluidanimate, those functions are AdvanceParticlesMTi, ProcessCollisionsMTi and RebuildGridMti. In facesim, those functions are One_Newton_Step_Toward_Steady_State_CG_
Helper_I, One_Newton_Step_Toward_Steady_State_CG_Helper_III and dP_From_dF. In all other benchmarks, no instance of DRAM contention was reported. This matches our expectations because no DRAM contention is known in those benchmarks from the existing literature.

Performance Optimization

We tried to fix the issue of DRAM contention in canneal and streamcluster. In canneal, we applied interleaved allocation to the two reported objects. The details of the findings and optimization are described in the case study in Section 7.9.1. For streamcluster, we applied two different optimizations. The first one is to use interleaved allocation for the identified object. The second one changes the data access pattern and results in better cache hit rates. On Spica, the L1 hit rate increases from 72% to 94%. The details of this optimization are described in the case study in Section 7.9.2.

Figure 7.28 shows the speedup that was obtained by these modifications. In contrast to the artificial benchmarks described in the last section, this speedup is not directly a measure of the severity of the contention. The quality of the optimization also influences the achieved speedup. The execution time measurements, that Figure 7.28 is based on have an average coefficient of variation of 28.15% and at most 32.51%. In canneal, the DRAM contention problem is not severe. A maximum speedup of 20% can be achieved on Spica, which is the system with the lowest DRAM bandwidth. On Comet, there is a 1% slowdown. On the other two systems, there is a small speedup of a few percents. The DRAM contention is more severe in streamcluster. The interleaved allocation brings higher speedups compared to canneal. With interleaved allocation, a speedup between 7% on Rigel and 65% on Spica was obtained. With the copy shuffle optimization that reduces the number of DRAM accesses, a speedup between 40% on Rigel and 2.54 times on Spica was achieved. On the systems with lower DRAM bandwidth, a higher speedup was achieved. The existing literature also states that streamcluster has higher memory bandwidth requirements compared to canneal [10].

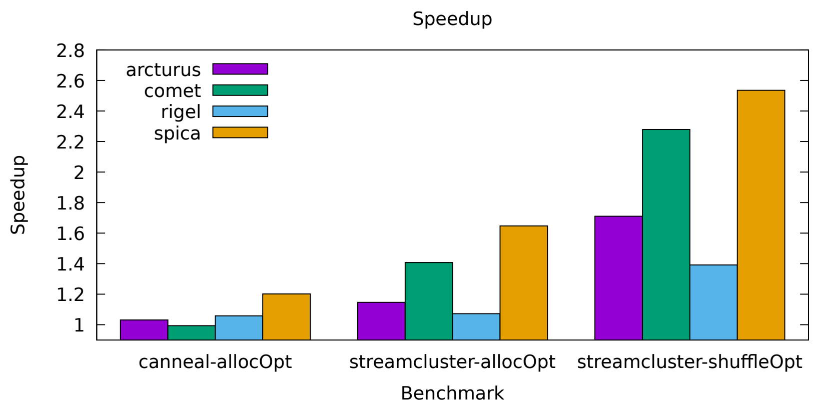


Figure 7.28. The speedup obtained through optimization of the allocation in streamcluster and canneal. The optimization of the shuffle operation of streamcluster leads to an even higher speedup.

Detection Results in Canneal

Figure 7.29 shows that on Spica, the detection rates are the highest. On Comet and Arcturus, there are some cases of detection. On Rigel, DRAM contention is never detected. Rigel is also the system with the highest DRAM bandwidth. Figure 7.30 shows that even with the improved allocation, cases of DRAM contention are reported. But there are fewer cases reported compared to the original version. Overall canneal is only slightly affected by DRAM contention, and the detection results reflect that. Only in some cases, it is detected. We see a similar result in the artificial benchmarks. If the DRAM contention is not severe, detection is not guaranteed.

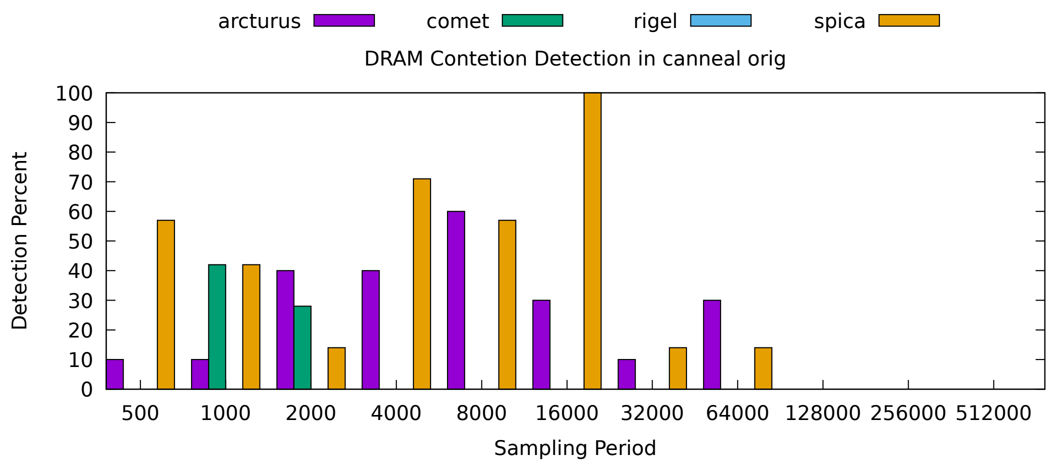


Figure 7.29. The detection of DRAM contention in the original version of the canneal benchmark.

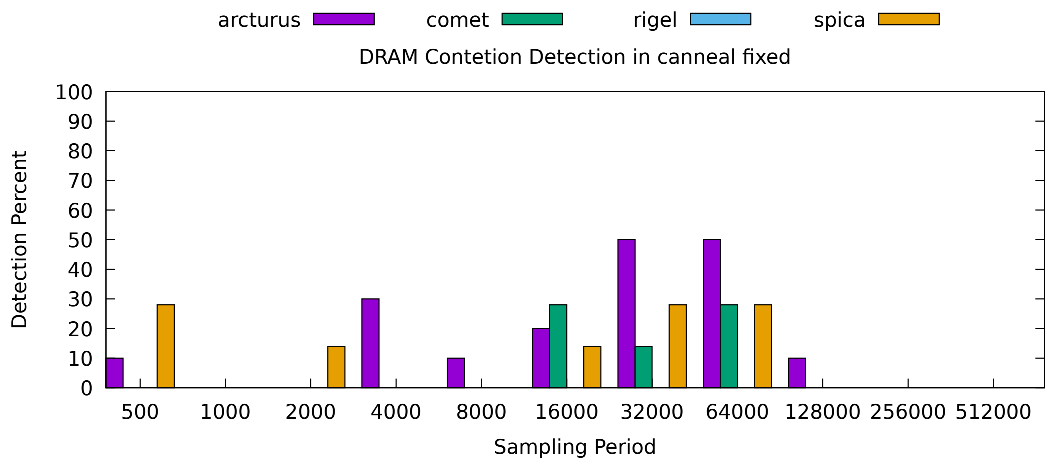


Figure 7.30. The detection of DRAM contention in the optimized version of the canneal benchmark.

Detection Results in Streamcluster

Figure 7.31 shows the DRAM contention detection results of the unoptimized version of streamcluster. Spica has the lowest DRAM bandwidth and DRAM contention is always detected on this system. On Arcturus, it is also often detected. Both of them are Broadwell architecture. On Comet, even though speedups were higher than on Arcturus the detection results are worse. The contention is only detected in a few cases. On Rigel, where interleaved allocation brings only a slight performance benefit, DRAM contention is only found once.

After applying the interleaved allocation optimization, the number of detections drops, as shown in Figure 7.32. Only some cases of DRAM contention are found on Arcturus and Comet.

It drops even further after applying the better optimization, which improves cache hit rates. Those results are shown in Figure 7.33. The number of DRAM accesses drops significantly. At a sampling period of 8000, the average number of DRAM samples taken in a profile of the identified function drops from 399 to 18. In many cases, there are not enough samples available to calculate a reliable average of the latency based on the conditions described in Section 4.2.

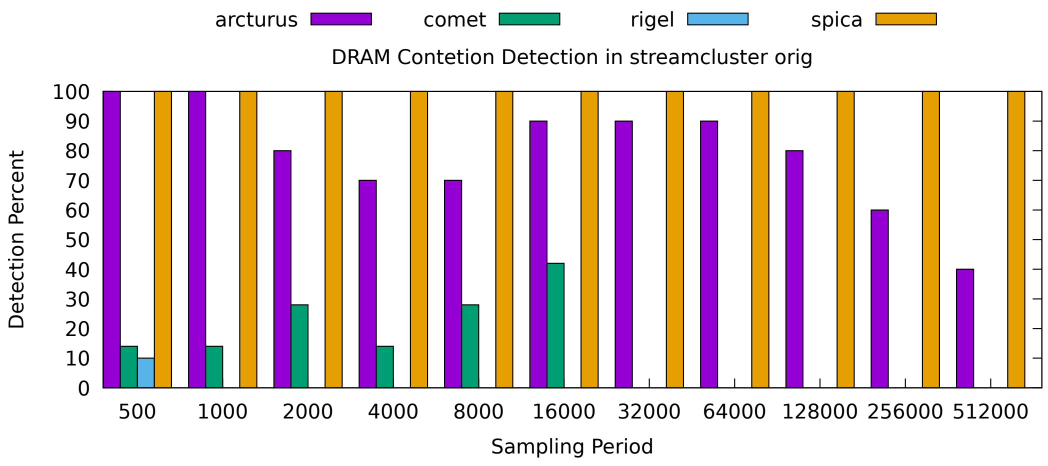


Figure 7.31. The detection of DRAM contention in the original version of the streamcluster benchmark.

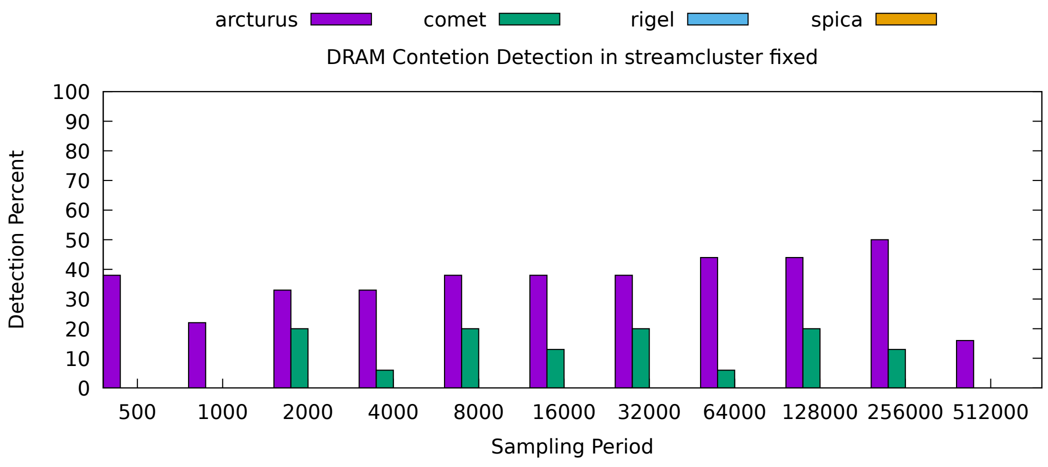


Figure 7.32. The detection of DRAM contention in the streamcluster benchmark with interleaved allocation of the array block.

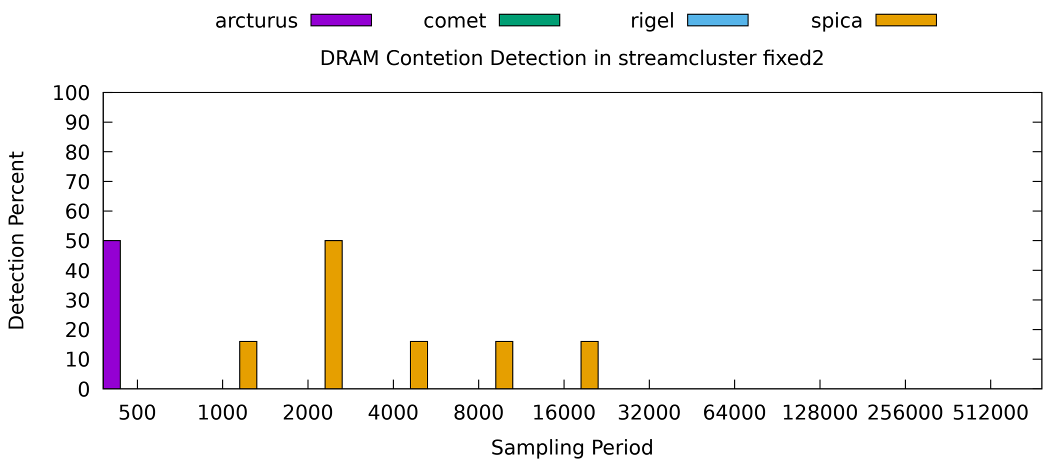


Figure 7.33. The detection of DRAM contention in the streamcluster benchmark with an optimized shuffle implementation.

DRAM Latency

The average DRAM access latency of canneal and streamcluster in both the original version and fixed version is shown in Figure 7.34. The maximum coefficient of variation of the latency is 60.80% for canneal and 14.91% for streamcluster. On average it is 21.17%. We do not include the version of streamcluster with the improved shuffle operation. Because of the improved cache hit rates in this version, most of the time there are not enough DRAM access samples available to calculate an average DRAM latency, that would satisfy the conditions mentioned in Section 4.3.5. The hardware threshold is drawn as a blue line. Spica has the lowest DRAM bandwidth and also benefits the most from the optimizations. The latency values on Spica, for the initial version, are far above the limit, whereas the fixed version has a latency below the limit. On Rigel, the latencies are always below the limit. Rigel is the system with the highest memory bandwidth and the least benefit from the optimizations. On Arcturus and Comet, the applications suffer slightly from the limited DRAM bandwidth. The latencies are close to the limit, and there is no clear difference between the original and optimized version. In all cases, the latency of streamcluster is higher than that of canneal. Streamcluster suffers more from the DRAM contention, and interleaved allocation leads to higher speedups for streamcluster compared to canneal. Overall we can see a trend that more severe DRAM contention results in higher latency. Also in realistic applications, the DRAM access latency is a metric for the severity of DRAM contention.

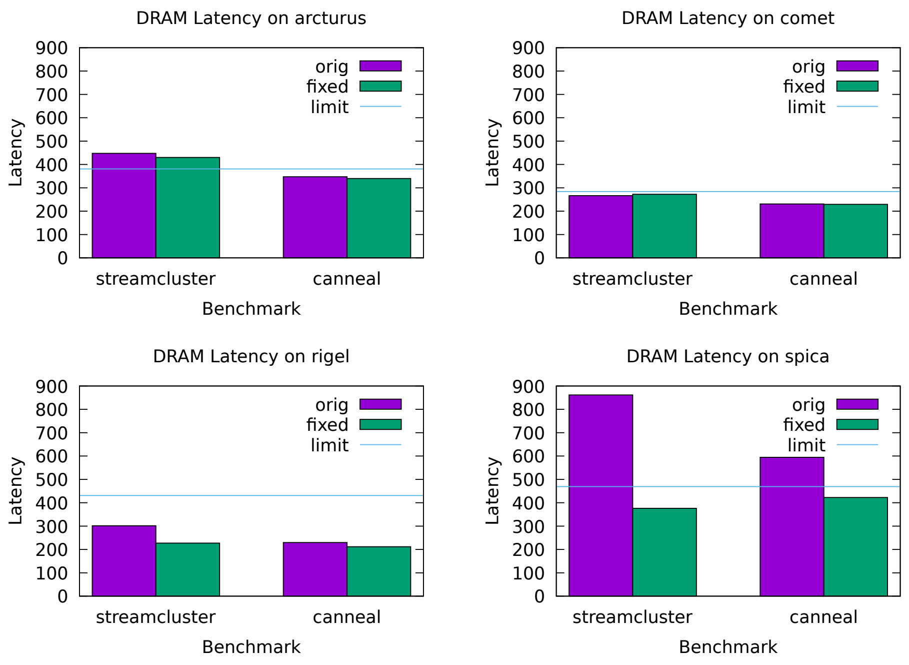


Figure 7.34. The average DRAM access latency of the streamcluster benchmark and canneal benchmark.

Influence of the Sampling Period

The effect of the sampling period is not visible as clearly as for the artificial benchmarks. In the artificial benchmarks, only a sampling period of 500 or 1000 produced reliable results. In streamcluster and canneal, also higher sampling periods are viable. We observe slightly worsened detection results at a sampling period of 32,000 or higher. The PARSEC benchmarks run much longer than the artificial benchmarks. Thus at the same sampling period, the PARSEC benchmarks will accumulate more samples. A suitable choice of the sampling period depends on the profiled application.

7.4 NUMA Imbalance

Because the NUMA imbalance metric is simply based on the number of accesses, we do not evaluate it with special micro-benchmarks. We conduct the evaluation using the small kernels and the PARSEC benchmarks.

7.4.1 Small Kernels

For the Stream benchmarks, the NUMA imbalance is low. This is the expected behavior because the initialization, which means the first touch of the arrays is done using all of the available threads. The profiling data of the matrix multiplication benchmark does not contain enough DRAM accesses to calculate the NUMA imbalance metric. The results are shown in Table 7.6. Because the measured low imbalance matches our expectations, we did not implement and test interleaved allocation.

Table 7.6. The NUMA imbalance of the small kernels. For the matrix multiplication benchmark there are not enough DRAM access samples to calculate the imbalance.

	add	copy	scale	triad	mMat
Arcturus	0.06	0.18	0.04	0.06	0
Comet	0.08	0.10	0.05	0.06	0
Rigel	0.02	0.04	0.02	0.02	0
Spica	0.05	0.17	0.05	0.06	0

7.4.2 PARSEC Benchmarks

Out of the PARSEC Benchmarks only two show a DRAM contention problem. The NUMA imbalance of those benchmarks is evaluated in Section 7.9.1 and Section 7.9.2.

7.5 Channel Imbalance

The evaluation of the channel imbalance serves two purposes. First, we want to know if the measurement works correctly. For this part of the evaluation, we use a micro-benchmark that deliberately causes an imbalance by using only specific channels. Second, we want to evaluate the imbalance in real applications. For this purpose, we use two groups of benchmarks. First, small kernels with a uniform and easily analyzable behavior and second the realistic PARSEC benchmarks.

7.5.1 Micro-Benchmark

With a micro-benchmark that accesses only specified channels, we verify if the traffic on the separate channels is correctly counted and if it is distinguishable. This benchmark works on Arcturus and Spica because for those systems we have reverse engineered the address mapping. The benchmark first allocates an array. Each individual element has a size of 64 Bytes to match the cache line size. Addresses below the granularity of a cache line do not change the mapping thus smaller elements are not of interest. Second, for each index within the array, the physical address is calculated. Each physical address is tested against the address mapping. If the address lies withing the specified channel, rank, and bank the index is saved in a list. After this preparation follows the actual benchmark step. In a parallel for loop, the saved indexes are used and the corresponding data in the main array is accesses. Each memory access is followed by cache line flush instruction to enforce a load from DRAM.

We show the results in Figure 7.35 and Figure 7.36 respectively. On both systems, we observe that the used channels clearly stand out from the unused channels. We also conduct the experiments with other channel number combinations and found that the approach works regardless of the channel that was used. In other words, also the usage of only channel 3 can be seen. On Arcturus, channel specification for the benchmark differs from the channel specification for the measurement. Because in hardware the channels are named as 0, 1, 4, and 5. However, the access benchmark is not aware of this hardware names and uses the channels 0, 1, 2, and 3. On Spica, the channel naming of measurement and access benchmark is the same.

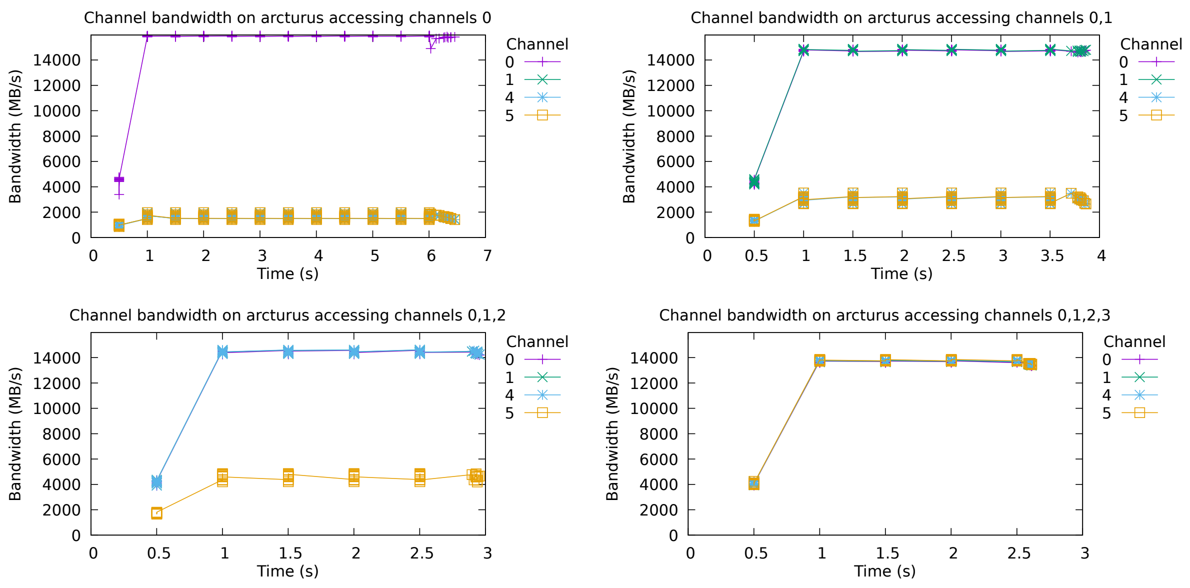


Figure 7.35. The measured consumed memory bandwidth on different channels on Arcturus when enforcing accesses to specific channels.

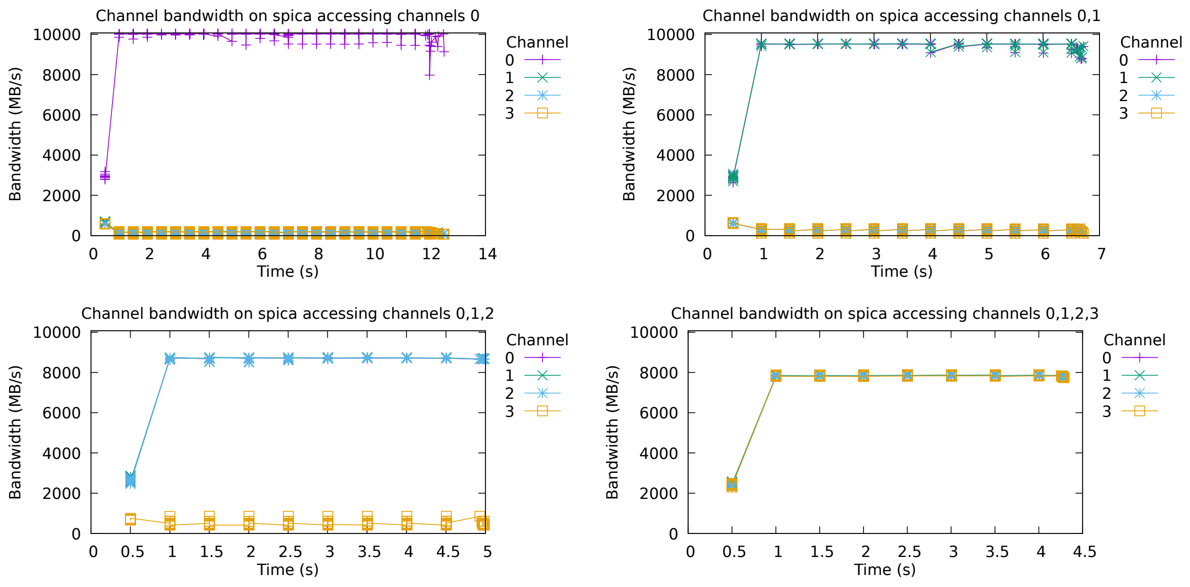


Figure 7.36. The consumed memory bandwidth on different channels on Spica when enforcing accesses to specific channels.

7.5.2 Small Kernels

The four Stream benchmarks in Figure 7.37, Figure 7.38, Figure 7.39, and Figure 7.40 show high bandwidth usage on all channels. It is the same on all channels and all systems. The channel bandwidth of the matrix multiplication benchmark shown in Figure 7.41 is low and the same for all channels.

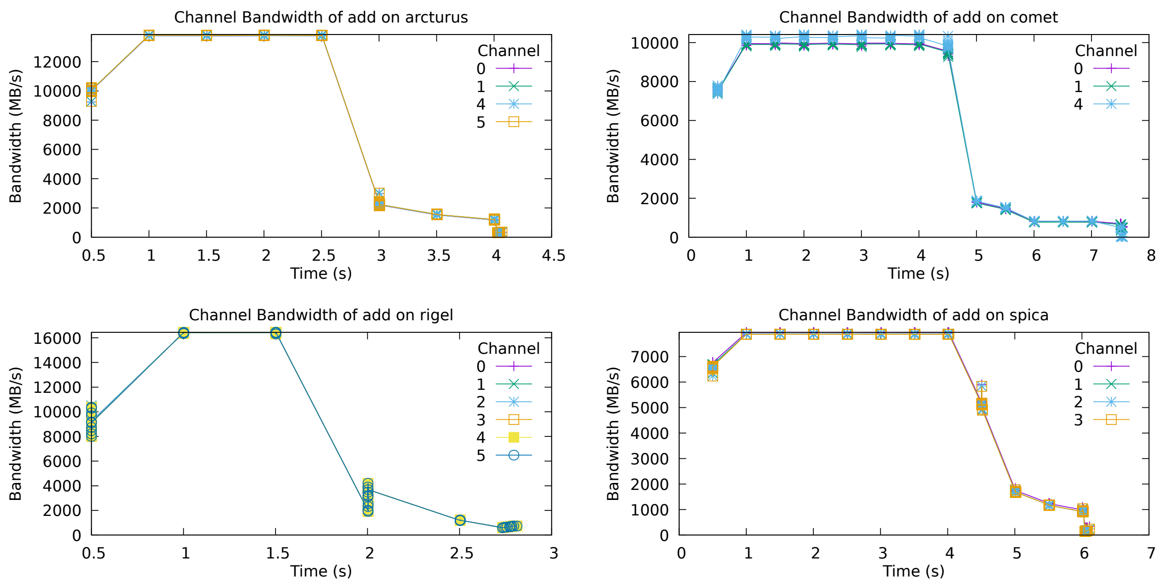


Figure 7.37. The consumed memory bandwidth on different channels when profiling Stream add.

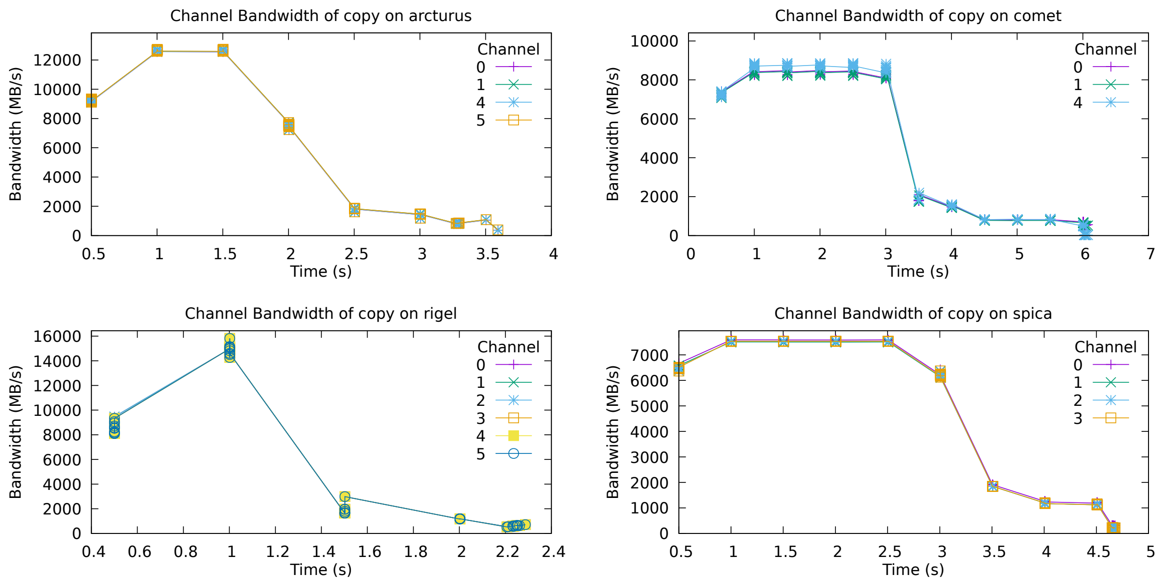


Figure 7.38. The consumed memory bandwidth on different channels when profiling Stream copy.

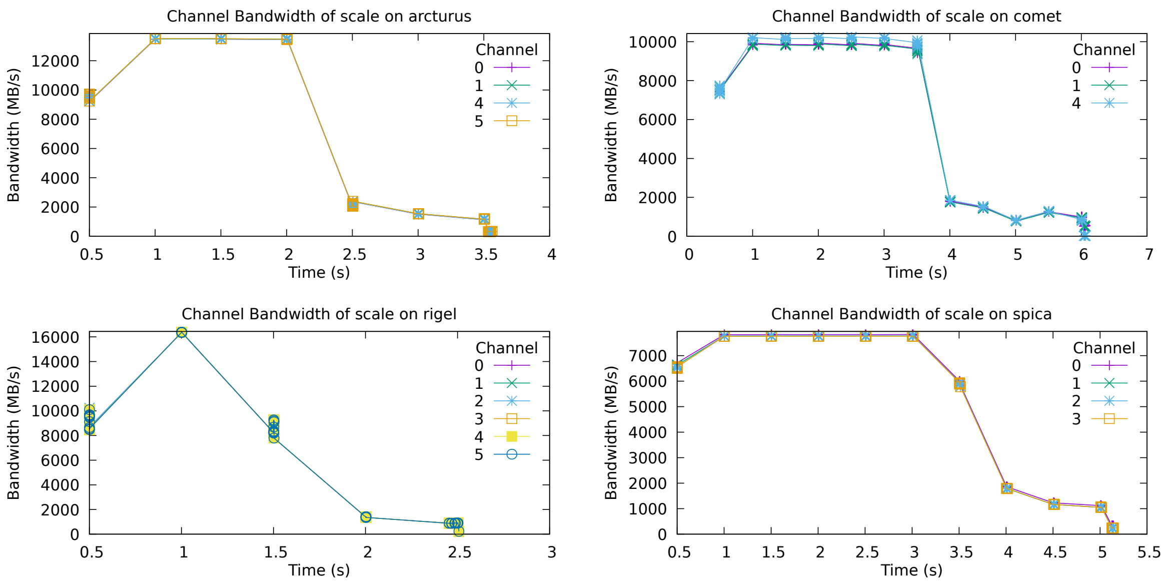


Figure 7.39. The consumed memory bandwidth on different channels when profiling Stream scale.

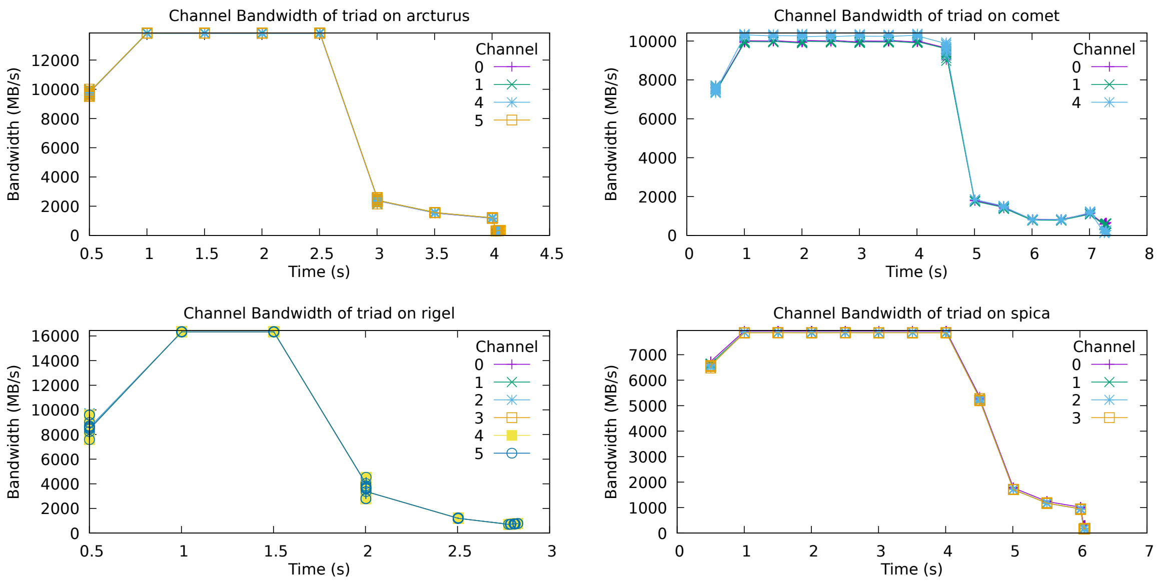


Figure 7.40. The consumed memory bandwidth on different channels when profiling Stream triad.

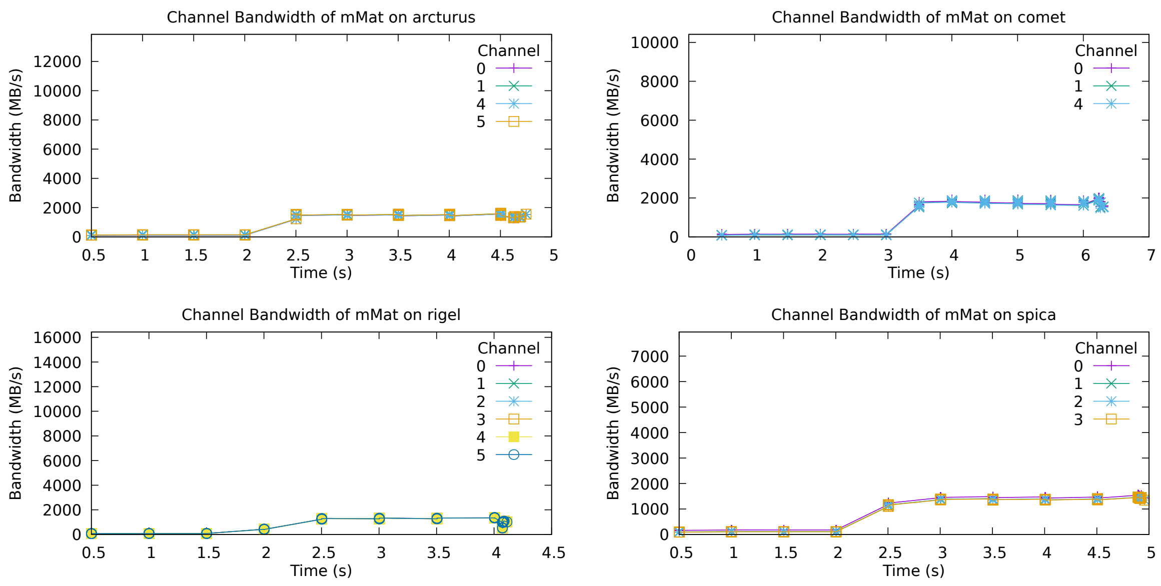


Figure 7.41. The consumed memory bandwidth on different channels when profiling OpenBLAS DGEMM.

7.5.3 PARSEC Benchmarks

Among the PARSEC benchmarks, there are many with very low bandwidth requirements. Those include bodytrack (Figure 7.43), dedup (Figure 7.45), facesim (Figure 7.46), ferret (Figure 7.47), and swaptions (Figure 7.50). Some benchmarks show short spikes of medium bandwidth requirements. Those are blackscholes (Figure 7.42), canneal (Figure 7.44), and freqmine (Figure 7.48). In all of the above-mentioned benchmarks, the bandwidth is equally balanced on all channels.

The benchmark with a long period of high bandwidth consumption is streamcluster shown in Figure 7.49. On Rigel and Spica, there is a slight difference in the bandwidth on different channels. The channels can be divided into two groups. One group of channels shows a lower bandwidth and the other group a higher bandwidth. The difference between the two groups is 5GB/s on Rigel and 2GB/s on Spica. On Arcturus and Comet, the bandwidth consumption is well balanced.

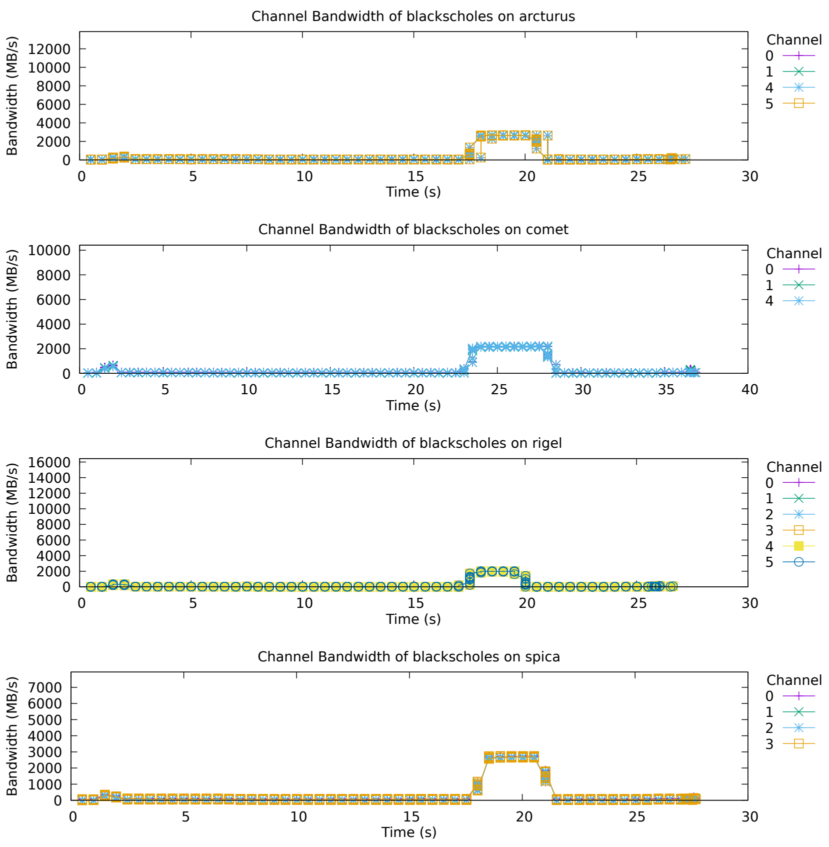


Figure 7.42. The bandwidth on the different channels when profiling blackscholes.

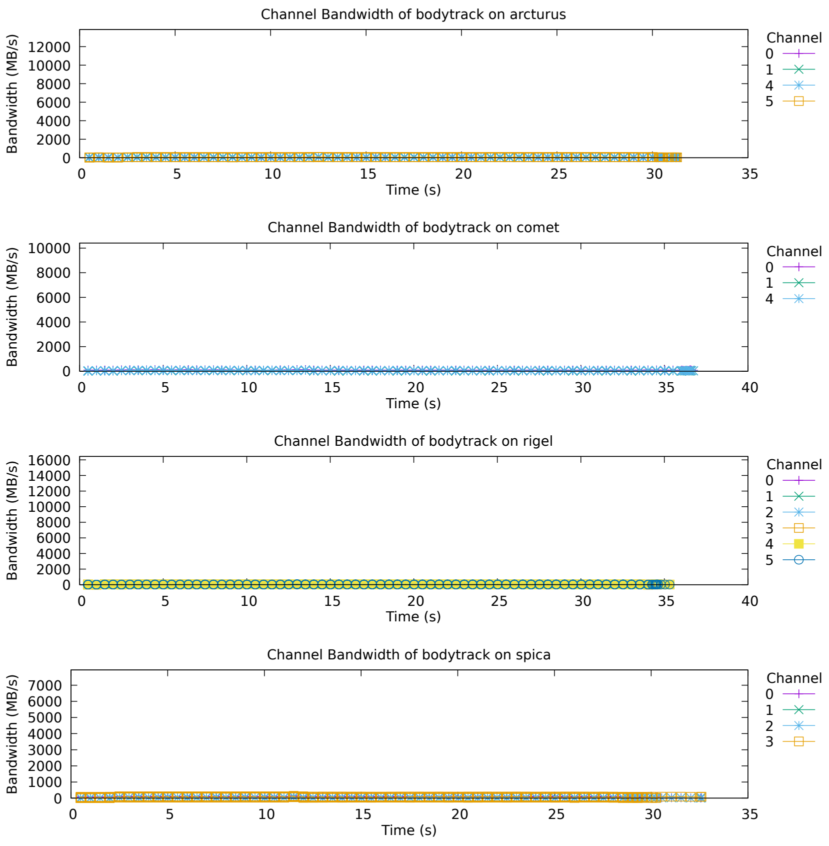


Figure 7.43. The bandwidth on the different channels when profiling bodytrack.

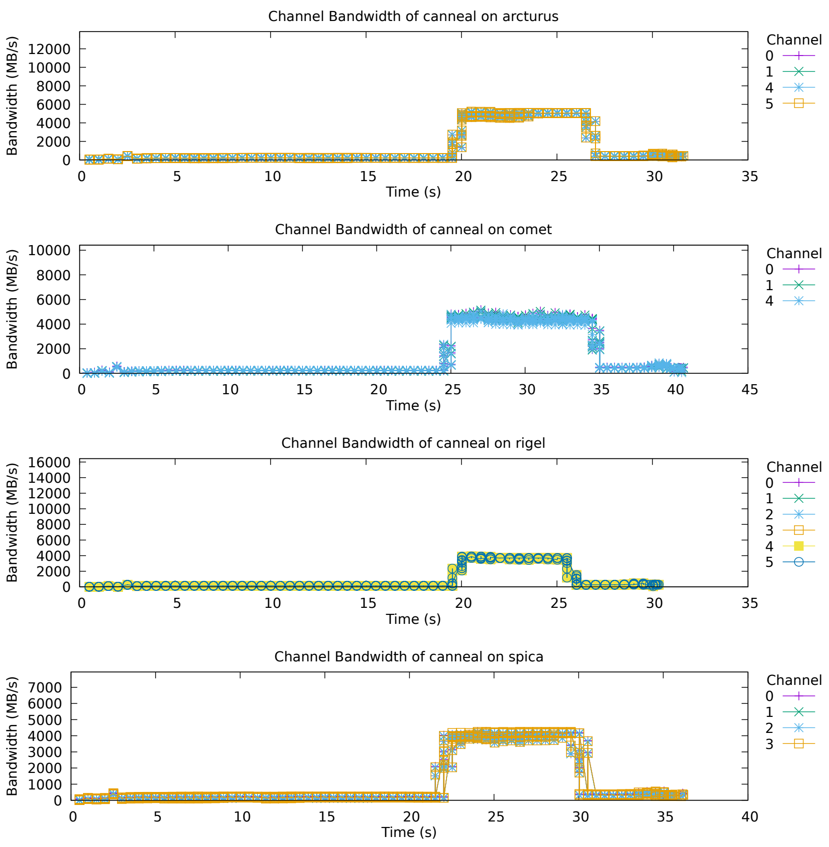


Figure 7.44. The bandwidth on the different channels when profiling canneal.

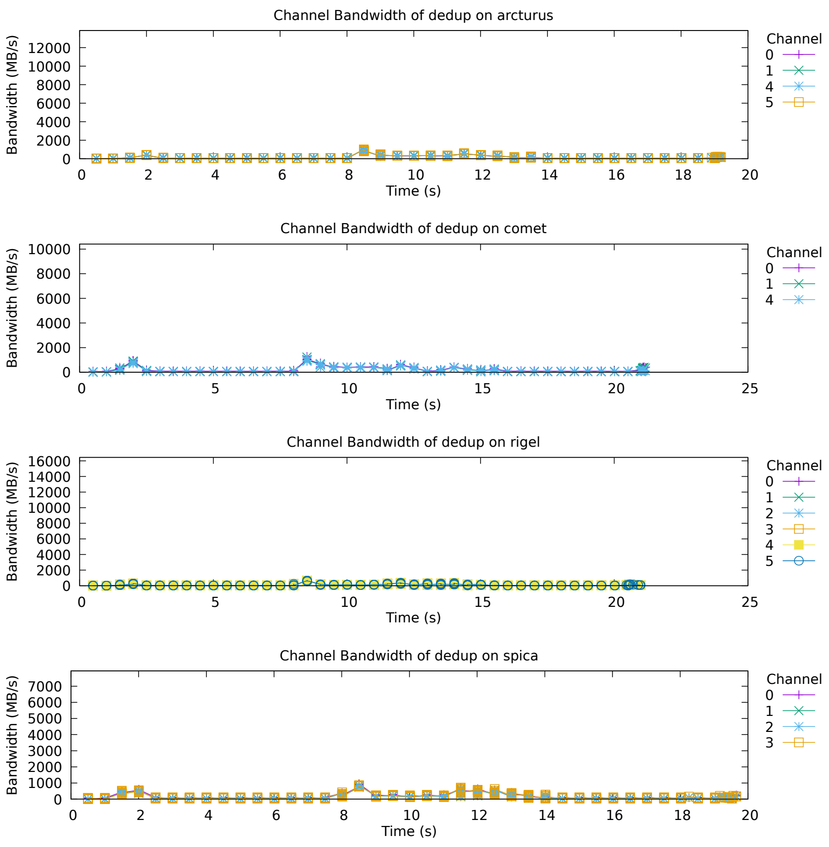


Figure 7.45. The bandwidth on the different channels when profiling dedup.

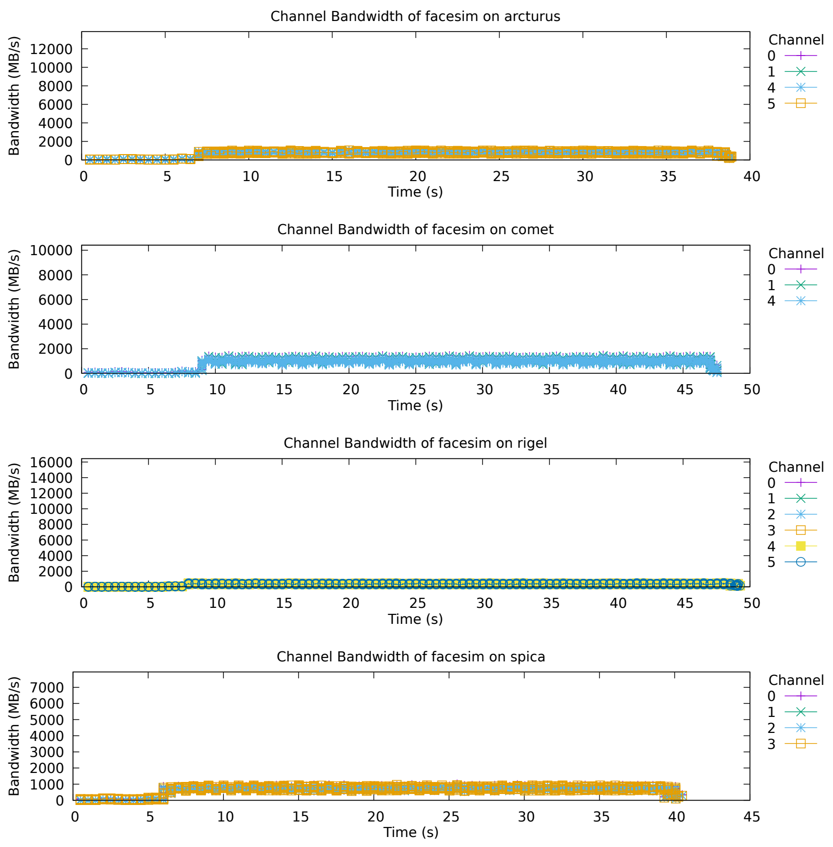


Figure 7.46. The bandwidth on the different channels when profiling facesim.

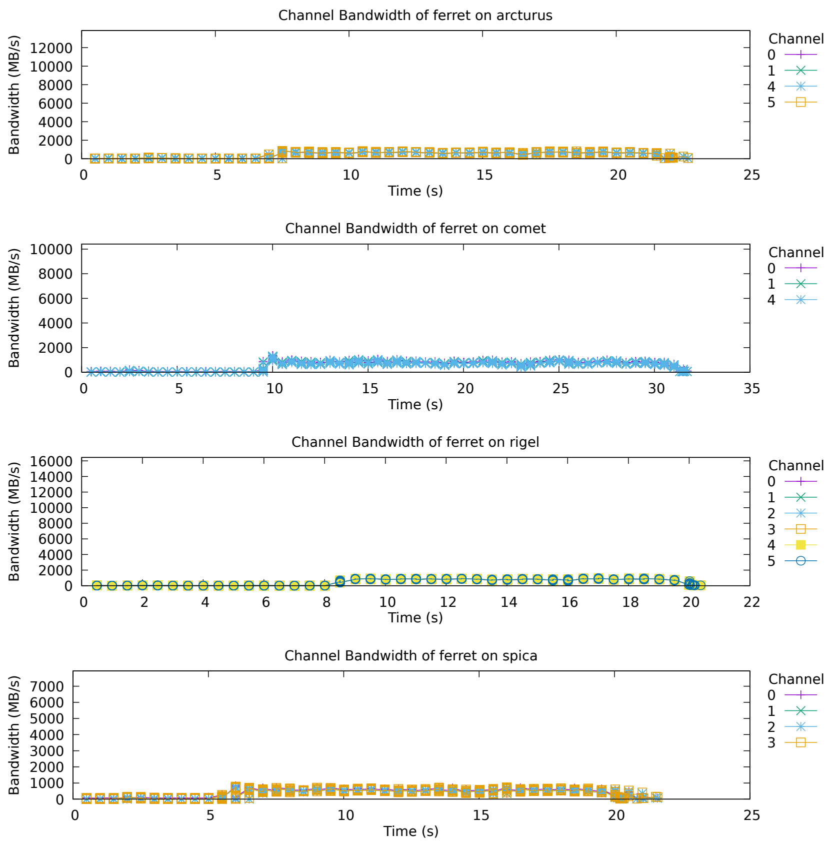


Figure 7.47. The bandwidth on the different channels when profiling ferret.

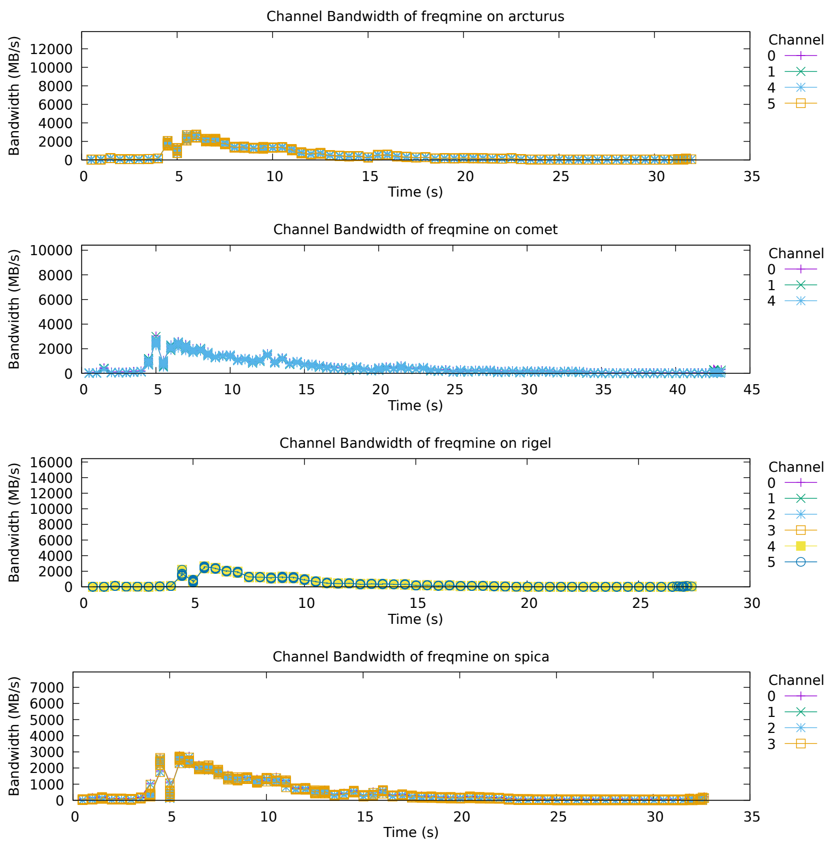


Figure 7.48. The bandwidth on the different channels when profiling freqmine.

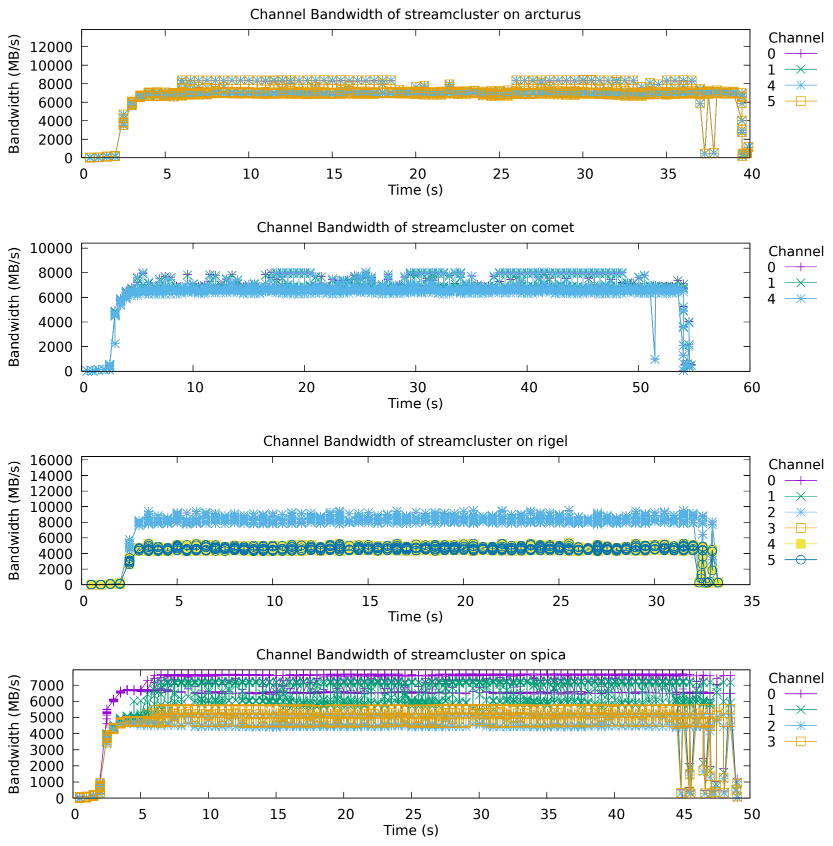


Figure 7.49. The bandwidth on the different channels when profiling streamcluster.

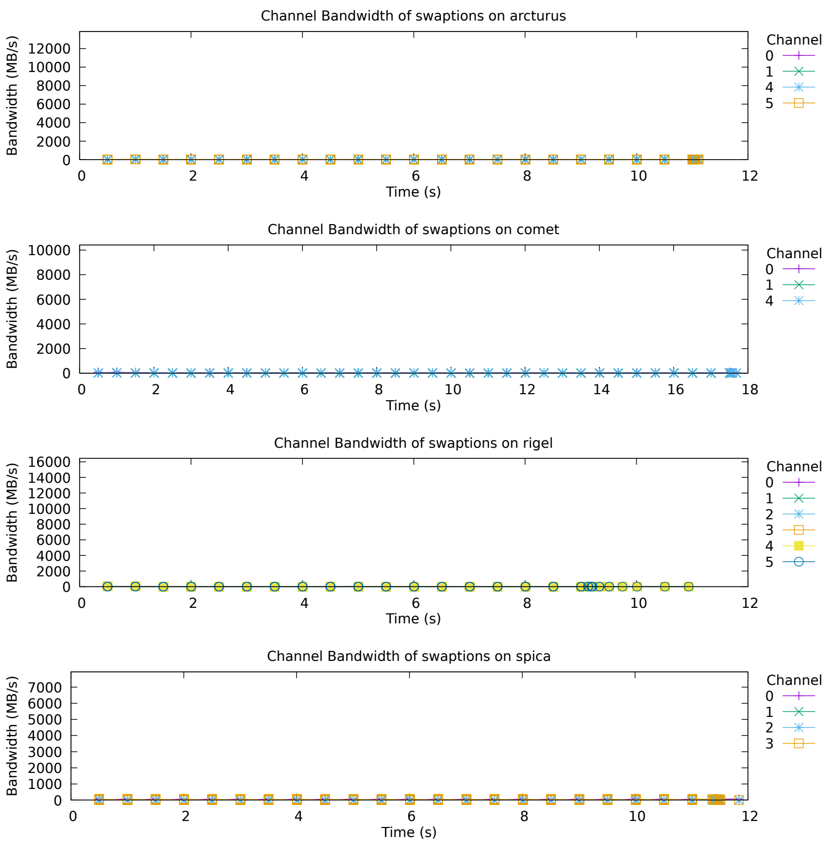


Figure 7.50. The bandwidth on the different channels when profiling swaptions.

7.6 Row Buffer Miss

We evaluate the row buffer miss detection using micro-benchmarks, the small kernels, and the PARSEC benchmarks. With the micro-benchmarks, we verify that the measurement is correct. In the small kernels, we find cases of low row buffer hit rate and introduce a performance optimization. The PARSEC benchmarks have good row buffer hit rates. For the row buffer hit rate the development with a changing number of threads is interesting because we expect a higher concflict ratio with more threads. Thus we use the per-thread display instead of a time-resolved graph in these evaluations.

7.6.1 Micro-Benchmarks

For a basic correctness check, we evaluate the measurement approach with a micro-benchmark. Our experiments confirm that the page empty, page conflict, and page hit ratio can be measured using the method introduces in Section 4.6.

Our evaluation is based on two known facts. First, in open page management, parallel memory accesses to different memory locations worsen the hit ratio and increase to conflict ratio. Multiple threads accessing different addresses cause many accesses to different rows in a short time. Second, when more banks are available the conflict rate should decrease and the hit rate should increase. Effectively, more row buffers are available for caching open rows. Thus, the micro-benchmark's memory accesses are done by a configurable number of threads and are limited to a configurable number of banks. It is based on the access benchmark used for the evaluation of the channel imbalance measurement. Due to hardware limitations, the experiments were only executed on Arcturus.

In Figure 7.51 four diagrams are shown. The different diagrams represent cases with a different number of banks used. In all of the four diagrams, we see an increase in the conflict ratio and a decrease in the hit ratio. By using more banks, the effect of the worsening hit ratio can be reduced.

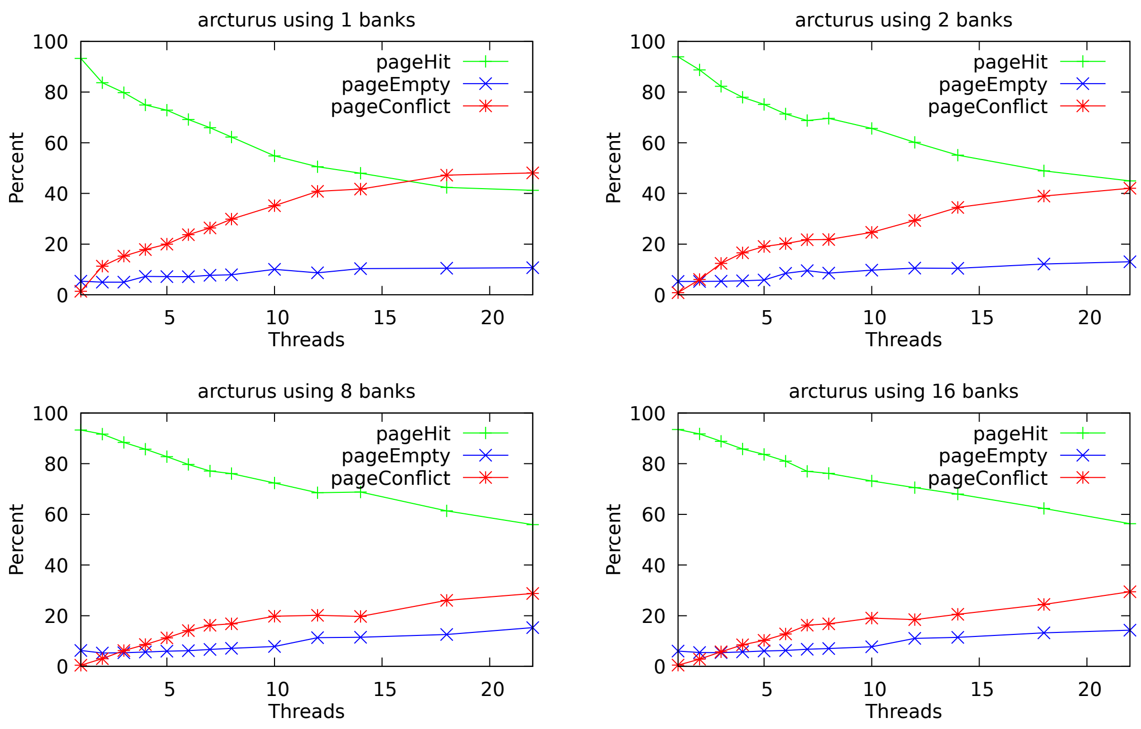


Figure 7.51. The row buffer hit and miss ratios over the number of threads. The four diagrams show the situation when access is limited to 1, 2, 8, or 16 banks.

The effect of the number of banks can be seen more clearly in Figure 7.52. When using one thread, as shown in the left diagram, the number of banks does not influence the hit ratio. In contrast, when using 22 threads, the number of banks has a significant influence on the hit rate.

Figure 7.53 shows the speedup obtained when increasing the number of threads. The number of available banks limits the speedup.

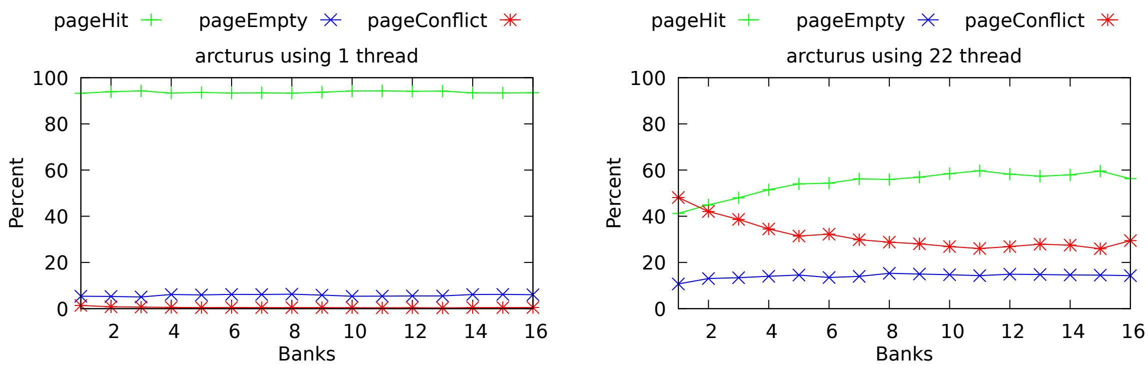


Figure 7.52. The row buffer hit and miss ratios over the number of used banks. The two diagrams show the situation when accesses come from 1 thread or 22 threads.

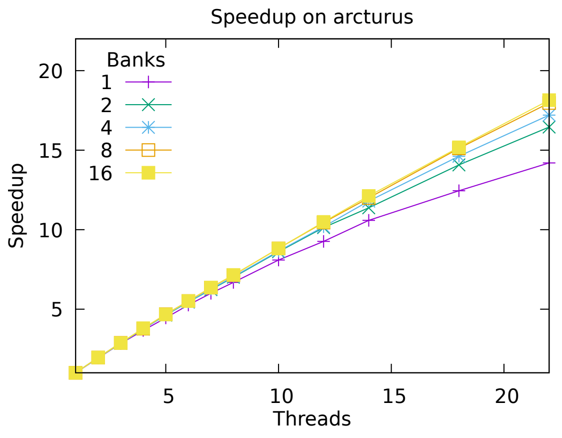


Figure 7.53. The parallel speedup for different number of used banks.

7.6.2 Small Kernels

For the small kernels, we can observe a similar phenomenon as for the micro-benchmarks. As shown in Figure 7.54, the conflict ratio increase with the number of threads while the hit ratio decreases. The copy benchmark from Stream is different from the other benchmarks. It has higher hit ratios, lower conflicts, and lower page empty ratios. The matrix multiplication benchmark which is not bandwidth bound has a high page empty ratio.

The difference in row buffer hit ratio between copy and scale is surprising because the source code of both benchmarks is very similar. Not just the row buffer hit rate, but also the performance between scale and copy is different as shown in Figure 7.55. Triad and add are expected to be slower because they access more data. GCC recognizes the memory copy pattern and uses a special internal memcopy function. Thus, the resulting assembly code is different. The source code and generated assembly code of scale are shown in Figure 7.56. The code of copy is shown in Figure 7.57. The compiler used is GCC 7.5. The compiler flags are -fopenmp and -O3 for both benchmarks. In the assembly code of copy in Figure 7.57b we can see four optimizations. First, the use of 256bit instructions instead of 128bit instructions. Second, the use of prefetch instructions. Third, the use of four moves in one loop iterations. Fourth, the use of streaming stores instead of conventional store instructions.

Figure 7.58 shows the execution time when applying different optimizations to Stream scale. It compares non-temporal stores (nt) and the vector instructions size (128 bit and 256 bit). Mostly the non-temporal stores are responsible for the improved performance. When using non-temporal stores, the performance of scale is very close to the performance of copy. Also, the row buffer hit rate is improved when using streaming stores as shown in Figure 7.59.

The same non-temporal store optimization can also be applied to Stream add and Stream triad. Figure 7.60 shows the improved performance and Figure 7.61 shows the improved row buffer hit rate.

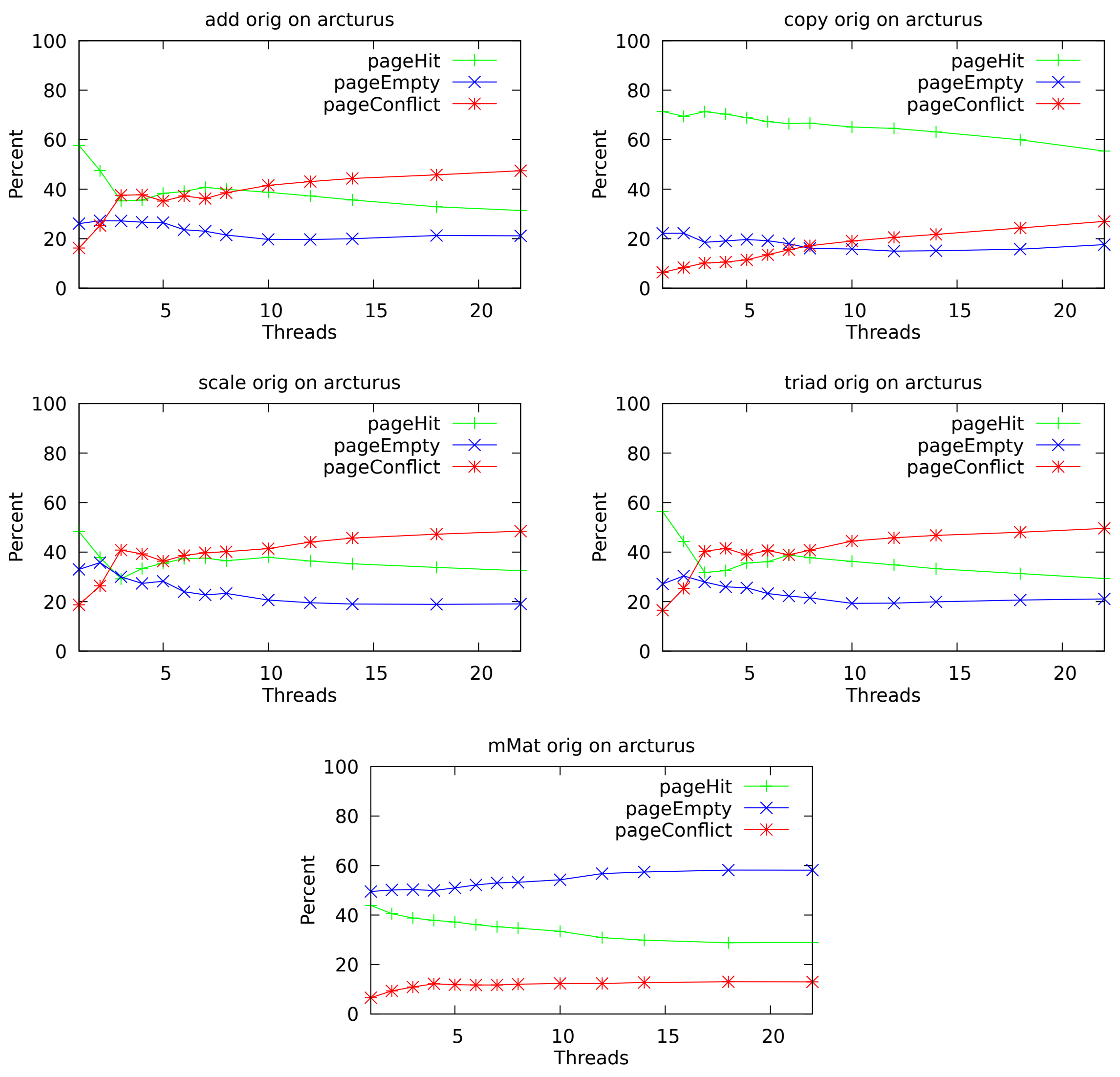


Figure 7.54. The row buffer status over the number of threads in the Stream benchmarks and the matrix multiplication benchmark.

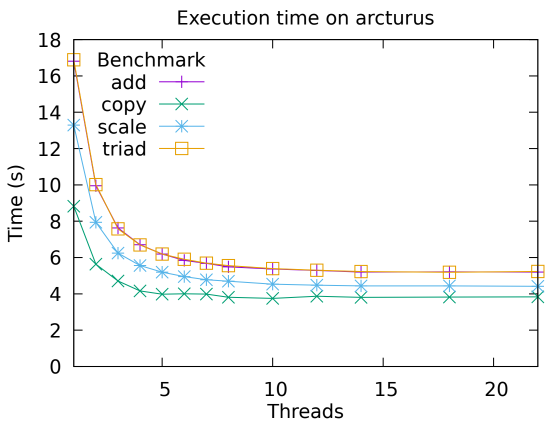


Figure 7.55. The execution time of the Stream benchmarks.

```
#pragma omp parallel for
  for (j=0; j<STREAM_ARRAY_SIZE; j++)
    b[j] = scalar*c[j];
#endif
```

(a) Original source code.

```
movapd (%r10,%rcx)
mulpd %xmm1, %xmm0
movups %xmm0, (%rdi,%rcx)
```

(b) Assembly generated by GCC.

Figure 7.56. Code of Stream scale

```
#pragma omp parallel for
  for (j=0; j<STREAM_ARRAY_SIZE; j++)
    b[j] = c[j];
#endif
```

(a) Original source code.

```
prefetcht0 0x100(%rsi)
prefetcht0 0x140(%rsi)
prefetcht0 0x180(%rsi)
prefetcht0 0x1c0(%rsi)
vmovdqu (%rsi),%ymm0
vmovdqu 0x20(%rsi),%ymm1
vmovdqu 0x40(%rsi),%ymm2
vmovdqu 0x60(%rsi),%ymm3
vmovntdq %ymm0,(%rdi)
vmovntdq %ymm1,0x20(%rdi)
vmovntdq %ymm2,0x40(%rdi)
vmovntdq %ymm3,0x60(%rdi)
```

(b) Assembly generated by GCC.

Figure 7.57. Code of Stream copy

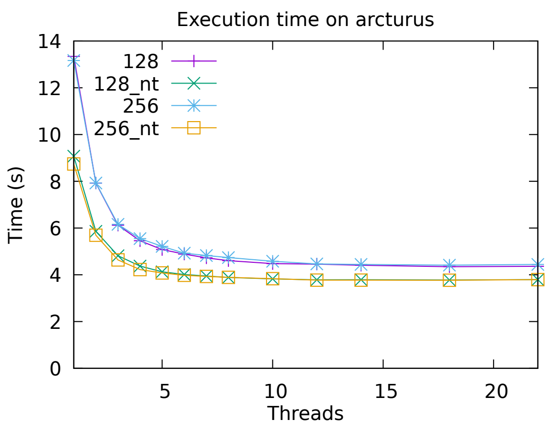


Figure 7.58. The execution time of differnt optimizaions of Stream scale.

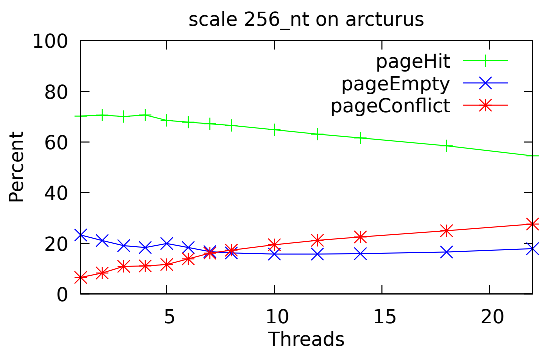


Figure 7.59. Row buffer hit rate of Stream scale with non-temporal stores optimizaion.

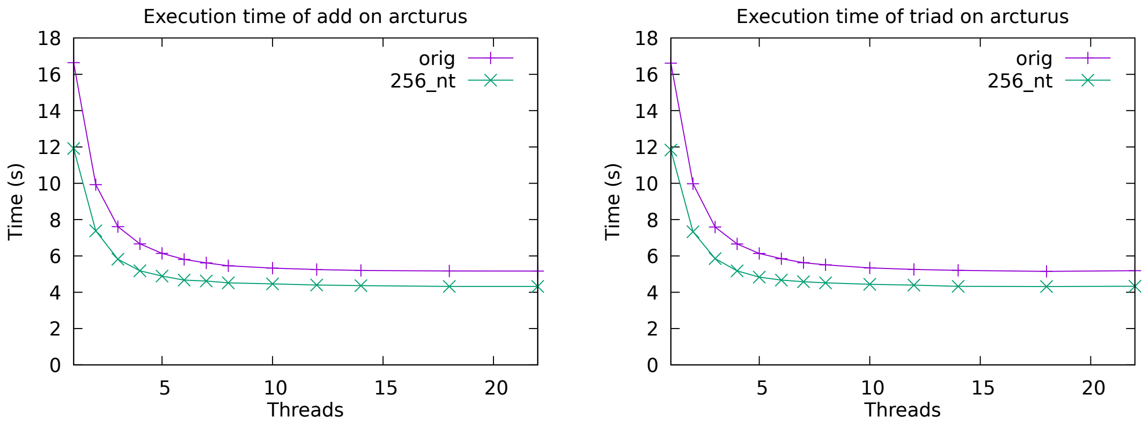


Figure 7.60. The execution time of the original version and the optimized version of Stream triad and Stream add.

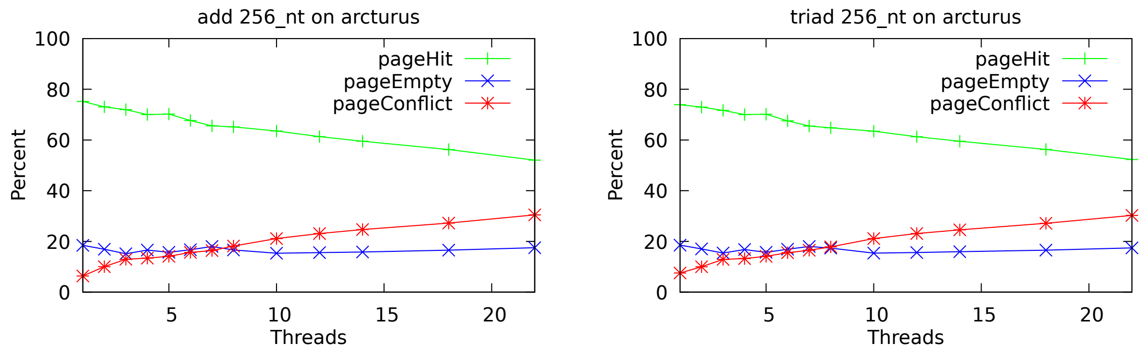


Figure 7.61. The row buffer hit status of the optimized Stream triad and Stream scale.

7.6.3 PARSEC Benchmarks

Figure 7.62 to Figure 7.70 show the results of the PARSEC benchmarks. In all of the PARESC benchmarks, the conflict ratio is very low, usually below 15%. Generally the hit ratio is higher, it is higher than the percentage of page empty. They perform better than the kernel benchmarks. Streamcluster (Figure 7.69) shows worsening hit ratio when increasing the number of threads. In contrast, the other benchmarks have relatively stable ratios when changing the number of threads.

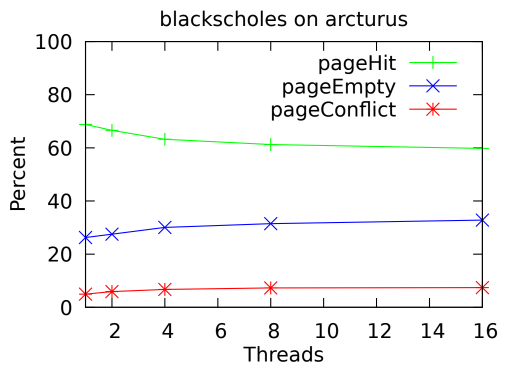


Figure 7.62. The row buffer status over the number of threads of blackscholes.

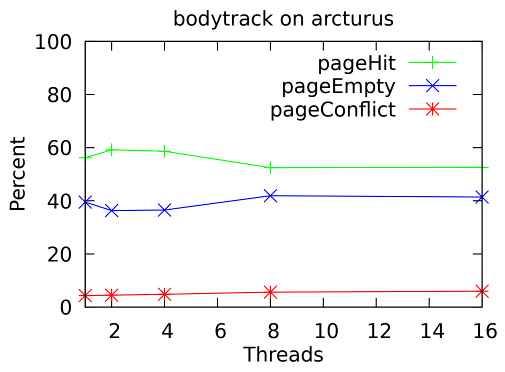


Figure 7.63. The row buffer status over the number of threads of bodytrack.

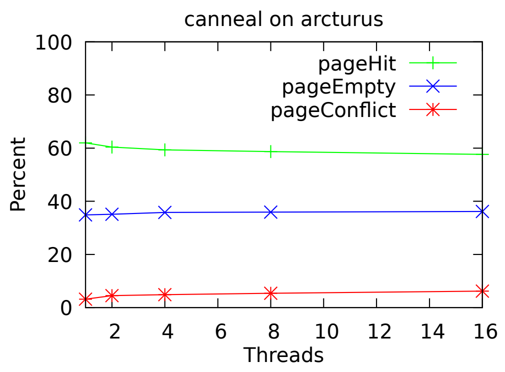


Figure 7.64. The row buffer status over the number of threads of canneal.

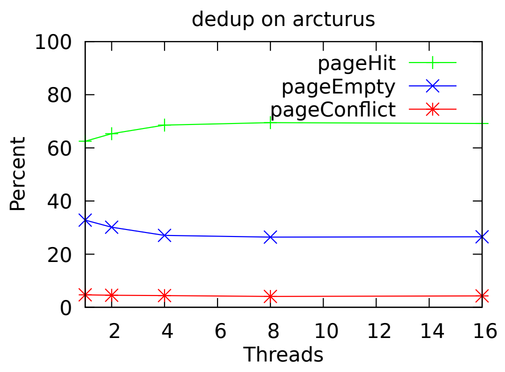


Figure 7.65. The row buffer status over the number of threads of dedup.

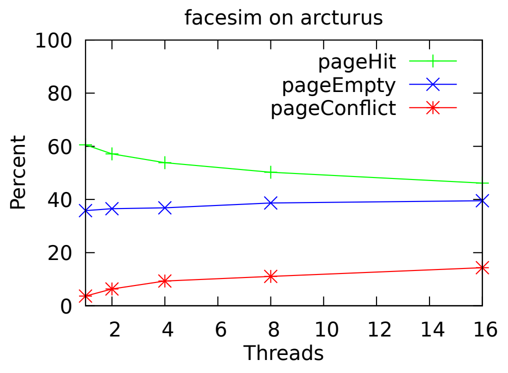


Figure 7.66. The row buffer status over the number of threads of facesim.

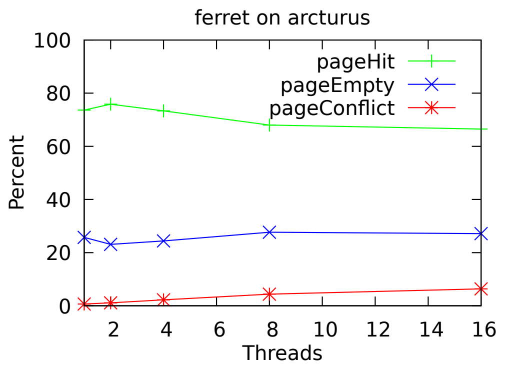


Figure 7.67. The row buffer status over the number of threads of ferret.

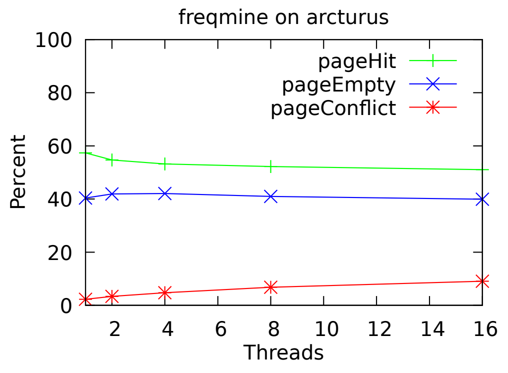


Figure 7.68. The row buffer status over the number of threads of freqmine.

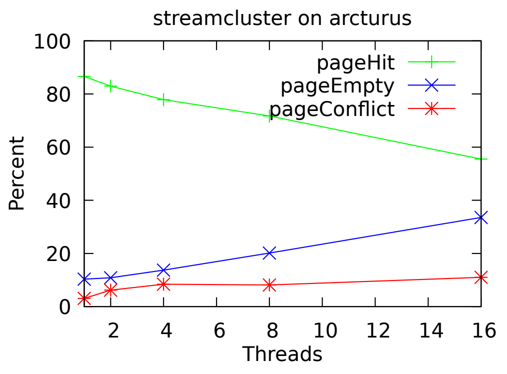


Figure 7.69. The row buffer status over the number of threads of streamcluster.

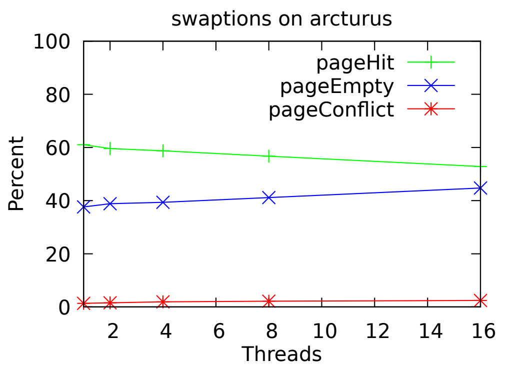


Figure 7.70. The row buffer status over the number of threads of swaptions.

7.7 Profiling Overhead

Profiling needs to be done only once, no matter what performance problem is analyzed. We first discuss the average overhead of PerfMemPlus and then go into the details for individual systems and benchmarks.

The overhead of PerfMemPlus profiling varies with the sampling period. As shown in Figure 7.71, it ranges between 3% and 9% on average. In this figure, the average overall benchmarks and all systems is reported. In some cases, there is a negative overhead when profiling. In those cases, the overhead is counted as zero overhead in this calculation of the average. In Figure 7.71, the overhead increases from a sampling period of 32000 to 64000. This happens because we did not profile the artificial DRAM contention benchmarks with a sampling period higher than 32000. Those benchmarks have very low overhead, thus when they are not included in the average overhead, the average overhead rises. The highest overhead observed occurs on Spica when profiling dedup with 48 threads. The overhead, in this case, is 160%.

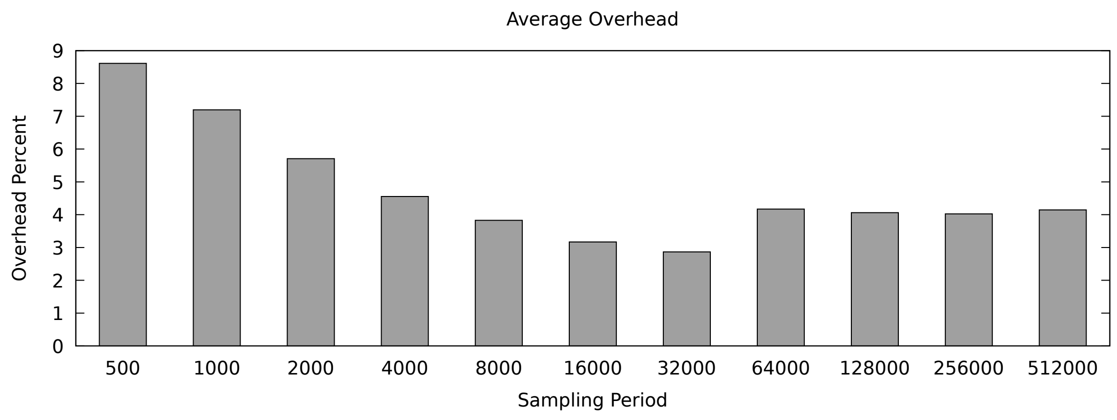


Figure 7.71. The average overhead of PerfMemPlus profiling across all systems and all benchmarks.

7.7.1 Artificial False Sharing Benchmarks

The average overhead of profiling the artificial false sharing benchmarks is shown in Figure 7.72. Some of the artificial micro-benchmarks show high variations in the execution time, especially in the original versions that suffer from false sharing. We do not use the results from those benchmarks for the evaluation of the overhead. Instead, we provide overhead results from profiling the fixed version of those benchmarks. All diagrams are drawn with the same scale of up to 35% overhead.

In boost_spinlock (Figure 7.73), the maximum overhead is 17%. It drops with higher sampling periods. On Spica, there is a negative overhead in some cases. The overhead of the locked benchmark is at most 24%, as can be seen in Figure 7.74. It quickly drops to low levels at sampling periods of 8000 or higher. A similar maximum overhead occurs in the lockless benchmark, as shown in Figure 7.75. The overhead in ref_count (Figure 7.76) is at most 12%. There is no fixed version of the true sharing benchmark. The overhead of profiling the true sharing benchmark is shown in Figure 7.77.

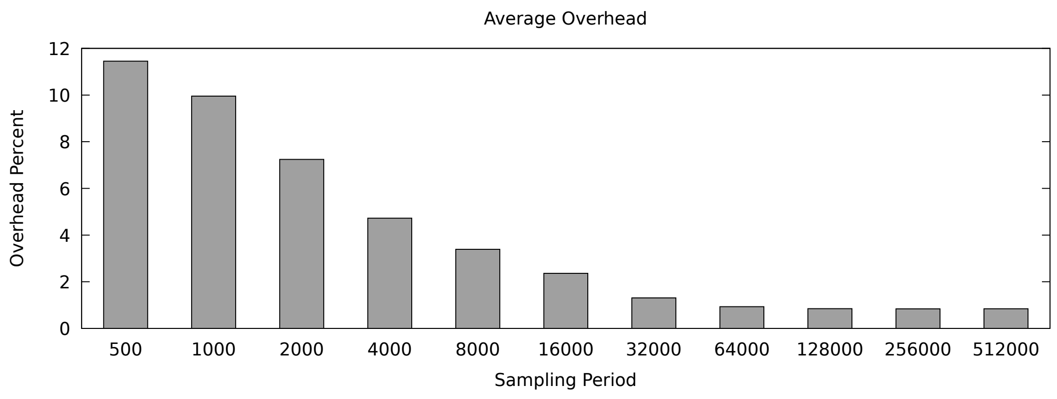


Figure 7.72. The average overhead of profiling the artificial false sharing benchmarks.

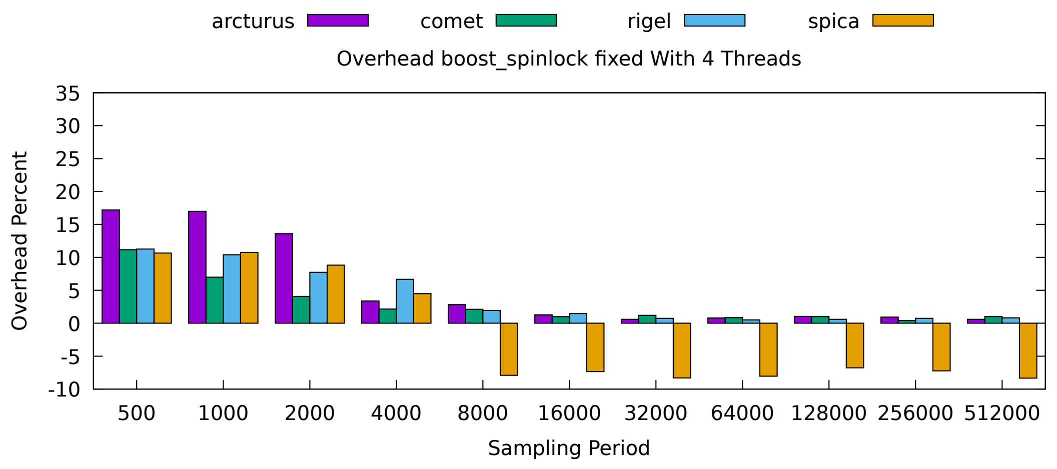


Figure 7.73. The overhead of profiling boost spinlock.

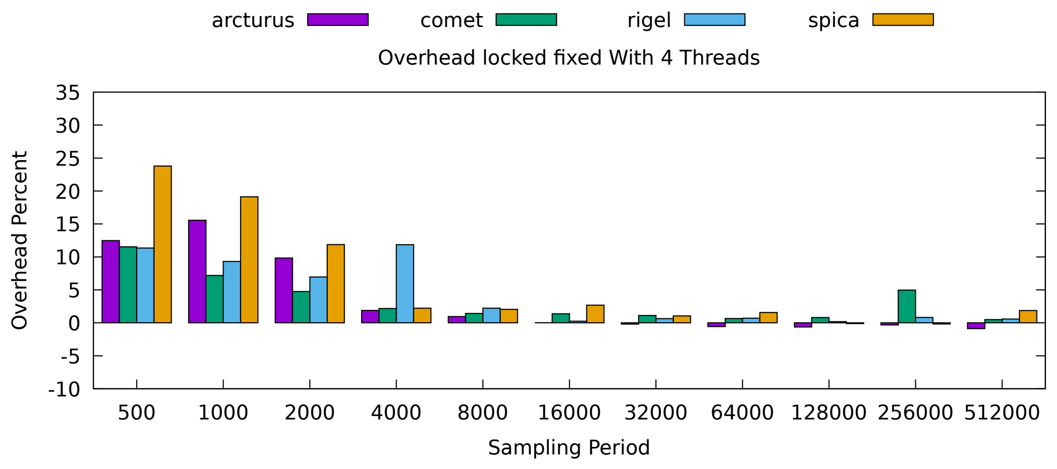


Figure 7.74. The overhead of profiling locked.

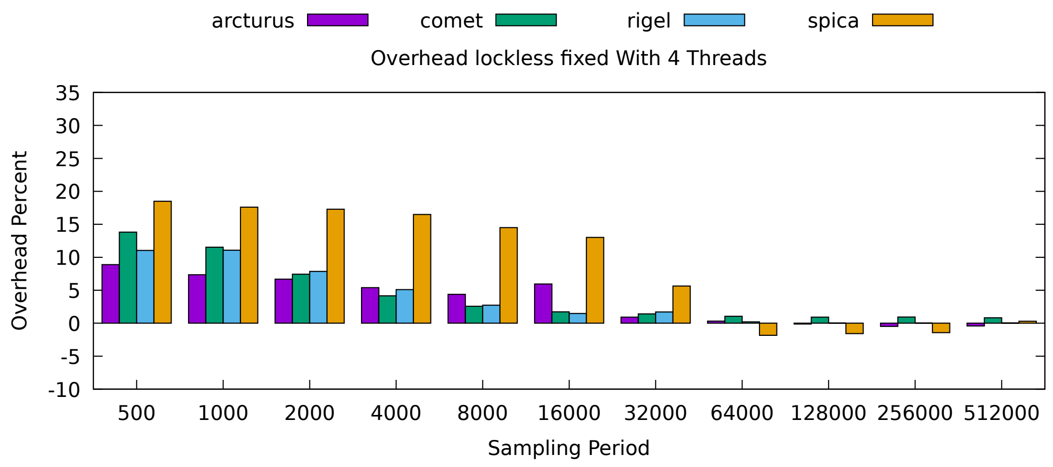


Figure 7.75. The overhead of profiling lockless.

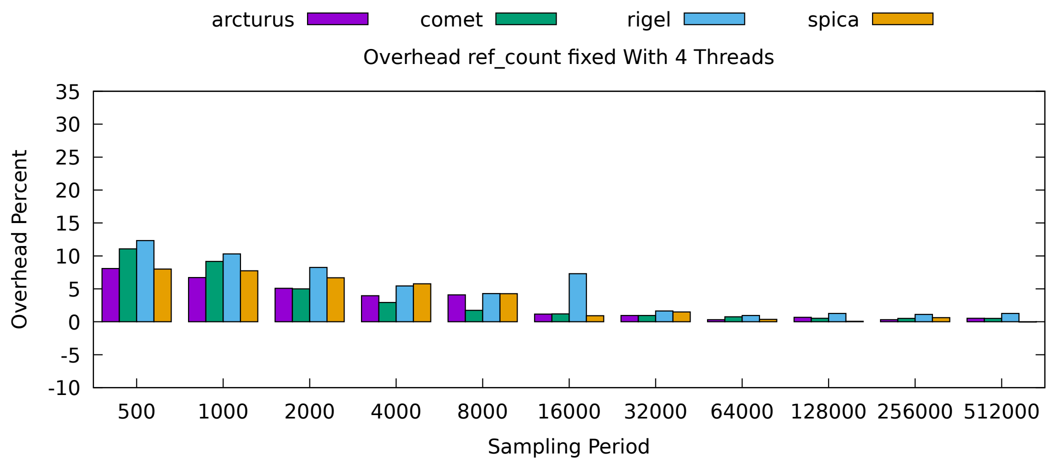


Figure 7.76. The overhead of profiling ref count.

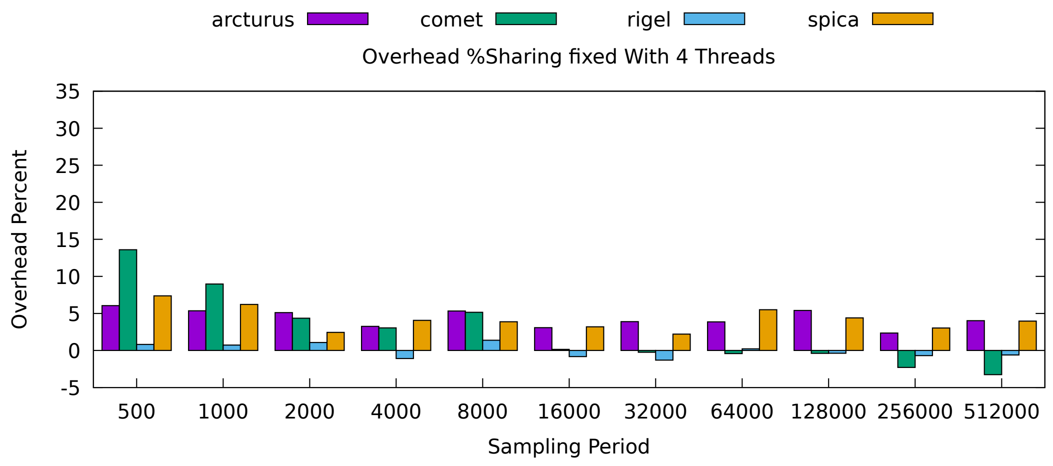


Figure 7.77. The overhead of profiling true sharing.

7.7.2 Phoenix Benchmarks

In the Phoenix benchmarks, we also saw unstable execution times in the original versions. Thus, we report the speedup for the fixed versions. The average overhead of profiling the Phoenix benchmarks is shown in Figure 7.78.

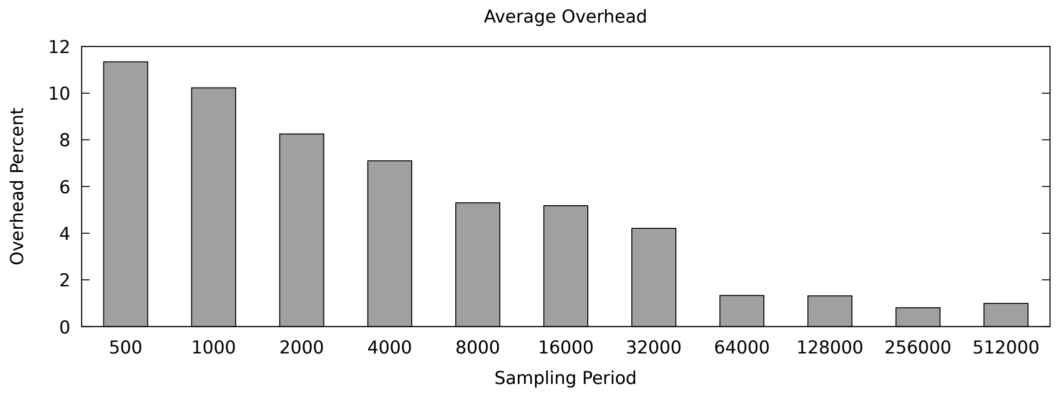


Figure 7.78. The average overhead of profiling the Phoenix benchmarks.

Figure 7.79 shows the overhead when profiling the linear regression benchmark. On Spica, the overhead is much higher than on the other systems. Histogram has a high variation of the execution time in the if Figure 7.80 shows the overhead that occurs when profiling histogram. The profiling overhead in histogramFs is displayed in Figure 7.81. There is a speedup when profiling with low sampling periods on all systems.

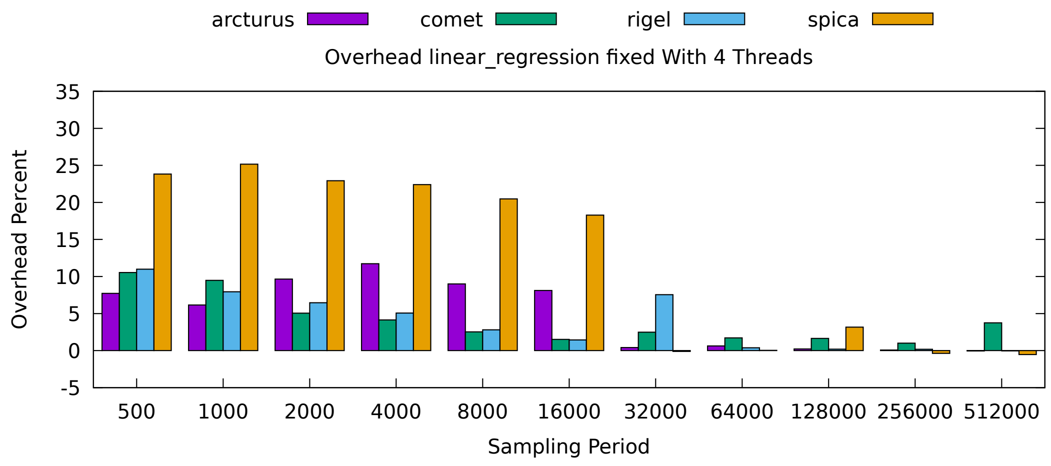


Figure 7.79. The overhead of profiling linear regression.

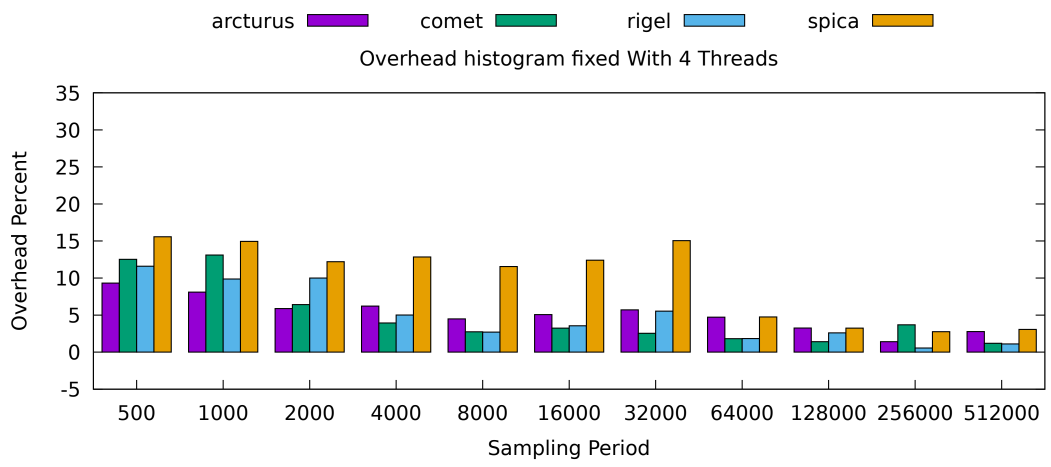


Figure 7.80. The overhead of profiling histogram (original input).

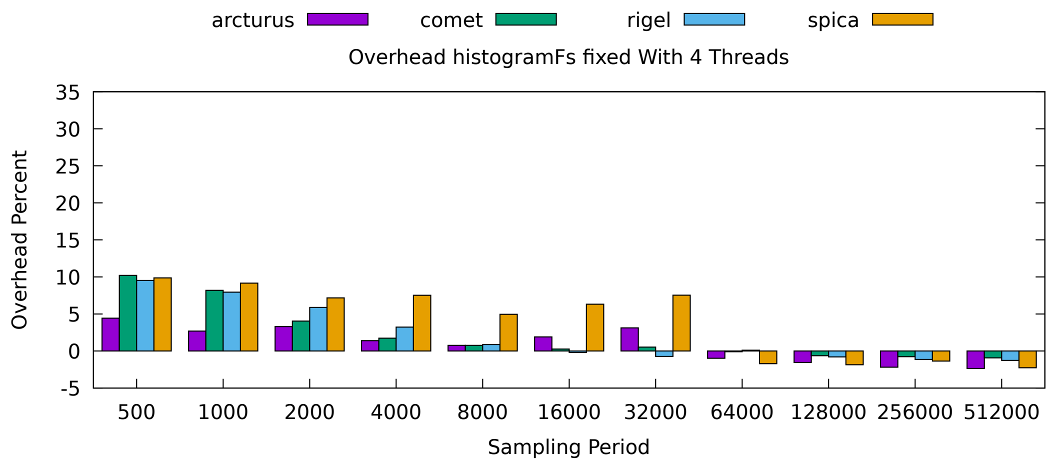


Figure 7.81. The overhead of profiling histogram (false sharing input).

7.7.3 Artificial DRAM Contention Benchmarks

The average overhead of profiling the artificial DRAM contention benchmarks is shown in Figure 7.82. With a higher number of threads, the overhead decreases.

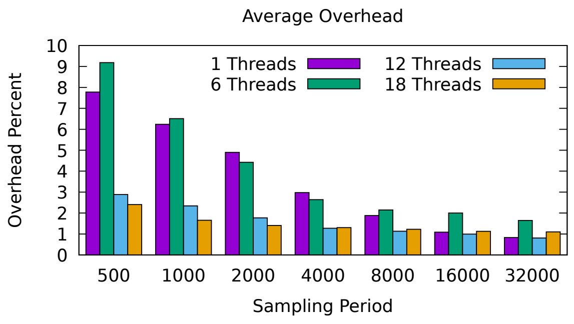


Figure 7.82. The average overhead of profiling the artificial DRAM contention detection benchmarks.

For the artificial benchmarks used for the DRAM contention detection evaluation, we show the overhead with 1 thread and 18 threads. We chose 18 threads because even the smallest system supports 18 threads and because we want to show the difference when using one thread or many threads. All diagrams show the same overhead range of -2% to +20%. In general, the overhead is lower with a higher thread count. On Comet, with a sampling period of 500, the overhead is much higher than on the other systems when profiling countv (Figure 7.83) and sumv (Figure 7.85). With one thread, the overhead drops quickly with higher sampling periods. When using 18 threads, there is only a small decrease in the overhead.

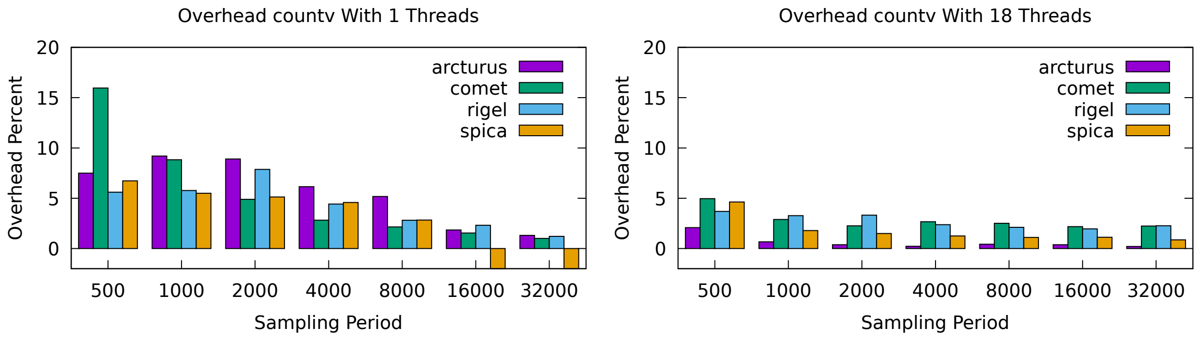


Figure 7.83. The overhead of profiling countv with one thread (left) and 18 threads (right).

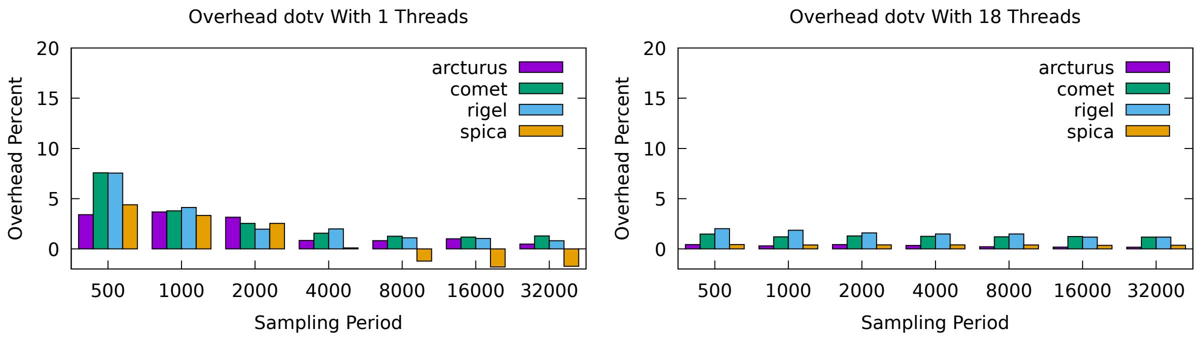


Figure 7.84. The overhead of profiling dotv with one thread (left) and 18 threads (right).

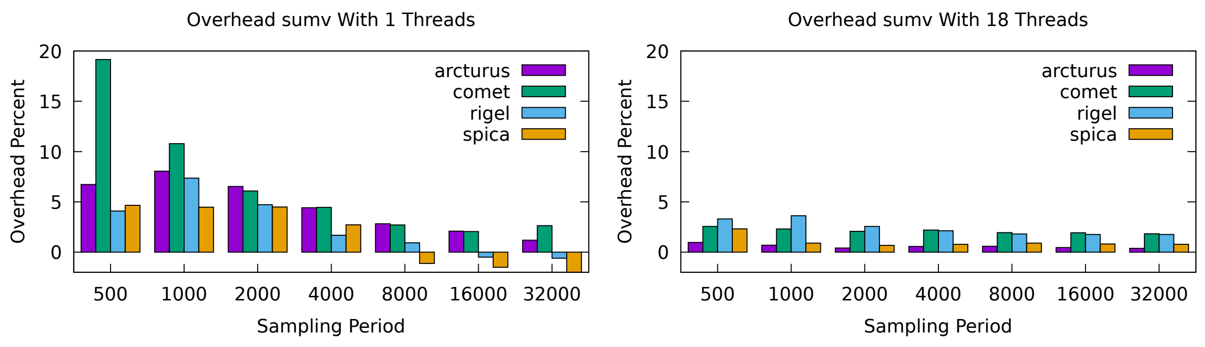


Figure 7.85. The overhead of profiling sumv with one thread (left) and 18 threads (right).

7.7.4 PARSEC Benchmarks

The average overhead of profiling the PARSEC benchmarks increased with the number of threads as shown in Figure 7.86.

The overhead of profiling the PARSEC benchmarks differs greatly depending on the benchmark. All diagrams show the same overhead range of -5% to 35%. We show the results when profiling the benchmarks using 4 threads and when the number of threads is equal to the number of physical cores of the system.

When using four threads, the overhead of profiling blackscholes is always below 10%, as shown in Figure 7.87. When using 56 threads on Rigel, the overhead rises to up to 35% but drops to below 3% when profiling with a sampling period of 4000 or higher (Figure 7.88). The overhead of profiling bodytrack is similar in both profiled thread count versions, as shown in Figure 7.89 and Figure 7.90. When profiling canneal, the overhead rises when profiling with more threads. It is shown in Figure 7.91 and Figure 7.92 respectively. The overhead of profiling dedup on Spica with four threads (Figure 7.93) is higher than on the other systems. In contrast to the overhead of profiling most of the other benchmarks, it does not decrease with higher sampling periods. Figure 7.94 shows that when increasing the number of threads, the profiling overhead increases. On Spica, we see an unusual high overhead of up to 160% that does not decrease with higher sampling periods. The profiling overhead of facesim is similar when using four threads (Figure 7.95) and when using 32 threads (Figure 7.96). The overhead decreases with higher sampling periods. On Spica, the overhead increases with the number of threads. The profiling overhead of ferret is very low. It never exceeds 10%, as shown in Figure 7.97. There is also a negative

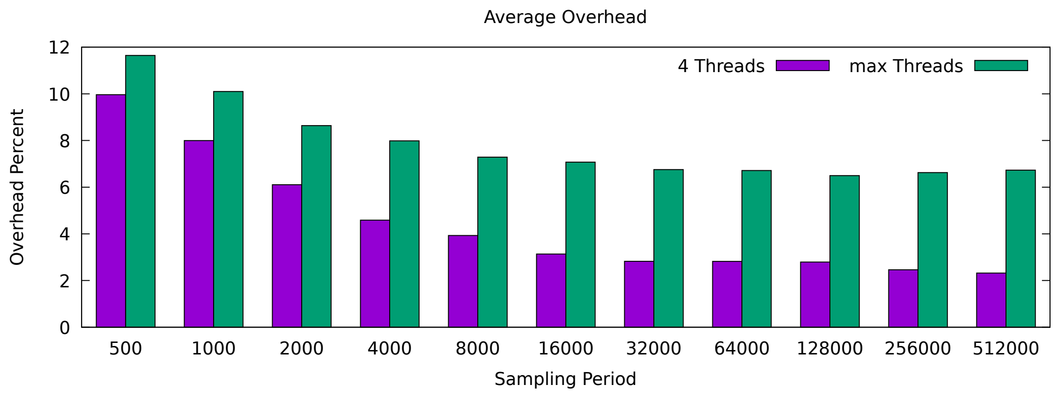


Figure 7.86. The average overhead of profiling the PARSEC benchmarks with four threads and with the maximum thread count.

overhead on Comet. With a higher number of threads, the overhead decreases further, as shown in Figure 7.98. Profiling ferret can be done with low overhead. Both with four threads and 32 threads, as shown in Figure 7.99 and Figure 7.100. The profiling overhead of freqmine is much lower with the maximum thread count than with four threads. It drops from up to 15%, as shown in Figure 7.101 to a maximum of 3%, as shown in Figure 7.102. A similar phenomenon is observed when profiling raytrace. The overhead decreases when using more threads. The detailed numbers are plotted in Figure 7.103 and Figure 7.104. The overhead of profiling streamcluster is generally low, as visualized in Figure 7.105 and Figure 7.106. But on Rigel, with 56 threads and a sampling period of 500, the overhead reaches 30%. The overhead of profiling swaptions is shown in Figure 7.107 and Figure 7.108. It increases when using more threads. Except for Spica, where it decreases. The profiling overhead of vips decreases with a higher number of threads. From up to 21%, as shown in Figure 7.109 to a maximum of 5%, as shown in Figure 7.110. The profiling overhead of x264 behaves similar to vips. It decreases from up to 15%, as shown in Figure 7.111 to a maximum of 5%, as shown in Figure 7.111.

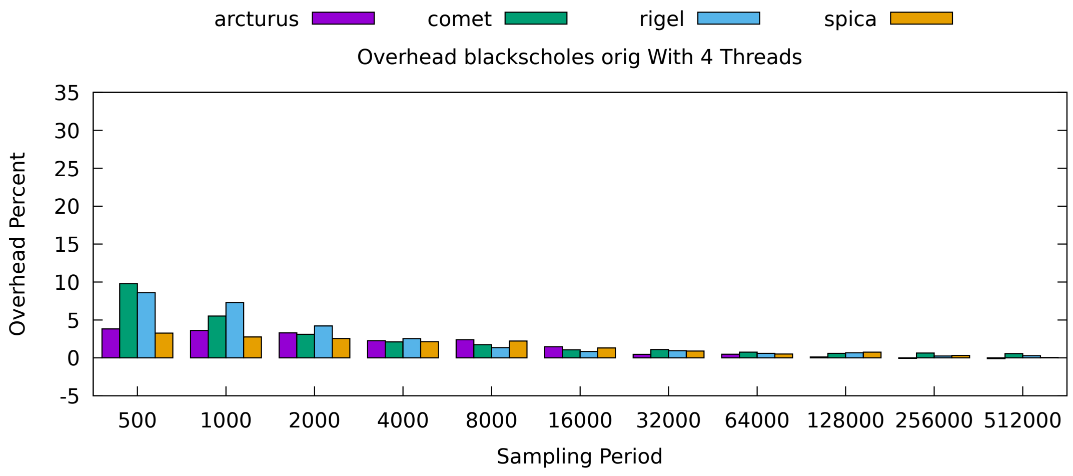


Figure 7.87. The overhead of profiling blacksholes with 4 threads.

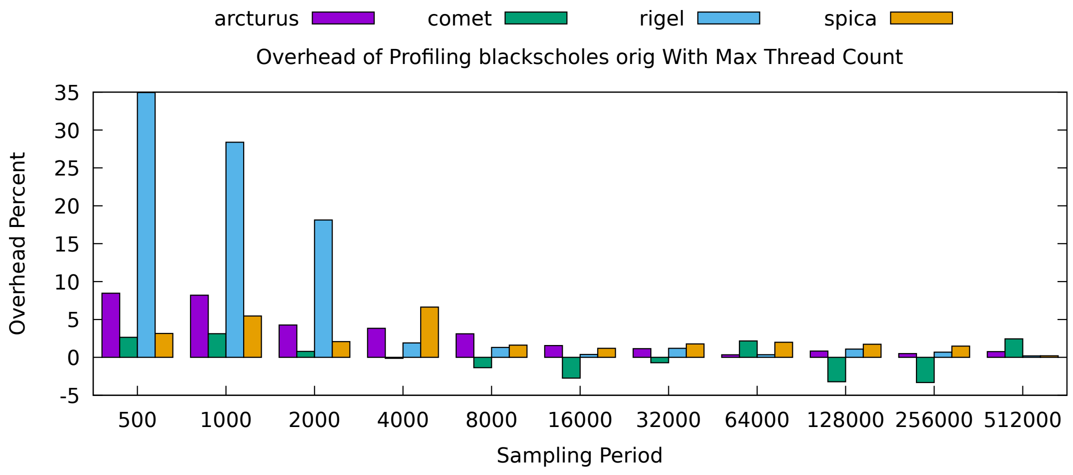


Figure 7.88. The overhead of profiling blacksholes with as much threads as physical cores available (Arcturus 44, Comet 36, Rigel 56, Spica 48).

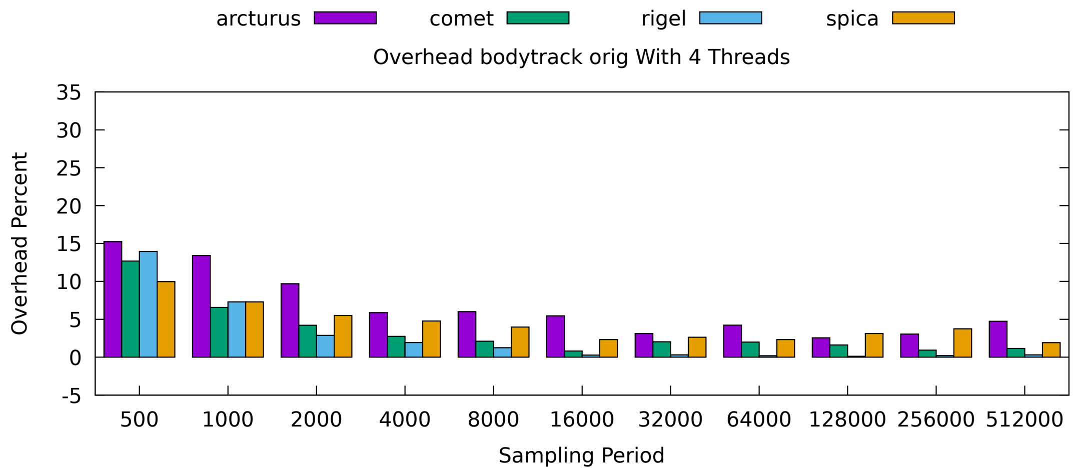


Figure 7.89. The overhead of profiling bodytrack with 4 threads.

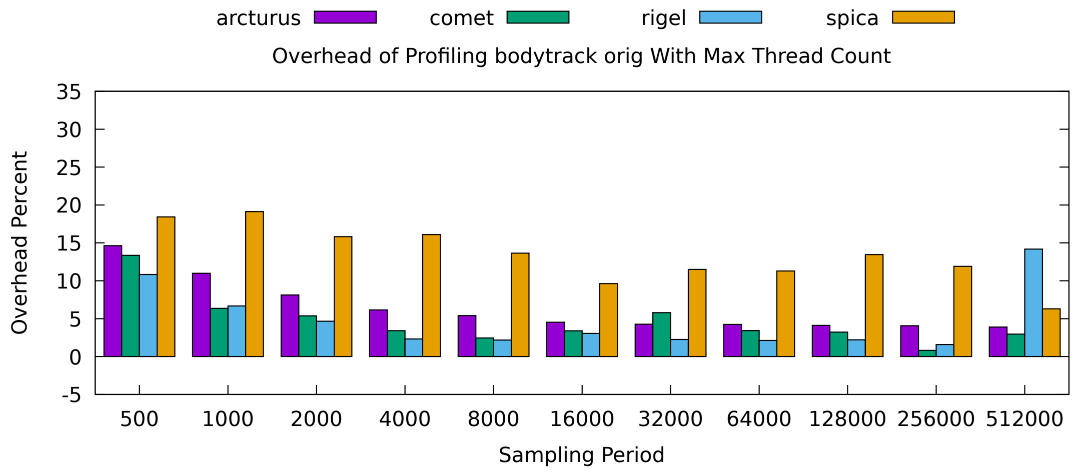


Figure 7.90. The overhead of profiling bodytrack with as much threads as physical cores available (Arcturus 44, Comet 36, Rigel 56, Spica 48).

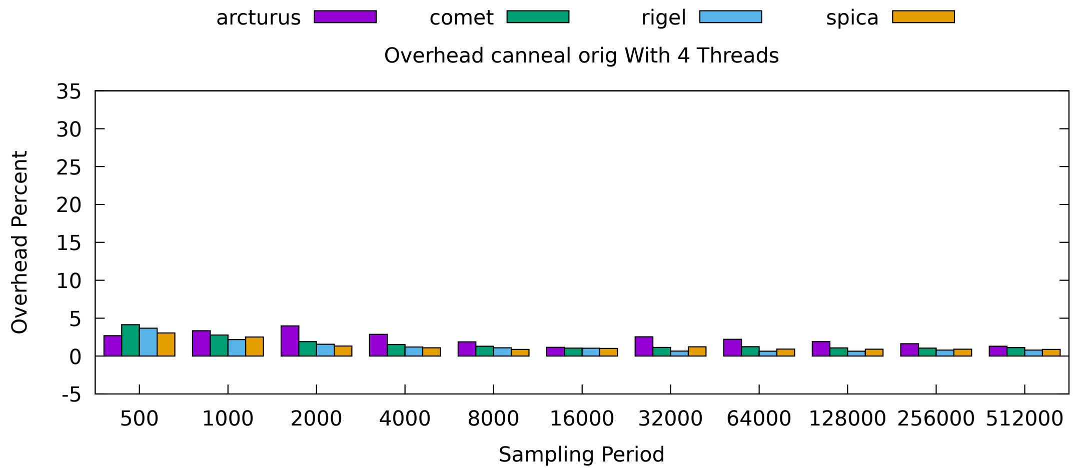


Figure 7.91. The overhead of profiling canneal with 4 threads.

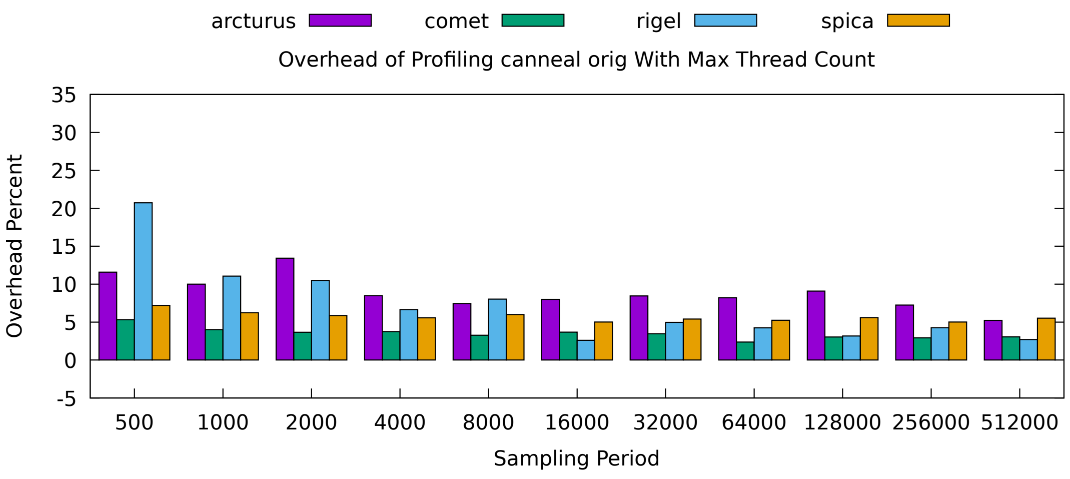


Figure 7.92. The overhead of profiling canneal with as much threads as physical cores available (Arcturus 44, Comet 36, Rigel 56, Spica 48).

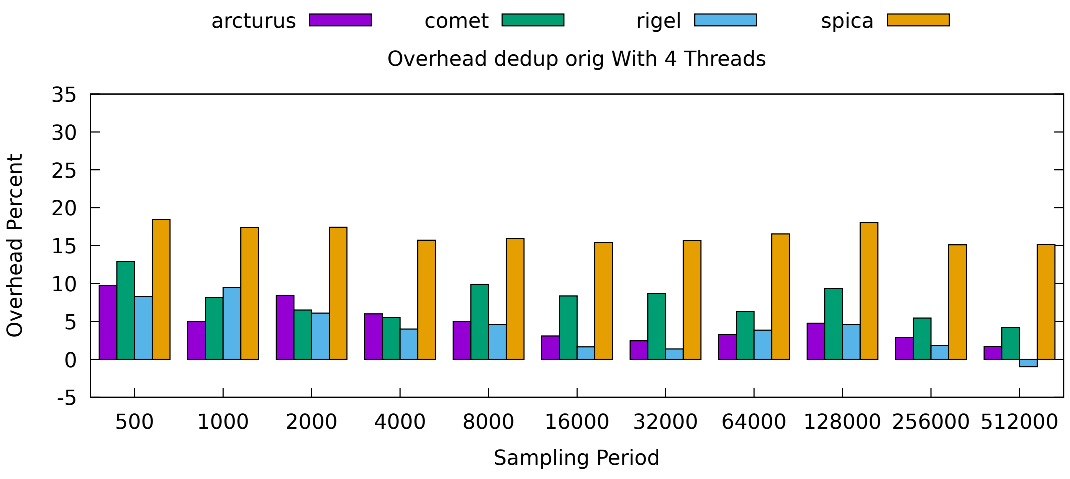


Figure 7.93. The overhead of profiling dedup with 4 threads.

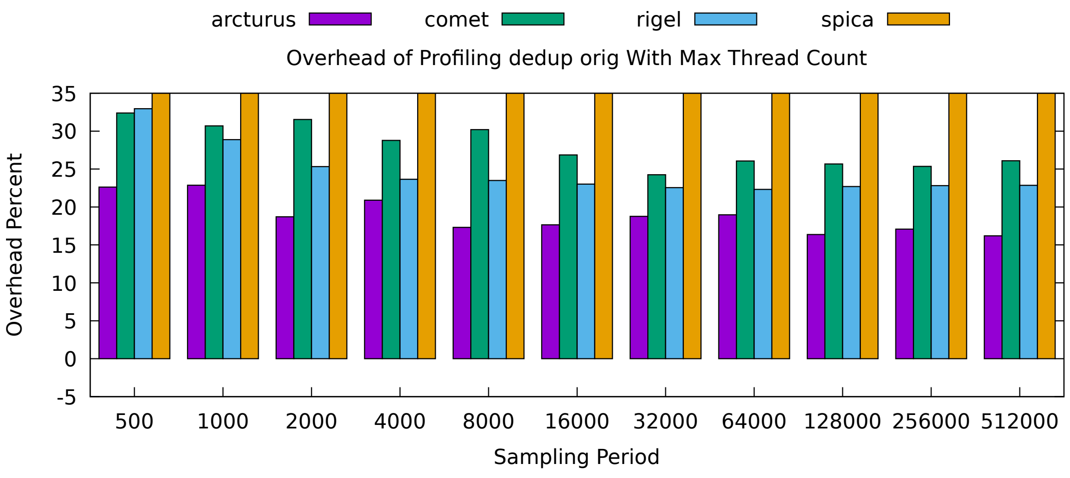


Figure 7.94. The overhead of profiling dedup with as much threads as physical cores available (Arcturus 44, Comet 36, Rigel 56, Spica 48). Because the overhead on Spica reaches 160% with all sampling rates the visualization is cut in this diagram.

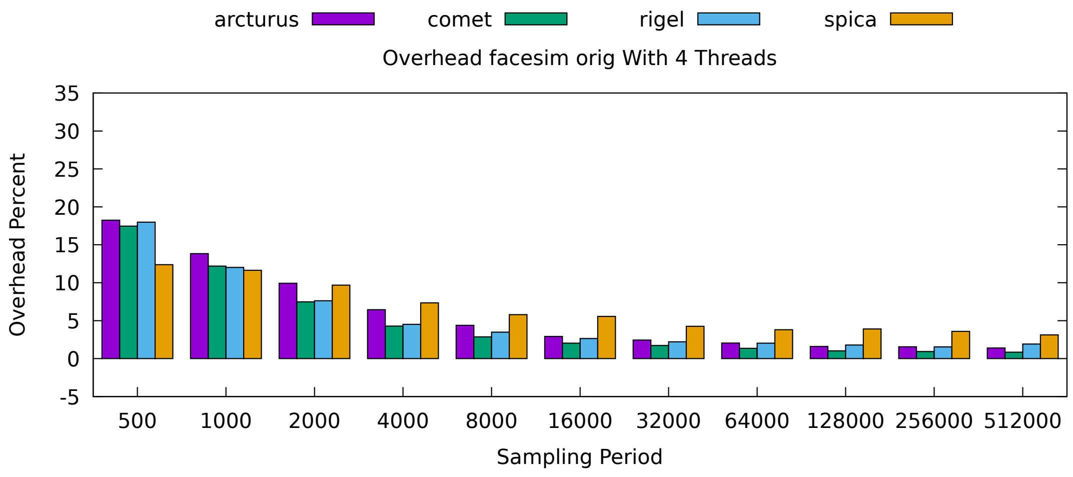


Figure 7.95. The overhead of profiling facesim with 4 threads.

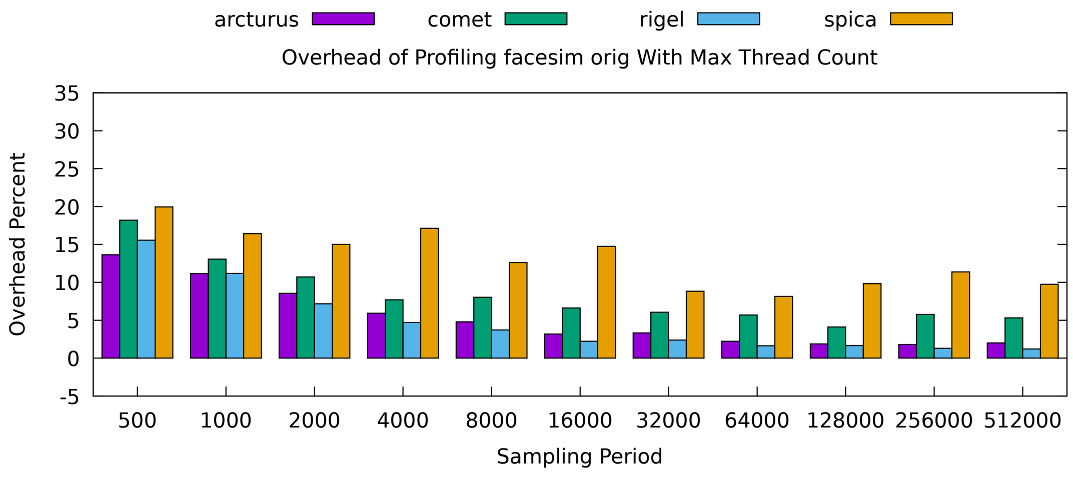


Figure 7.96. The overhead of profiling facesim with 32 threads.

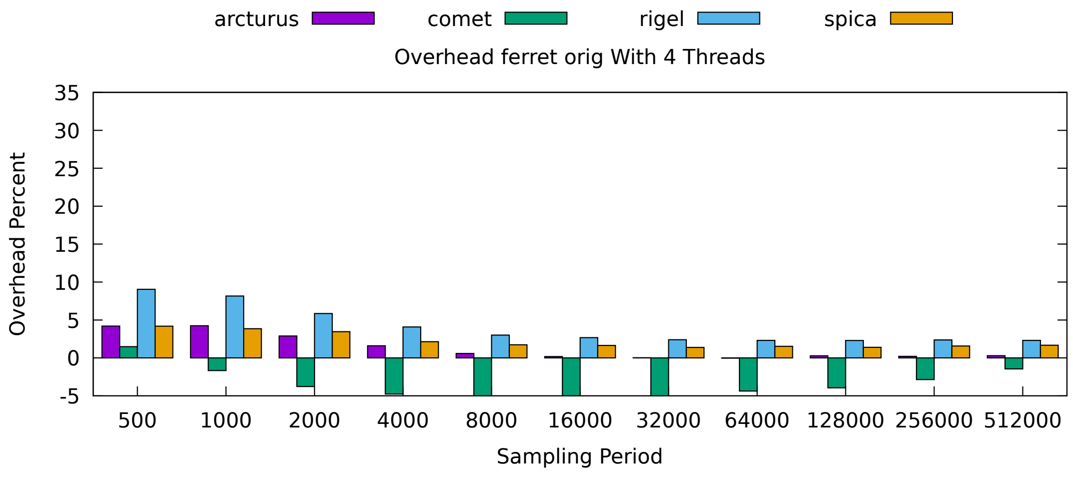


Figure 7.97. The overhead of profiling ferret with 4 threads.

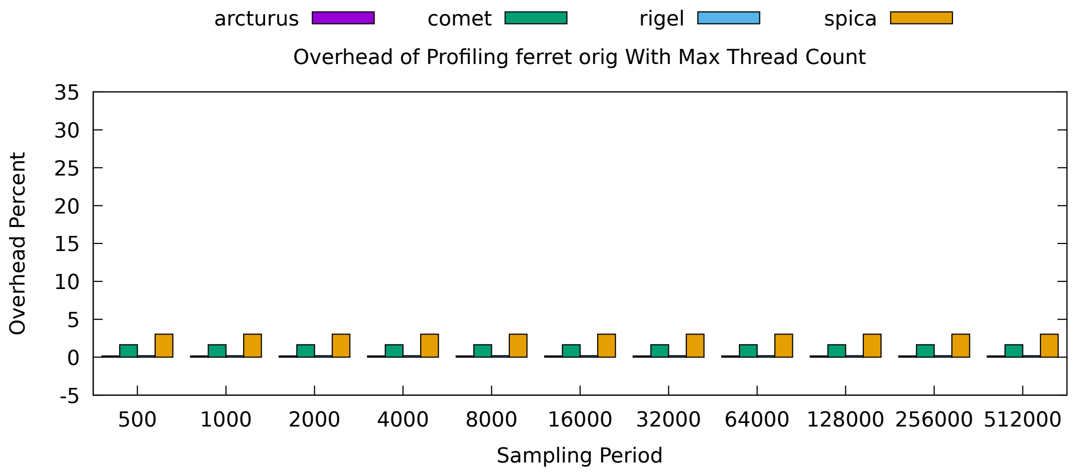


Figure 7.98. The overhead of profiling ferret with as much threads as physical cores available (Arcturus 44, Comet 36, Rigel 56, Spica 48).

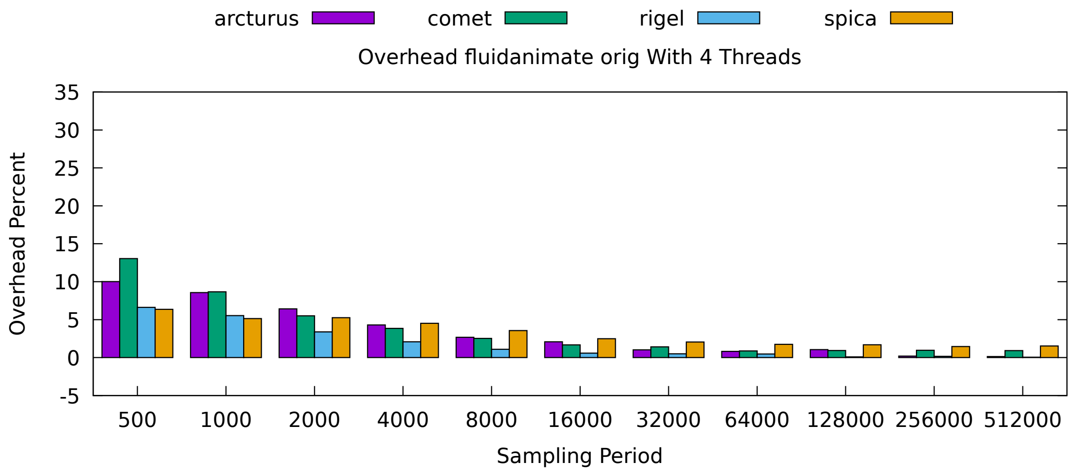


Figure 7.99. The overhead of profiling fluidanimate with 4 threads.

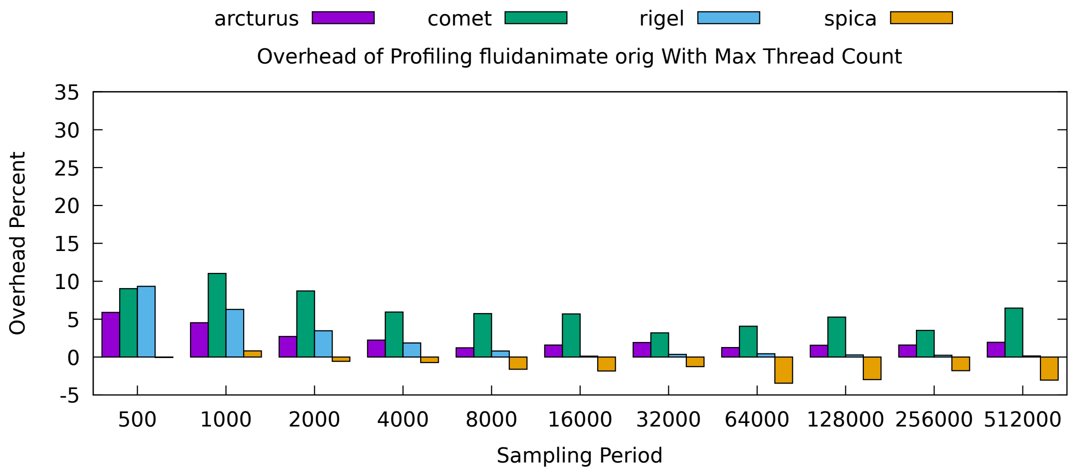


Figure 7.100. The overhead of profiling fluidanimate with 32 threads.

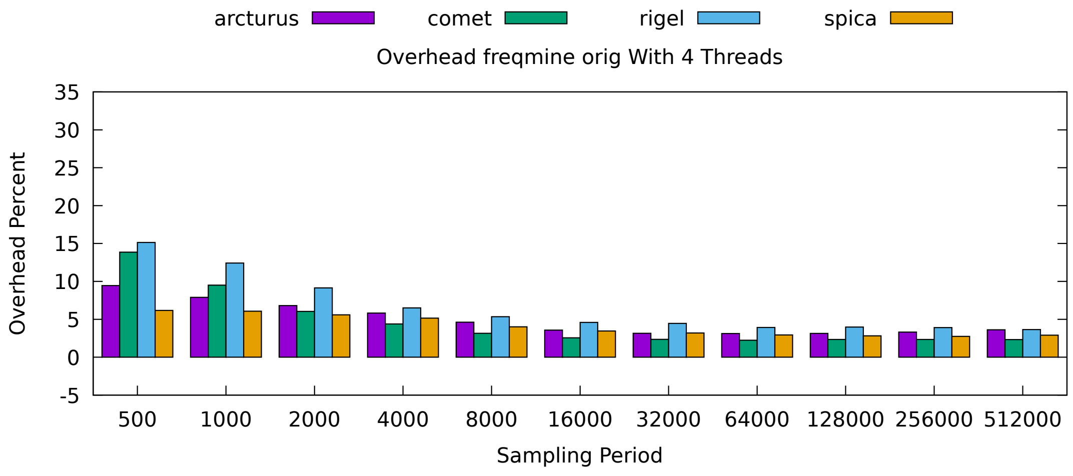


Figure 7.101. The overhead of profiling freqmine with 4 threads.

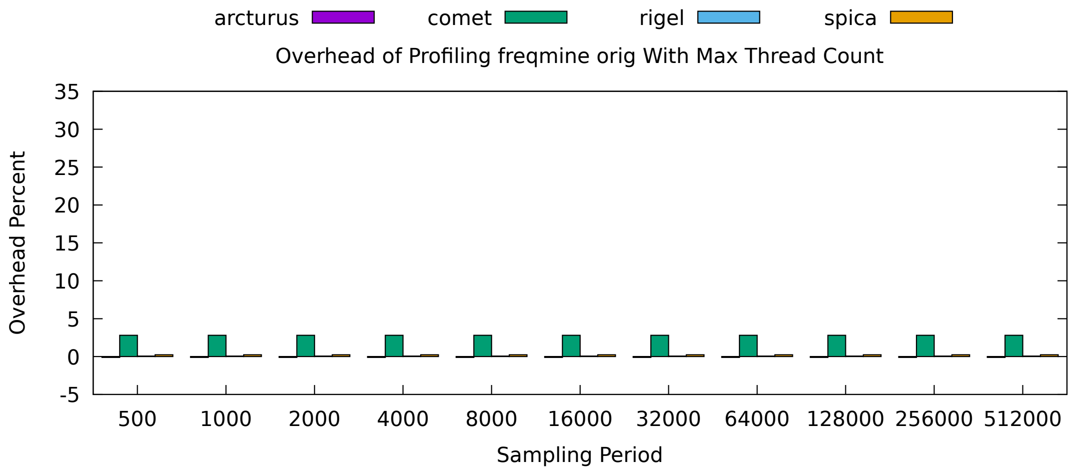


Figure 7.102. The overhead of profiling freqmine with as much threads as physical cores available (Arcturus 44, Comet 36, Rigel 56, Spica 48).

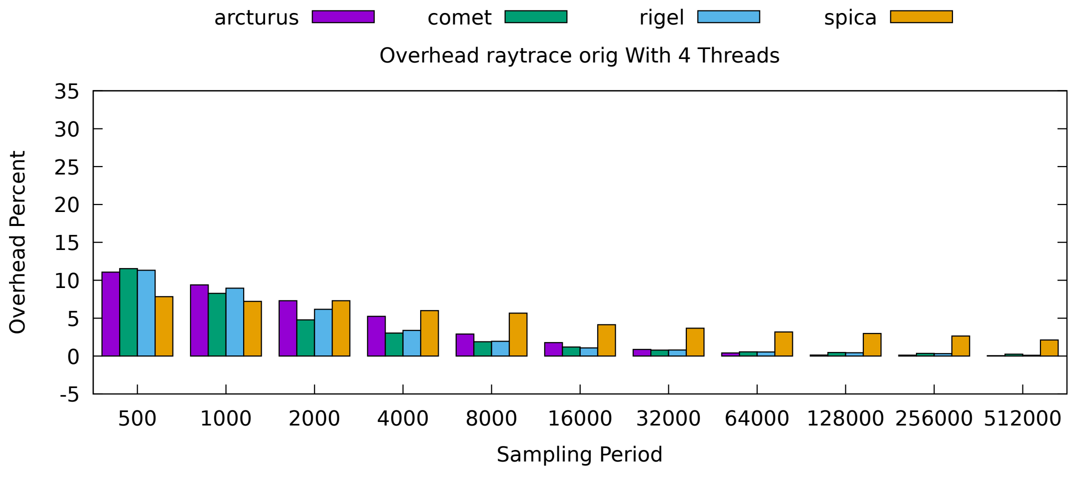


Figure 7.103. The overhead of profiling raytrace with 4 threads.

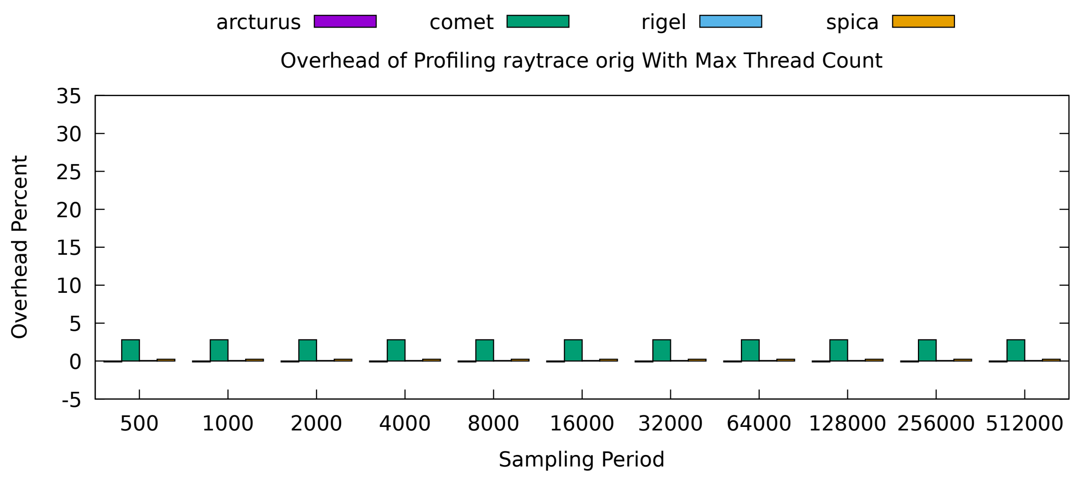


Figure 7.104. The overhead of profiling raytrace with as much threads as physical cores available (Arcturus 44, Comet 36, Rigel 56, Spica 48).

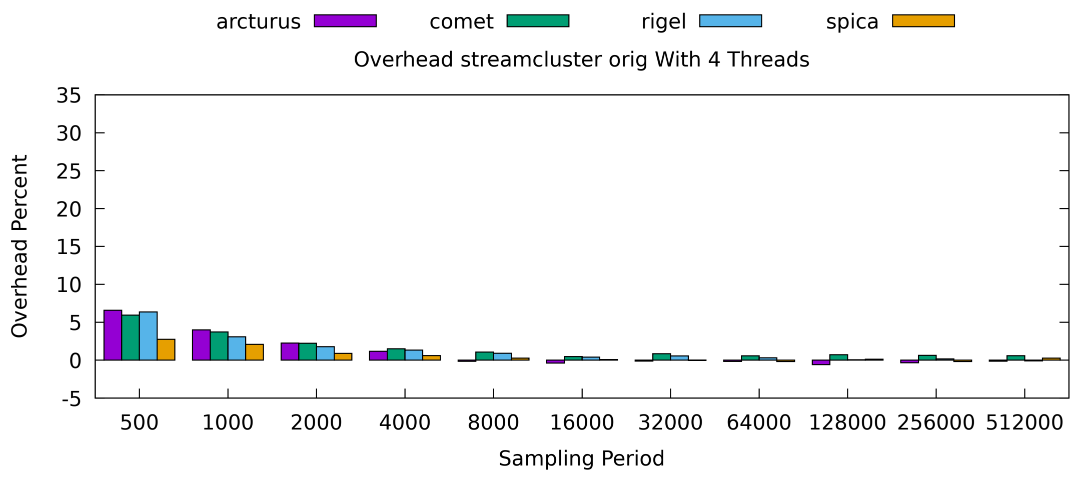


Figure 7.105. The overhead of profiling streamcluster with 4 threads.

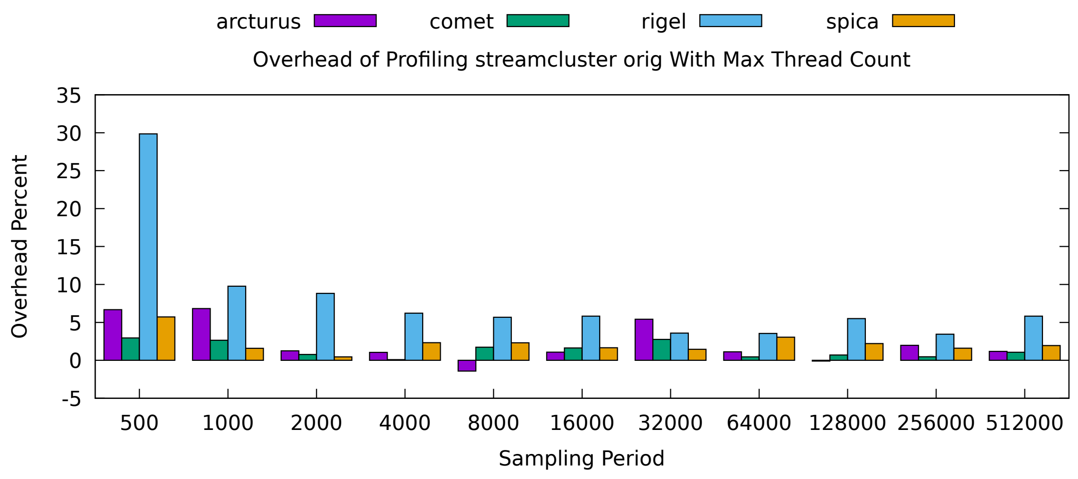


Figure 7.106. The overhead of profiling streamcluster with as much threads as physical cores available (Arcturus 44, Comet 36, Rigel 56, Spica 48).

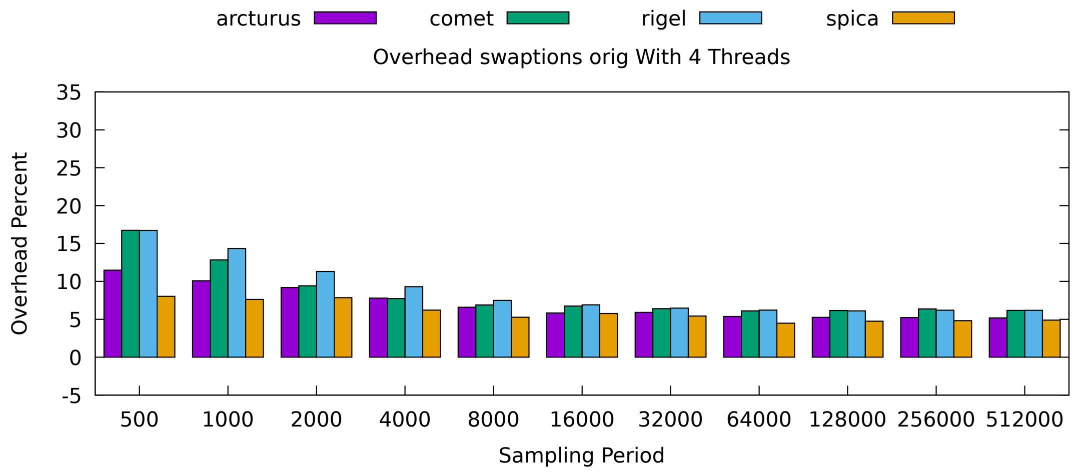


Figure 7.107. The overhead of profiling swaptions with 4 threads.

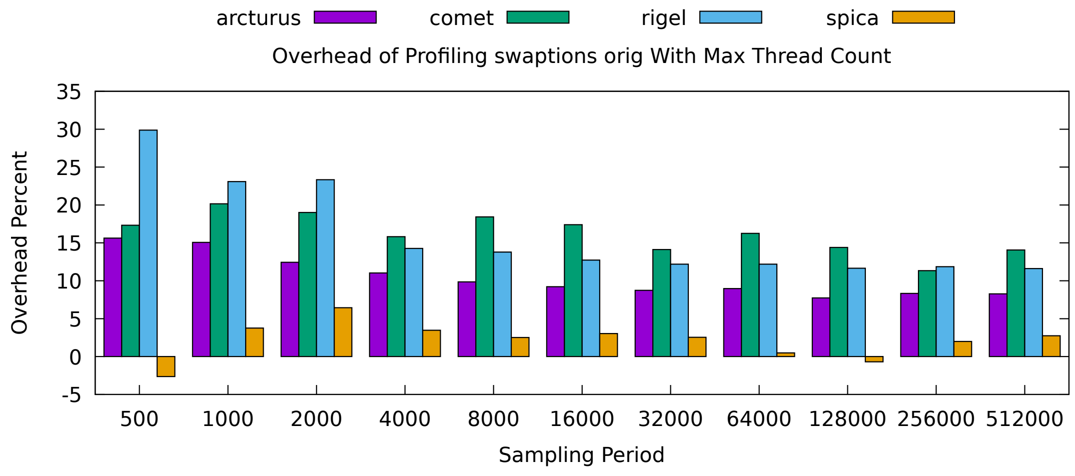


Figure 7.108. The overhead of profiling swaptions with as much threads as physical cores available (Arcturus 44, Comet 36, Rigel 56, Spica 48)..

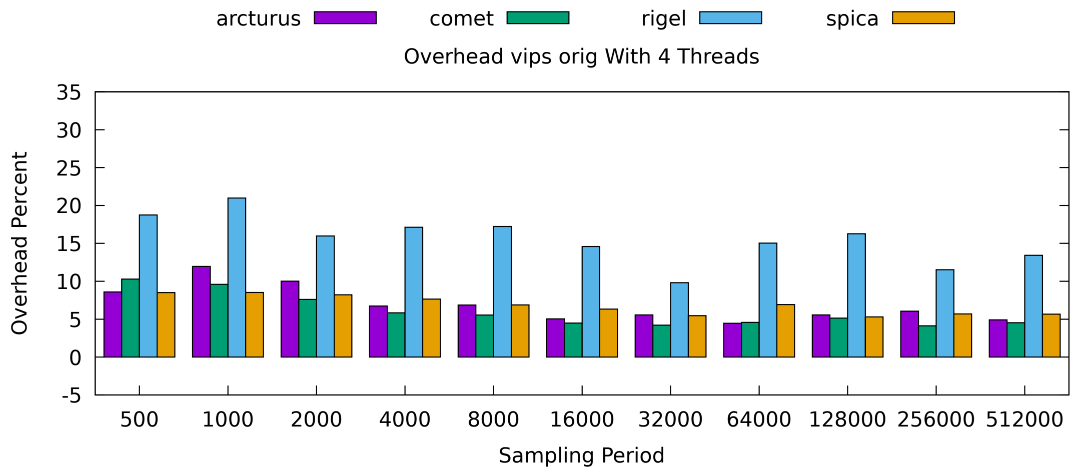


Figure 7.109. The overhead of profiling vips with 4 threads.

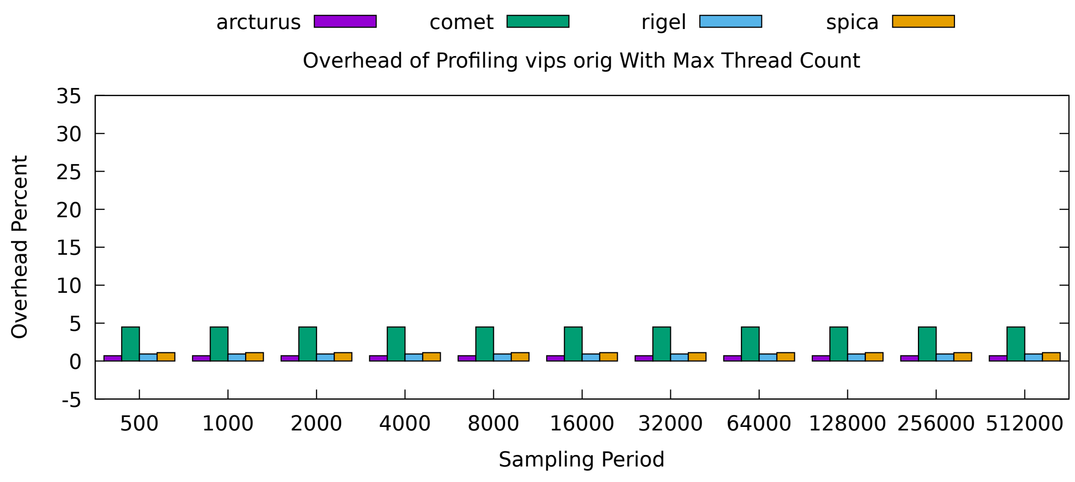


Figure 7.110. The overhead of profiling vips with as much threads as physical cores available (Arcturus 44, Comet 36, Rigel 56, Spica 48).

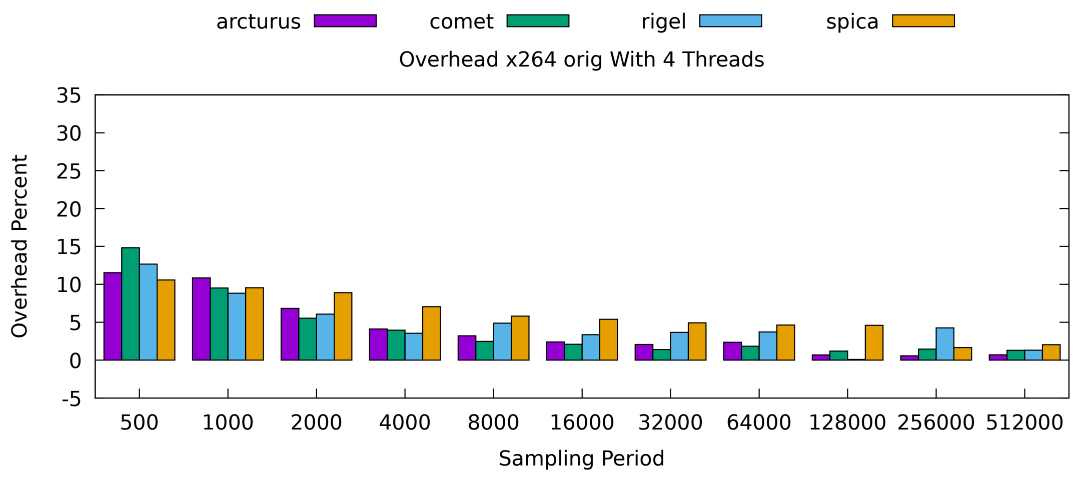


Figure 7.111. The overhead of profiling x264 with 4 threads..

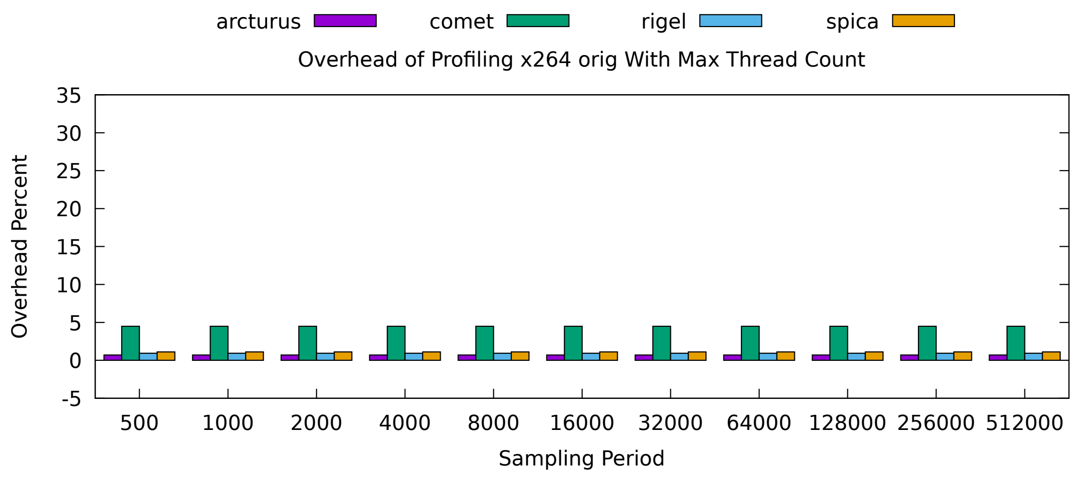


Figure 7.112. The overhead of profiling x264 with as much threads as physical cores available (Arcturus 44, Comet 36, Rigel 56, Spica 48).

7.7.5 Comparison with Existing Tools

PerfMemPlus supports the automatic discovery of two different performance problems and, in addition, manual analysis of performance data, that was captured with one profiling run. No existing tool has this amount of features. We compare the overhead of PerfMemPlus with tools that support a part of the features. Overall, even though PerfMemPlus supports more performance problem discovery features, the overhead is similar to tools that specialize on one specific performance problem. Some of the specialized false sharing detection tools slightly outperform PerfMemPlus in terms of overhead.

False Sharing Detection

Among the false sharing detection tools, Predator has a rather high overhead. It is less than 50% in 17 applications of PARSEC and Phoenix [68]. In the remaining 5 applications, it has a higher overhead of up to 23 times. Feather has low overhead but requires some custom optimizations in the Linux kernel to achieve it. The overhead increases with the number of threads and also depends on the sampling period. In the PARSEC benchmarks, it is between 1% and 209%. With four threads, the geometric mean ranges from 3% to 5%. With 16 threads, from 7% to 9% and with 32 threads, from 12% to 15% [13]. Jayasena et al. report an overhead of 2% but do not show a detailed analysis of the overhead [55]. TMI has an average overhead of 2% and a maximum overhead of 17% [17]. The average overhead of Sheriff is 20% but there are two exceptions with an overhead of 8.2 and 11.4 times [66]. PerfMemPlus achieves good detection results of greater than 90% with a sampling period of up to 4000. A sampling period of 8000 it performs only slightly worse. At those sampling periods, the average overhead is below 5% and below 4% respectively. The overhead is in the same range as the best false sharing detection tools.

DRAM Contention Detection

For DRAM contention detection, there is no existing tool with the same feature set available. However, DR-BW is solving a related problem of finding remote DRAM contention in NUMA systems. It has an average overhead of 3.3% with a maximum of 10% when using 64 threads [128]. The choice of a reasonable sampling rate for DRAM detection is more difficult. In the micro-benchmarks, a sampling period of 500 or 1000 is required for accurate detection. At this sampling period, the average overhead is below 3% and below 2% respectively, as shown in Figure 7.82. In the PARSEC applications, a sampling period of up to 4000 is suitable. When profiling PARSEC at this sampling rate, the average overhead about 8%. This is shown in Figure 7.86.

Manual Analysis

Regarding tools with manual analysis features and visualizations, the overhead of MemAxes is reported in only two benchmarks. In both cases, it is below 10% [30]. NumaMMA, which provides a time/address visualization that is similar to the one of PerfMemPlus, comes with an overhead of at most 12% [120].

7.7.6 Possibilities to Reduce the Overhead

Some of the specialized tools have a lower overhead than PerfMemPlus. A measure to reduce the overhead for false sharing detection and DRAM contention detection would be to increase the latency limit from which latency samples are taken. This is a feature in the hardware to ignore low-latency loads. But because we want a full picture of the memory accesses for the detection of other problems, we have set this limit to zero. With a limit of zero, we can record cache hit rates. If those are not of interest, increasing this threshold is a good way to reduce the overhead. The minimum allocation size to track is another way to reduce the overhead. If there are many malloc calls in an application, each one incurs additional overhead. By ignoring allocations smaller than a specified size, the overhead can be reduced.

7.8 Data Processing Overhead

PerfMemPlus requires a post-processing phase after the profiling to prepare the samples and allocation data for analysis. The details of this process are described in Section 5.1. We analyze the post-processing delay using the PARSEC benchmarks executed with the maximum thread count supported by the system. Those are the closest to the application of PerfMemPlus on real HPC applications.

The post-processing consists of three phases. First, the samples captured by perf are written into the SQLite database. Second, the data of the allocation tracker is written into the SQLite database. Finally, the allocation data is merged with the samples. For each memory access sample, the accessed object is assigned. The average post-processing time, split into the three phases, is shown in Figure 7.113. The import of the allocation consumes most of the time. It does not change with the sampling period because the sampling rate has no influence on the allocation tracking. The time required import of the samples and processing of the samples decreases with higher sampling periods. The time required for reading the allocation data depends on the number of allocations that were recorded. The minimum allocation size can be adjusted to change the amount of recorded data.

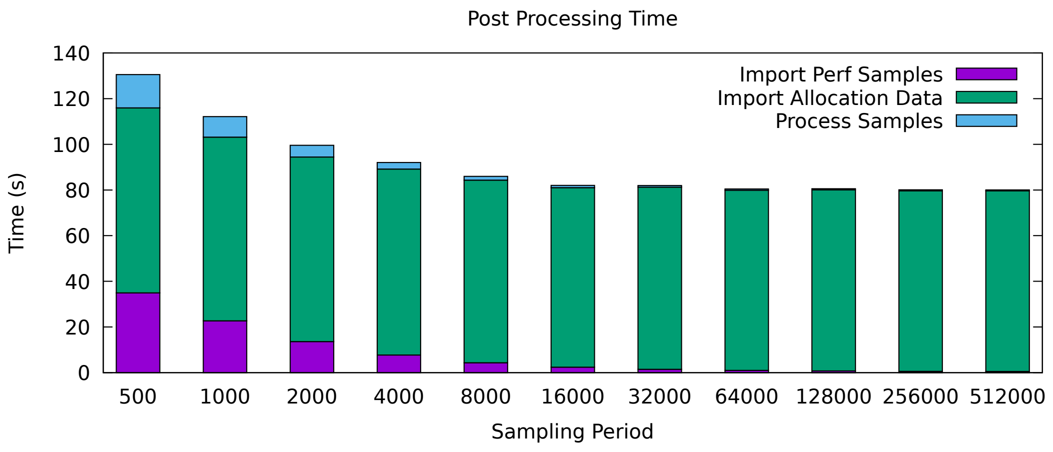


Figure 7.113. The processing time of PerfMemPlus split into three phases. The displayed data is the average over all PARSEC benchmarks executed with the maximum thread count.

The post-processing time and recorded data size differ greatly depending on the profiled benchmark. Figure 7.114 shows a comparison of the different benchmarks. The size of the allocation data is independent of the sampling rate. It is shown as the bar on the left for each benchmark. The next two bars show the size of the sample data. The size of the sample data depends on the sampling rate. Thus, this figure shows the size of the sample data for different sampling periods. As expected, the sampling data size decreases with higher sampling periods. The two bars on the right show the post-processing time. They correspond to the axis on the right. In the benchmarks where the allocation data size is large, the processing time is also large. In some benchmarks, like canneal or swaptions, an increased sampling period does only slightly decrease the post-processing time. Because reading the allocation data takes most of the time, and importing and processing the samples is quick in comparison. Facesim and fluidanimate also have long post-processing times. In those benchmarks, the sample import and processing takes more time. Thus the sampling rate has a higher influence on the post-processing time. Especially in facesim, where the allocation data size is below 40 MB. If the allocation tracking is turned off, the fraction required for importing the allocation data will be zero, and the sample processing will be close to zero because all samples will be updated to the same object.

In summary, the sampling period and the minimum allocation size both affect the post-processing time. It depends on the application, which of the parameter needs to be adjusted to reduce the post-processing delay. The size of the recorded data is a good indicator of the expected processing time.

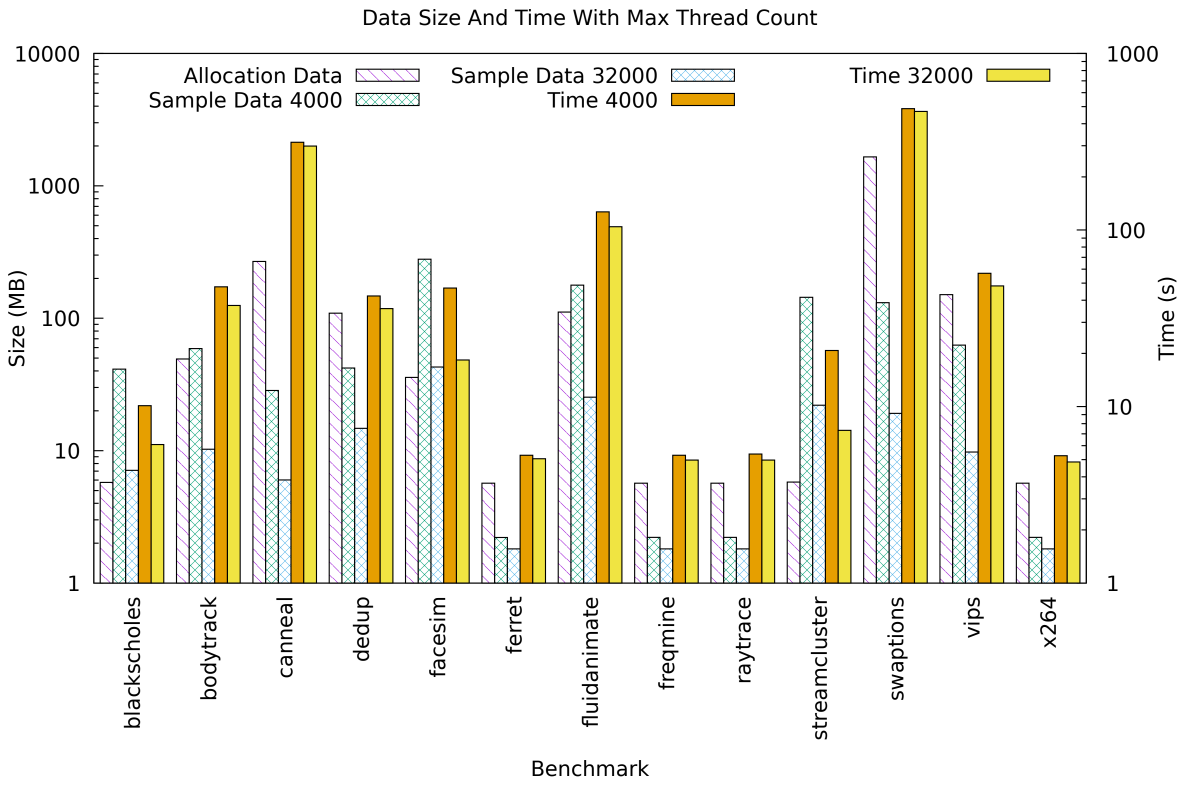


Figure 7.114. The data size and processing time when profiling the PARSEC benchmarks. The first three bars on the left side of every benchmark are data sizes and correspond to the axis on the left. The two bars on the right side of each benchmark are execution times and correspond to the axis on the right.

7.9 Case Studies

In this section we discuss how we found performance problems in three PARSEC benchmarks and two other machine learning applications.

7.9.1 Canneal

Our tool has automatically discovered DRAM contention on Spica but not on the other systems. A summary of the reported metrics is shown in Table 7.7. On Spica, the relative latency reaches values of up to two. On the other systems, the relative latency is below one. Table 7.1 shows that Spica has the lowest per-node bandwidth of all systems. We conclude that on this system, with the low DRAM speed, the performance suffers. But on the other systems, with a higher DRAM speed, the available bandwidth is enough.

The origin of contention was reported in the function netlist_elem::swap_cost when it accesses the std::vector called elements. This matches previous findings of Eyerman [26], and the fact that this benchmark has the second-highest bandwidth requirement [10] of all PARSEC benchmarks. For the identified function and object, there is a NUMA imbalance of 1, which indicates that data is completely allocated on one node but then accessed from all nodes.

Looking at the details, the automatic discovery reported two instances of memory bandwidth limitation. Both occur in the function netlist_elem::swap_cost. The first one when accessing the std::vector elements. The second one when accessing the std::vector locations.

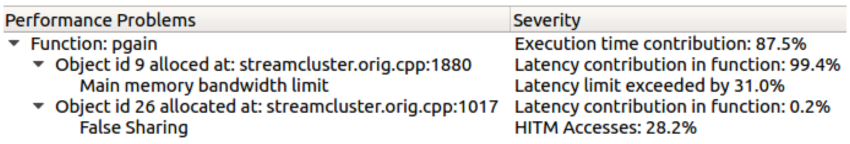
In order to make better use of both memories, we applied interleaved allocation to those two objects by implementing a custom allocator for std:vector. There are performance improvements on Spica, but not on the other systems. Even with interleaved allocation, the NUMA imbalance does not drop to 0. That is because, in Canneal, the array elements accessed by each thread are decided randomly. Thus even if the data is distributed evenly across the whole system, the accesses can still be imbalanced. After applying interleaved allocation, the relative latency drops to values around one on Spica.

Table 7.7. Interleaved allocation only leads to speedup of the Canneal benchmark on the slower systems. On the faster systems no DRAM contention is detected.

Server	Speedup	Allocation	Relative Latency	NUMA Imbalance
Arcturus	0.0%	default	0.91	1.00
		interleave	0.79	0.76
Comet	3.0%	default	0.81	1.00
		interleave	0.81	0.77
Rigel	-5.1%	default	0.53	0.90
		interleave	0.49	0.67
Spica 2 Nodes	30.4%	default	1.27	1.00
		interleave	0.90	0.76
Spica 4 Nodes	42.0%	default	2.0	1.00
		interleave	1.10	0.80

7.9.2 Streamcluster

Our automatic analysis reports that, in the function pgain, accesses to the array block exceed the available bandwidth and that there a is high NUMA imbalance. This findings match those of previous analysis [9,10,61,128]. Table 7.8 shows the relative latency, NUMA imbalance, and speedup on different servers. The report given by PerfMemPlus is shown in Figure 7.115. In the function pgain, accesses to the array block exceed the available bandwidth. The NUMA imbalance only considers the object and function where DRAM contention was reported. The relative latency is high on Spica, around the limit of 1 on Arcturus and Comet, and low on Rigel. The NUMA imbalance is high, and the interleaved allocation of the identified object results in increased performance. The speedup is higher on the systems with higher relative latency.



Performance Problems	Severity
▾ Function: pgain	Execution time contribution: 87.5%
▾ Object id 9 alloced at: streamcluster.orig.cpp:1880	Latency contribution in function: 99.4%
Main memory bandwidth limit	Latency limit exceeded by 31.0%
▾ Object id 26 allocated at: streamcluster.orig.cpp:1017	Latency contribution in function: 0.2%
False Sharing	HITM Accesses: 28.2%

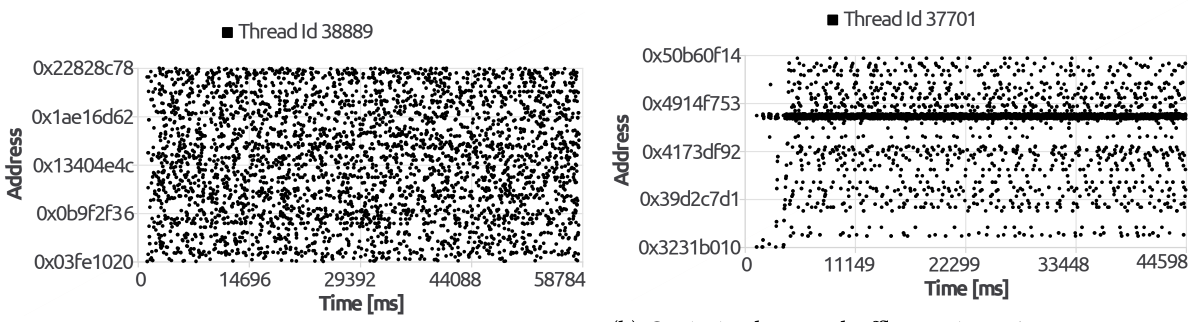
Figure 7.115. The automatic discovery report lists the performance problems and their locations in the streamcluster benchmark.

Table 7.8. The NUMA imbalance of Streamcluster is high on all systems. Interleaved allocation brings higher speedups on the systems with higher relative latency.

Server	Speedup	Allocation	Relative Latency	NUMA Imbalance
Arcturus	30.2%	default	1.06	1.00
		interleave	0.91	0.01
Comet	41.2%	default	0.94	1.00
		interleave	0.96	0.11
Rigel	9.5%	default	0.64	0.91
		interleave	0.51	0.01
Spica 2 Nodes	66.6%	default	1.84	1.00
		interleave	0.80	0.01
Spica 4 Nodes	70.9%	default	2.35	1.0
		interleave	0.85	0.02

Additionally, false sharing occurs in the function pgain accessing an array called work_mem. The array and access locations were pointed out by the automatic discovery of PerfMemPlus. When we checked the source code at the indicated location we found that there is already padding to prevent false sharing in this application. But it assumes a cache line size of 32 bytes. We set the padding to match the real cache line size of 64 bytes. Like shown in Figure 7.2 it leads to a speedup between 1% and 6% depending on the system.

The third known problem of bad locality [75] cannot be detected by our automated approach. However, it can be diagnosed using visualizations created by PerfMemPlus. The function pgain accessing the array block was identified as the main offender from the function and object profiles. We focused the analysis on this function and object. Figure 7.116a shows the addresses within the array block that are accessed over time. This diagram shows one specific thread, but it looks similar for other threads. Each point represents one access sample. The figure shows no clear structure and addresses are accessed randomly. Moreover, every thread accesses the whole range of the array.



(a) Initial pointer shuffle version. Accesses are distributed randomly throughout the whole array.

(b) Optimized copy shuffle version. Accesses concentrate on a small space of the array indicated by the thick horizontal line.

Figure 7.116. The access pattern of the array block in the function pgain in one thread of the streamcluster benchmark.

The reason for this random access is that the clustering part is repeatedly executed and data needs to be processed multiple times in different orders for this algorithm to work correctly. In this implementation, not the data itself is shuffled, but pointers to the data are shuffled. Consequently, in every iteration different addresses are accessed by every thread. A comment in the source code indicates that it was done to avoid copying the large data elements and to increase performance. We changed it to a copy based shuffle operation that copies the actual data and does not change the pointers as already suggested by Majo et al. [75]. The accessed addresses stay the same even though the underlying data changes. By using PerfMemPlus to display the access patterns such problems can be discovered easier compared to manual examination, which was used in the previous publication. In the optimized version, we saw that a thread mostly accesses the same part of the array throughout the execution. This is indicated by the horizontal line in Figure 7.116b. The diagrams for other threads look similar with the horizontal line shifted on the vertical axis because they access a different part of the array. A colored version of this diagram, that shows four threads is available in Figure 5.12. This optimization resulted in an improved L1 hit rate in the concerned functions and objects. On Spica, the L1 hit rate increases from 72% to 94%. The result of this optimization is a speedup between 40% on Rigel and 2.5 times on Spica, as shown on the right side of Figure 7.28.

7.9.3 Freqmine

We found a case of false sharing in freqmine. It was not found by specialized false sharing detection tools [13, 55]. The performance penalty is similar to the one in streamcluster. Both of them have found the false sharing in streamcluster.

Perf c2c [77] does also detect modified cache lines. However, it cannot identify the object and cannot confirm whether there is true or false sharing. With a manual exploration of the data captured by Intel VTune [47] false sharing can also be detected. But it can not automatically confirm false sharing.

We verified that it is an actual case of false sharing by reading the source code. The falsely shared object is the class stack with a size of 20 bytes. There is an array named list, which is defined in fp_tree.cpp, with one element of type stack for each thread. If the stack objects are placed in one cache line, false sharing occurs. Because allocation is done using the new operator for each stack item individually, the placement depends on the memory allocator. In our case, we confirmed that individual stack items are placed in the same cache line with an empty space of 16 bytes between the objects. The method shown in Section 5.3.2 prints the allocations that were issued from the same place in the source code and that share cache lines. Write accesses happen at multiple lines inside the parallel section of the FP_tree::FP_growth function. The fields FS and top of the class stack are written.

There are a few possible reasons why the previously developed tools have missed this case of false sharing. First of all, in freqmine, the occurrence of false sharing depends on the memory allocator. If the memory allocator makes a different choice, then false sharing may not occur. The performance impact of false sharing in this benchmark is low. The existing tools may not consider such a light case of false sharing.

The machine learning approach by Jayasena et al. [55] is based on a trained classifier. This tool uses data from the PMU that also includes the HITM flag. Thus, it should have the required data for detection. But if the training data does not represent the behavior that happens in freqmine, false sharing can not be found.

The second tool [13] detects false sharing independently of objects. Only after a pair of addresses has been identified the object is resolved. It is not explained if an object is identified by a common call path or individual malloc calls. There are statistical limitations, that could lead to missing a case of false sharing. First, a thread must execute a certain number of memory accesses to be considered. Second, their approach needs to work with a limited amount of debug registers and can only cover a part of the addresses. There are four debug registers available on Intel processors. Each has alignment and size constraints. The constraints are:

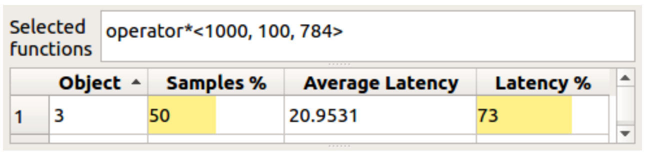
- Address length of 1,2,4 or 8 bytes can be monitored.
- The address must be naturally aligned to the monitored length.

For example, if a four byte address is monitored the start address must be aligned to four bytes. Because cache lines are 64 bytes long, only a part of a cache line can be covered. The address where a previous access was recorded doesn't need to we watched. But all other addresses in the cache line should be covered. First, the size of the watched addresses must be picked. This is done assuming an infinite amount of debug registers. Monitoring of eight byte regions would be best because it allows covering a higher fraction of the cache line. But if the eight byte regions would partially overlap with the previously published address smaller ranges are used. After this first step, there is a list of debug register configurations with start address and size. From this list, four entries are chosen randomly and the real debug registers are programmed. The authors mention that it is a probabilistic approach, that has a high probability to detect false sharing, but it is not guaranteed to detect it.

7.9.4 Mnist

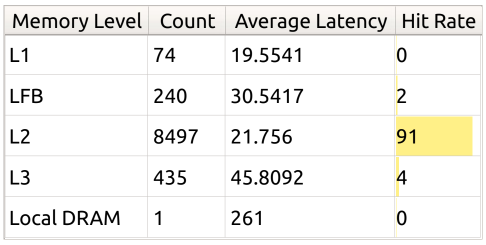
The application called Mnist [116] is a neural network implemented in plain C++ without using any specialized libraries. It performs handwritten digit recognition on the Mnist [63] dataset. It is a single-threaded application. As it is typical for neural networks, this application executes many matrix multiplications. The matrix multiplications are implemented as operator* function templates. The template parameters specify the matrix size.

The function profile showed that the operator*<1000, 100, 784> has a higher latency than any other function. The objects accessed by this function are shown in Figure 7.117. All high latency accesses go to one object with id 3. The allocation call stack showed that object 3 is a matrix. Figure 7.118 shows the cache hit rates of the function operator*<1000, 100, 784> when it accesses the object 3. About 91% of the memory accesses hit the L2 cache. All other functions have cache L1 hit rates over 95%. Only this one function performs badly.


Selected functions: operator*<1000, 100, 784>

	Object	Samples %	Average Latency	Latency %
1	3	50	20.9531	73

Figure 7.117. The object profile of objects accessed by operator*<1000, 100, 784>. The screenshot is cut to show only the most significant object. Accesses to one specific object are responsible for the majority of latency.



Memory Level	Count	Average Latency	Hit Rate
L1	74	19.5541	0
LFB	240	30.5417	2
L2	8497	21.756	91
L3	435	45.8092	4
Local DRAM	1	261	0

Figure 7.118. The cache hit rates and latencies of the operator*<1000, 100, 784> function accessing one specific matrix with id 3. Most of the accesses hit the L2 cache. Other functions have high L1 cache hit rates.

We implemented cache blocking in the operator*<1000, 100, 784> function. We only changed the one template specialization that processes the identified matrix type and did not modify other functions that multiply matrices of different sizes. PerfMemPlus pointed out only the function and object where modification was necessary and did not report cases where no optimization was necessary. The performance was increased by 12.8% on Spica and 7.7% on Comet. L1 cache hit rates of the specific function and object were increased to 99.8%.

Figure 7.119 shows the overhead in relation to the sampling period. As expected, the overhead rises when more samples are taken. Under 3% overhead can be achieved.

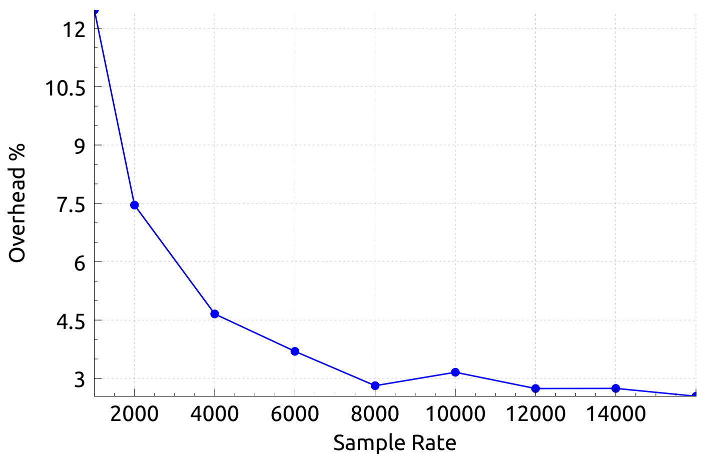


Figure 7.119. The overhead of profiling the Mnist benchmark.

7.9.5 N3LP

N3LP [100] is a neural machine translation application. It is implemented using the Eigen library [35] for arithmetic operations.

The automatic analysis showed that there is DRAM contention in six different functions. Figure 7.121 shows a profile of those functions. It is detected on all systems. The relative latency is higher than in any of the other benchmarks, as shown in Table 7.9.

To get more details, we looked at inlined functions, callstacks and source code lines of the reported functions, which is possible due to the precision of the instruction sampling. Almost all the latency of the function SoftMax::backward is produced by the source code line: *grad.weight += delta*input.transpose().* Where delta and input are two vectors. Assuming that the Eigen library already does as much optimization as possible, we suspected that the culprit must be in the way it is used. We resolved the call stacks to each of the other reported functions to find out from where in the user code they are called. Figure 7.122 shows one of those callstacks. Normal profilers use the fraction of the execution time to sort functions. Whereas PerfMemPlus shows the latency contribution, coming from each calling function. In summary, we identified that mainly the functions LSTM::backward and SoftMax::backward are calling the Eigen operations with high latency. The cache and memory details window of PerfMemPlus showed that the average L1 cache hit rate in those six functions is only 40% and only 15% in SoftMax::backward. The NUMA imbalance is at most 0.29. There is only a small imbalance, and as expected, interleaved allocation does reduce the NUMA imbalance, but only results in a minimal performance benefit of at most 6.2%.

Instead of interleaved allocation, Qiao et al. [100] introduced a different optimization that increases data locality. In this optimization, the single vectors are merged into a matrix. This way Eigen can execute a matrix-matrix multiplication instead of multiple vector-vector multiplications and Eigen is able to apply better optimizations such as cache blocking. Applying this optimization, we achieved higher cache hit rates and performance improvements of 3.45x on Spica with 48 threads and 3.14x on Comet with 36 threads. Parallel scalability was significantly improved as shown in Figure 7.120. With a sampling period of 8000 the overhead is around 13% on Comet and Spica.

Table 7.9. N3LP suffers from severe DRAM contention on all systems. The NUMA imbalances is low and consequently interleaved allocation does not result in large speedups.

Server	**Speedup**	**Allocation**	**Relative Latency**	**NUMA Imbalance**
Arcturus	0.0%	default	2.67	0.23
		interleave	1.75	0.0
Comet	6.2%	default	3.70	0.29
		interleave	2.74	0.02
Rigel	0.0%	default	2.50	0.20
		interleave	2.4	0.0
Spica 2 Nodes	2.0%	default	4.56	0.24
		interleave	3.53	0.01
Spica 4 Nodes	6.0%	default	4.33	0.20
		interleave	3.59	0.097

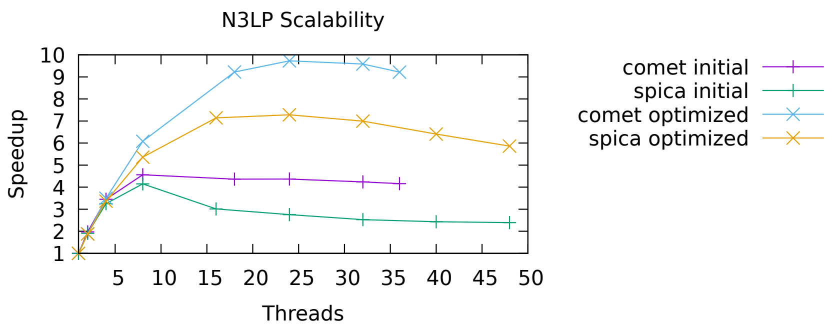
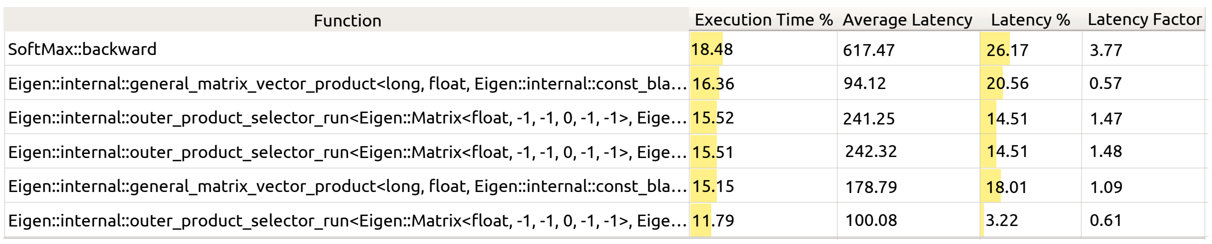
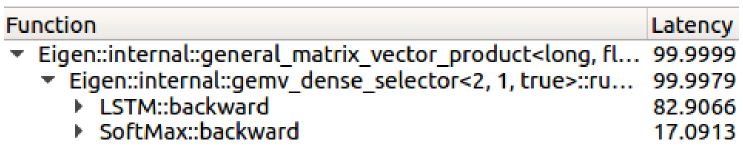


Figure 7.120. The parallel scaling of the initial and the optimized version of N3LP on Comet and Spica.



Function	Execution Time %	Average Latency	Latency %	Latency Factor
SoftMax::backward	18.48	617.47	26.17	3.77
Eigen::internal::general_matrix_vector_product<long, float, Eigen::internal::const_bla…	16.36	94.12	20.56	0.57
Eigen::internal::outer_product_selector_run<Eigen::Matrix<float, -1, -1, 0, -1, -1>, Eige…	15.52	241.25	14.51	1.47
Eigen::internal::outer_product_selector_run<Eigen::Matrix<float, -1, -1, 0, -1, -1>, Eige…	15.51	242.32	14.51	1.48
Eigen::internal::general_matrix_vector_product<long, float, Eigen::internal::const_bla…	15.15	178.79	18.01	1.09
Eigen::internal::outer_product_selector_run<Eigen::Matrix<float, -1, -1, 0, -1, -1>, Eige…	11.79	100.08	3.22	0.61

Figure 7.121. The function profile, which shows the functions contributing to the latency in N3LP.



Function	Latency
▼ Eigen::internal::general_matrix_vector_product<long, fl…	99.9999
▼ Eigen::internal::gemv_dense_selector<2, 1, true>::ru…	99.9979
▶ LSTM::backward	82.9066
▶ SoftMax::backward	17.0913

Figure 7.122. The callstack of one of the reported functions in the Eigen library. The percentage of total latency is shown in the right column.

Chapter 8

Conclusion

In this thesis, we show that instruction sampling can be used to diagnose a variety of different memory performance problems in an application. Hardware instruction sampling enables profiling with low overhead and provides data about the hardware that is not available through purely software-based methods. We show that it is feasible to collect data from the complete application, without limitation to specific parts, and then automatically search for different types of memory performance problems within this data. This enables easy performance analysis of unknown applications. Especially our automatic discovery features simplify the search for performance problems. For this, we introduce a new approach for finding DRAM contention and methods to diagnose the cause of the contention. We also improve existing false sharing detection techniques. The profiling overhead is comparable to other existing approaches that can only find one specific type of performance problem.

Instead of looking at the consumed DRAM bandwidth, we proposed that the DRAM access latency is a better indicator of DRAM contention. The DRAM access latency can be recorded with instruction sampling, and thus the origin of DRAM contention can be attributed to individual instructions and objects. The DRAM access latency also allows differentiating DRAM contention from harmless high bandwidth consumption. The severeness of the DRAM contention is also expressed by the DRAM access latency. The effectiveness of prefetching is also considered in the DRAM latency.

Our experiments show the instruction sampling latency consists of in-core delays and DRAM access delays. Nevertheless, it is suitable to use it for the detection of DRAM contention. However, in some cases, we would like to know the bandwidth consumption of a program and not just the information if there is bandwidth contention or not. In this case, we must rely on performance counters for memory bandwidth. There is a variety of counters that seemingly measure the main memory bandwidth. But we showed that only one of them, the IMC counter measures the correct memory bandwidth. Other counters do not include prefetched accesses. Thus, we suggest using the IMC counters for bandwidth measurements.

Our false sharing detection method uses the same data as the DRAM contention detection and manual analysis features. It does not produce false positives, and we found a previously unknown case of false sharing.

We provide an open-source tool that supports a wide range of existing processors, and support for future generations will be available because we rely on Linux Perf. Our profiling tool is optimized for analyzing memory performance problems and is easy to use for this purpose. With our SQLite base data format, we enable easy access and exploration of the instruction sampling data for future research.

8.1 Outlook

Instruction sampling is a hardware feature that was introduced in x86 processors over 10 years ago. But still, even with our new contributions, there is a lot of information in this data that is currently unused. For example, information about TLB misses, locked accesses, and access patterns are neglected in previous publications as well as in this thesis. With our tool implementation, we create a foundation for exploring these aspects in the instruction sampling data.

Our detection of DRAM contention and false sharing is still very simple because it gives only a binary classification. A metric, that can predict the possible performance gain that can be achieved by fixing the detected problems would be useful. The memory latency and the number of memory accesses provide a base for such an estimation. But current processors have many features to hide memory delays. How well those features work on a given code would have to be considered. We think that there are performance counters available that would be suitable to report such detail about the execution of instructions. By combining this information it will be possible to make a prediction of how much a certain performance problem influences the overall execution time of the application.

An obvious extension of our work would be to port it to AMD processors, which recently re-gained attention in the HPC market [28]. Because Linux Perf also supports AMD processors, no changes in the profiling software would be necessary. However, the exact events to monitor are different on AMD hardware, and similar to our experiments, the true meaning of the reported data has to be verified first. In this thesis, we provide a portable way to do this.

Choosing suitable profiling settings is troublesome in some cases. There must be enough samples to draw reliable conclusions. On the other hand, the overhead should not be too high. The profiling and data processing overhead depends on two settings. The sampling rate and the minimum allocation size. Depending on the application's execution time, amount of allocations, and amount of memory accesses, the optimal setting differs. If the profiling options would be set automatically, or adjusted during runtime as necessary, it would make our tool easier to use.

Bibliography

[1] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., and Tallent, N. R. HPCT OOLKIT : Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* (2010).

[2] Akiyama, S. A lightweight method to evaluate effect of approximate memory with hardware performance monitors. *IEICE Transactions on Information and Systems E102D*, 12 (2019), 2354–2365.

[3] Akiyama, S., and Hirofuchi, T. Quantitative Evaluation of Intel PEBS Overhead for Online System-Noise Analysis. *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers - ROSS '17* (2017).

[4] Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S.-t. A., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A., and Weihl, W. E. Continuous Profiling : Where Have All the Cycles Gone? Tech. rep., Systems Research Center, 1997.

[5] Andreolli, C. Integrated roofline model with intel advisor. https://software.intel.com/en-us/articles/integrated-roofline-model-with-intel-advisor.

[6] arm. Coresight debug and trace. https://developer.arm.com/ip-products/system-ip/coresight-debug-and-trace.

[7] Baptiste Lepers. Memprof cource code. https://github.com/Memprof, 2016.

[8] Barrow-Williams, N., Fensch, C., and Moore, S. A communication characterisation of Splash-2 and Parsec. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on* (2009), IEEE, pp. 86–97.

[9] Bhadauria, M., Weaver, V. M., and Mckee, S. A. Understanding parsec performance on contemporary CMPS. *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009* (2009), 98–107.

[10] Bienia, C. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, 2011.

[11] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (2008), pp. 72–81.

[12] Bingmann, T. Parallel Memory Bandwidth Benchmark. https://panthema.net/2013/pmbw/, 2013.

[13] Chabbi, M., Wen, S., and Liu, X. Featherlight on-the-fly false-sharing detection. *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2018), 152–167.

[14] Chandru, V., and Mueller, F. Reducing NoC and memory contention for manycores. *Lecture Notes in Computer Science 9637* (2016), 293–305.

[15] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S. H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009* (2009), pp. 44–54.

[16] Dean, J., Hicks, J. E., Waldspurger, C. A., Weihl, W. E., and Chrysos, G. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the Annual International Symposium on Microarchitecture* (1997), pp. 292–302.

[17] DeLozier, C., Eizenberg, A., Hu, S., Pokam, G., and Devietti, J. TMI: Thread memory isolation for false sharing repair. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO* (2017), 639–650.

[18] Diener, M., Cruz, E. H., Alves, M. A., and Navaux, P. O. Communication in Shared Memory: Concepts, Definitions, and Efficient Detection. *Proceedings - 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016* (2016), 151–158.

[19] Ding, W., Kandemir, M., Guttman, D., Jog, A., Das, C. R., and Yedlapalli, P. Trading cache hit rate for memory performance. In *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT* (2014), pp. 357–368.

[20] Doweck, J. Inside Intel Core Microarchitecture and Smart Memory Access. Tech. rep., Intel Corporation, 2006.

[21] Drebes, A., Pop, A., Heydemann, K., Cohen, A., and Drachtemam, N. Aftermath : A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems. *7th workshop on Programmability Issues for Heterogeneous Multicores* (2014).

[22] Drongowski, P. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. Tech. rep., 2007.

[23] Eklov, D., Nikoleris, N., and Hagersten, E. A software based profiling method for obtaining speedup stacks on commodity multi-cores. *ISPASS 2014 - IEEE International Symposium on Performance Analysis of Systems and Software* (2014).

[24] Eranian, S. What can performance counters do for memory subsystem analysis? In *Proceedings of the 2008 ACM SIGPLAN workshop on memory systems performance and correctness* (2008).

[25] Eranian, S. Linux perf_events status update. In *Scalable Tools Workshop* (2019).

[26] Eyerman, S., Du Bois, K., and Eeckhout, L. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. *ISPASS 2012 - IEEE International Symposium on Performance Analysis of Systems and Software* (2012), 145–155.

[27] Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Qu, V., and Roth, M. Traffic Management : A Holistic Approach to Memory Placement on NUMA Systems. *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems* (2013), 381–393.

[28] Feldman, M. AMD Notches EPYC Supercomputer Win with Next-Generation Zen Processor. https://www.top500.org/news/amd-notches-epyc-supercomputer-win-with-next-generation-zen-processor/.

[29] Fujitsu. FUJITSU Server PRIMERGY & PRIMEQUEST Memory Performance of Xeon scalable processor (Skylake-SP) based Systems.

[30] Gimenez, A., Gamblin, T., Jusufi, I., Bhatele, A., Schulz, M., Bremer, P. T., and Hamann, B. MemAxes: Visualization and Analytics for Characterizing Complex Memory Performance Behaviors. *IEEE Transactions on Visualization and Computer Graphics 27*, 5 (2017).

[31] Giménez, A., Gamblin, T., Rountree, B., Bhatele, A., Jusufi, I., Bremer, P. T., and Hamann, B. Dissecting On-Node Memory Access Performance: A Semantic Approach. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC* (2014), 166–176.

[32] GNU. gprof. https://sourceware.org/binutils/docs/gprof/.

[33] Graham Allan. Ddr4 bank groups in embedded applications. https://www.synopsys.com/designware-ip/technical-bulletin/ddr4-bank-groups.html.

[34] Gruss, D., Maurice, C., and Mangard, S. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *Lecture Notes in Computer Science* (2016), vol. 9721, pp. 300–321.

[35] Guennebaud, G., Jacob, B., et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[36] Günther, S. M., and Weidendorfer, J. Assessing cache false sharing effects by dynamic binary instrumentation. In *Workshop on Binary Instrumentation and Applications* (2009), pp. 26–33.

[37] Hackenberg, D., and Nagel, W. E. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *IEEE/ACM International Symposium on microarchitecture* (2009), pp. 413–422.

[38] Hennessy, J. L., and Patterson, D. a. *Computer architecture: a quantitative approach*, fourth ed. 2007.

[39] Hillenbrand, M. Physical Address Decoding in Intel Xeon v3/v4 CPUs: A Supplemental Datasheet.

[40] Huang, H., Liu, L., Song, F. L., and Ma, X. Y. Architecture supported synchronization-based cache coherence protocol for many-core processors. *Jisuanji Xuebao/Chinese Journal of Computers 32*, 8 (2009), 1618–1630.

[41] Huynh, A., and Taura, K. Delay Spotter: A Tool for Spotting Scheduler-Caused Delays in Task Parallel Runtime Systems. In *IEEE International Conference on Cluster Computing, ICCC* (2017), pp. 114–125.

[42] Intel Corporation. Disclosure of h/w prefetcher control on some intel processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.

[43] Intel Corporation. Finding your memory access performance bottlenecks. https://software.intel.com/en-us/articles/finding-your-memory-access-performance-bottlenecks.

[44] Intel Corporation. Intel advisor. https://software.intel.com/en-us/advisor.

[45] Intel Corporation. Performance Counter Monitor. http://www.intel.com/software/pcm.

[46] Intel Corporation. Intel Xeon Processor 5500 Series Datasheet, Volume 2, 2011.

[47] Intel Corporation. Avoiding and identifying false sharing among threads. https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads, 2012.

[48] Intel Corporation. Intel Xeon Processor E5-2600 Product Family Uncore Performance Monitoring Guide.

[49] Intel Corporation. Intel Xeon Processor E7 v4 Product Family Datasheet Volume 2: Registers, 2016.

[50] Intel Corporation. Intel® Xeon® Processor Scalable Memory Family Uncore Performance Monitoring Reference Manual. Tech. Rep. July, 2017.

[51] Intel Corporation. Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, 2018.

[52] Intel Croporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3.

[53] Iwainsky, C., Reichstein, T., Dahnken, C., Mey, D. A., Terboven, C., Semin, A., and Bischof, C. An approach to visualize remote socket traffic on the intel Nehalem-EX. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 6586 LNCS* (2011), 523–530.

[54] Jacob, B., Ng, S. W., Wang, D. T., and Schuette, M. *Memory systems: cache, DRAM, disk.* Morgan Kaufmann, 2010.

[55] Jayasena, S., Amarasinghe, S., Abeyweera, A., Amarasinghe, G., De Silva, H., Rathnayake, S., Meng, X., and Liu, Y. Detection of false sharing using machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC 13* (2013).

[56] Kandemir, M., Zhao, H., Tang, X., and Karakoy, M. Memory row reuse distance and its role in optimizing application performance. In *Performance Evaluation Review* (2015), vol. 43, pp. 137–149.

[57] Khan, T. A., and Pokam, G. Huron : Hybrid False Sharing Detection and Repair. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019).

[58] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *Proceedings - International Symposium on Computer Architecture* (2014), 361–372.

[59] Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M. S., and Nagel, W. E. The Vampir Performance analysis tool-set. *Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing* (2008), 139–155.

[60] Kozyrakis, C. Phoenix Project: Shared-memory implementation of Google's MapReduce model. https://github.com/kozyraki/phoenix/tree/master/phoenix-2.0.

[61] Lachaize, R., Lepers, B., and Quéma, V. MemProf: A Memory Profiler for NUMA Multicore Systems. *Proceedings of the 2012 USENIX Annual Technical Conference* (2012).

[62] Lawrence Livermore National Laboratory. Llnl squoia benchmarks. http://asc.llnl.gov/sequoia/benchmarks.

[63] LeCun, Y., Cortes, C., and Burges, C. The Mnist Database of Handwritten Digits. http://yann.lecun.com/exdb/mnist/, 2016.

[64] Li, A., Liu, W., Kristensen, M. R., Vinter, B., Wang, H., Hou, K., Marquez, A., and Song, S. L. Exploring and analyzing the real impact of modern on-package memory on hpc scientific kernels. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017* (2017).

[65] Linux. Perf. https://perf.wiki.kernel.org/index.php/Main_Page.

[66] Liu, T., and Berger, E. D. SHERIFF: Precise Detection and Automatic Mitigation of False Sharing. *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (2011).

[67] Liu, T., and Liu, X. Cheetah: detecting false sharing efficiently and effectively. In *Proceedings of the International Symposium on Code Generation and Optimization* (2016).

[68] Liu, T., Tian, C., Hu, Z., and Berger, E. D. PREDATOR: Predictive False Sharing Detection. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2014).

[69] Liu, X., and Mellor-Crummey, J. A data-centric profiler for parallel programs. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC '13* (2013).

[70] Liu, X., and Mellor-Crummey, J. Pinpointing data locality bottlenecks with low overhead. In *ISPASS 2013 - IEEE International Symposium on Performance Analysis of Systems and Software* (2013), pp. 183–193.

[71] Liu, X., and Mellor-Crummey, J. A tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2014), 259–272.

[72] Liu, X., Sharma, K., and Mellor-Crummey, J. ArrayTool: A Lightweight Profiler to Guide Array Regrouping. *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (2014), 405–416.

[73] Liu, X., and Wu, B. ScaAnalyzer: a tool to identify memory scalability bottlenecks in parallel programs. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC'15* (2015).

[74] Luo, L., Sriraman, A., Fugate, B., Hu, S., Pokam, G., Newburn, C. J., and Devietti, J. LASER: Light, Accurate Sharing dEtection and Repair. *Proceedings - International Symposium on High-Performance Computer Architecture* (2016), 261–273.

[75] Majo, Z., and Gross, T. R. (Mis) Understanding the NUMA Memory System Performance of Multithreaded Workloads. In *IEEE International Symposium on Workload Characterization (IISWC)* (2013), pp. 11–22.

[76] Marin, G., and Mellor-Crummey, J. Pinpointing and exploiting opportunities for enhancing data reuse. *ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and Software* (2008), 115–126.

[77] Mario, J. C2C - False Sharing Detection in Linux Perf. https://joemario.github.io/blog/2016/09/01/c2c-blog, 2016.

[78] Martonosi, M., Gupta, A., and Anderson, T. MemSpy: Analyzing Memory System Bottlenecks in Programs. *ACM SigMETRICS & PERFORMANCE* (1992).

[79] Maurice, C., le Scouarnec, N., Neumann, C., Heen, O., and Francillon, A. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Lecture Notes in Computer Science* (2015), vol. 9404, pp. 48–65.

[80] Mazaheri, A., Jannesari, A., Mirzaei, A., and Wolf, F. Characterizing loop-level communication patterns in shared memory. *Proceedings of the International Conference on Parallel Processing* (2015), 759–768.

[81] Mazaheri, A., Wolf, F., and Jannesari, A. Unveiling thread communication bottlenecks using hardware-independent metrics. *ACM International Conference Proceeding Series* (2018).

[82] McCalpin, John D. STREAM benchmark. http://www.cs.virginia.edu/stream/.

[83] McCurdy, C., and Vetter, J. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. *ISPASS 2010 - IEEE International Symposium on Performance Analysis of Systems and Software* (2010), 87–96.

[84] McCurdy, C., Vetter, J. S., Worley, P., and Maxwell, D. Memphis on a Cray XT: Pinpointing Memory Performance Problems on Cray Platforms. *Cray Users Group Conference (CUG 2011)* (2011).

[85] McDonald, N. libdes:a c++ discrete event simulation framework. https://github.com/nicmcd/libdes.

[86] Mi, W., Feng, X., Xue, J., and Jia, Y. Software-hardware cooperative DRAM bank partitioning for chip multiprocessors. *Lecture Notes in Computer Science 6289 LNCS*, 2005 (2010), 329–343.

[87] Molka, D., Hackenberg, D., Schöne, R., and Nagel, W. E. Cache coherence protocol and memory performance of the Intel Haswell-EP architecture. *Proceedings of the International Conference on Parallel Processing* (2015), 739–748.

[88] Molka, D., Schöne, R., Hackenberg, D., and Nagel, W. E. Detecting Memory-Boundedness with Hardware Performance Counters. *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering - ICPE '17* (2017), 27–38.

[89] Muralidhara, S. P. S., Subramanian, L., Mutlu, O., Kandemir, M., and Moscibroda, T. Reducing memory interference in multicore systems via application-aware memory channel partitioning. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO* (2011), 374–385.

[90] Nanavati, M., Spear, M., Taylor, N., Rajagopalan, S., Meyer, D. T., Aiello, W., and Warfield, A. Whose cache line is it anyway?: Operating system support for live detection and repair of false sharing. *Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys 2013* (2013), 141–154.

[91] Nowak, A., and Bitzes, G. The overhead of profiling using PMU hardware counters. *openlab.web.cern.ch*, CERN Openlab Report (2014).

[92] Oh, B., Kim, N. S., Ahn, J., Li, B., Dreslinski, R. G., and Mudge, T. A load balancing technique for memory channels. In *ACM International Conference Proceeding Series* (2018).

[93] Pan, X., Gownivaripalli, Y. J., and Mueller, F. TintMalloc: Reducing Memory Access Divergence via Controller-Aware Coloring. *Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016* (2016), 363–372.

[94] Pan, X., and Jonsson, B. Modeling cache coherence misses on multicores. *ISPASS 2014 - IEEE International Symposium on Performance Analysis of Systems and Software* (2014), 96–105.

[95] Patterson, D. A., and Hennessy, J. *Computer Organisation and Design*, fourth ed. 2012.

[96] Pessl, P., Gruss, D., Maurice, C., Schwarz, M., and Mangard, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. *USENIX Security Symposium* (2016).

[97] Pesterev, A. *Locating Cache Performance Bottlenecks Using Data Profiling*. PhD thesis, Massachusetts Institute of Technology, 2010.

[98] Pesterev, A., Zeldovich, N., Morris, R. T., and Orlando, T. P. Locating cache performance bottlenecks using data profiling. In *EuroSys '10* (2010), p. 335.

[99] Pop, A., and Cohen, A. OpenStream: Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs Antoniu. *IACM Transactions on Architecture and Code Optimization* (2013).

[100] Qiao, Y., Hashimoto, K., Eriguchi, A., Wang, H., Wang, D., Tsuruoka, Y., and Taura, K. Parallelizing and optimizing neural Encoder Decoder models without padding on multi-core architecture. *Future Generation Computer Systems* (2018).

[101] Reinders, J. Processor tracing. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[102] Roth, M., Best, M. J., Mustard, C., and Fedorova, A. Deconstructing the overhead in parallel applications. *Proceedings - 2012 IEEE International Symposium on Workload Characterization, IISWC 2012* (2012), 59–68.

[103] Roy, P., and Liu, X. StructSlim: A Lightweight Profiler to Guide Structure Splitting. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (2016), pp. 36–46.

[104] Roy, P., Song, S. L., Krishnamoorthy, S., and Liu, X. Lightweight Detection of Cache Conflicts. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (2018), pp. 200–213.

[105] Samsung Electronics. M386A8K40BM1 Serial Presence Detect, 2015.

[106] Samsung Electronics. M386A8K40BM1 Datasheet, 2017.

[107] Seaborn, M. How physical addresses map to rows and banks in dram. http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html.

[108] Selva, M., Morel, L., and Marquet, K. Numap: A portable library for low-level memory profiling. *Proceedings - 2016 16th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2016* (2017), 55–62.

[109] Sharma, S. D., and Dagenais, M. Hardware-assisted instruction profiling and latency detection. *The Journal of Engineering 2016* (2016), 367–376.

[110] Song, W., Choi, H., Kim, J., Kim, E., Kim, Y., and Kim, J. Pikit: A new kernel-independent processor-interconnect rootkit. *Proceedings of the 25th USENIX Security Symposium* (2016), 37–51.

[111] Southern, G., and Renau, J. Deconstructing PARSEC Scalability. *11th Annual Workshop on Duplicating, Deconstructing and Debunking* (2015).

[112] Spin Developers. Spin sources. http://spinroot.com/spin/Src/index.html.

[113] Srinivas, M., Sinharoy, B., Eickemeyer, R. J., Raghavan, R., Kunkel, S., Chen, T., Maron, W., Flemming, D., Blanchard, A., Seshadri, P., Kellington, J. W., Mericas, A., Petruski, A. E., Indukuru, V. R., and Reyes, S. IBM POWER7 performance modeling, verification, and evaluation. *IBM Journal of Research and Development 55*, 3 (2011).

[114] Sudan, K., Chatterjee, N., Nellans, D., Awasthi, M., Balasubramonian, R., and Davis, A. Micro-pages: Increasing DRAM efficiency with locality-aware data placement. In *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS* (2010), pp. 219–230.

[115] Tallent, N. R., John, M. M. C., and Fagan, M. W. Binary analysis for measurement and attribution of program performance. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2009), 441–452.

[116] Taura, K. Mnist application. https://www.eidos.ic.i.u-tokyo.ac.jp/~tau/lecture/paralleldistributed/2016/examples/18mnist/, 2016.

[117] Terpstra, D., Jagode, H., You, H., and Dongarra, J. Collecting Performance Data with PAPI-C. *Tools for High Performance Computing* (2010).

[118] Thomadakis, M. E. The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms. Tech. rep., 2011.

[119] Trahay, F. numap library. https://github.com/numap-library/numap.

[120] Trahay, F., Selva, M., Morel, L., and Marquet, K. NumaMMA: NUMA memory analyzer. In *International Conference on Parallel Processing* (2018).

[121] Treibig, J., Hager, G., and Wellein, G. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. *Proceedings of the International Conference on Parallel Processing Workshops* (2010), 207–216.

[122] Viswanathan, V., Kumar, K., Willhalm, T., Lu, P., Filipiak, B., and Sakthivelu, S. Intel Memory Latency Checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker, 2018.

[123] Weaver, V. M. Self-monitoring overhead of the Linux perf_ event performance counter interface. *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2015), 102–111.

[124] Weidendorfer, J., Kowarschik, M., and Trinitis, C. A tool suite for simulation based analysis of memory access behavior. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 3038* (2004), 440–447.

[125] Weyers, B., Terboven, C., Schmidl, D., Herber, J., Kuhlen, T. W., Müller, M. S., and Hentschel, B. Visualization of Memory Access Behavior on Hierarchical NUMA Architectures. In *Proceedings of VPA 2014: 1st Workshop on Visual Performance Analysis* (2015), pp. 42–49.

[126] WikiChip. Broadwell uarch. https://en.wikichip.org/wiki/intel/microarchitectures/broadwell_(client).

[127] Xu, H., Wen, S., Gimenez, A., Gamblin, T., and Liu, X. Dr-bw mini-benchmarks source code. https://github.com/xuhao417347761/DR-BW/tree/master/mini-benchmarks.

[128] Xu, H., Wen, S., Gimenez, A., Gamblin, T., and Liu, X. DR-BW: Identifying Bandwidth Contention in NUMA Architectures with Supervised Learning. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS* (2017).

[129] Yasin, A. A Top-Down Method for Performance Analysis and Counters Architecture. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2014), pp. 35–44.

[130] Yoon, H., Meza, J., Ausavarungnirun, R., Harding, R., and Mutlu, O. Row Buffer Locality-Aware Data Placement in Hybrid Memories. *SAFARI Technical Report 005* (2011).

[131] Yu, C., Roy, P., Bai, Y., Yang, H., and Liu, X. LWPTool: A Lightweight Profiler to Guide Data Layout Optimization. *IEEE Transactions on Parallel and Distributed Systems 29*, 11 (2018), 2489–2502.

[132] Yun, H., Mancuso, R., Wu, Z. P., and Pellizzoni, R. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time Technology and Applications* (2014), IEEE, pp. 155–166.

[133] Zhang Xianyi et al. Openblas an optimized blas library. https://www.openblas.net.

[134] Zhao, Q., Koh, D., Raza, S., Amarasinghe, S., Bruening, D., and Wong, W. F. Dynamic cache contention detection in multi-threaded applications. *ACM SIGPLAN Notices 46*, 7 (2011), 27–37.