Detecting Memory-Boundedness with Hardware Performance Counters

Daniel Molka Robert Schöne Daniel Hackenberg

Wolfgang E. Nagel

Center for Information Services and High Performance Computing
Zellescher Weg 12-14, Dresden, Germany
{daniel.molka, robert.schoene, daniel.hackenberg, wolfgang.nagel}@tu-dresden.de

ABSTRACT

Modern processors incorporate several performance monitoring units, which can be used to count events that occur within different components of the processor. They provide access to information on hardware resource usage and can therefore be used to detect performance bottlenecks. Thus, many performance measurement tools are able to record them complementary to information about the application behavior. However, the exact meaning of the supported hardware events is often incomprehensible due to the system complexity and partially lacking or even inaccurate documentation. For most events it is also not documented whether a certain rate indicates a saturated resource usage. Therefore, it is usually difficult to draw conclusions on the performance impact from the observed event rates. In this paper, we evaluate whether hardware performance counters can be used to measure the capacity utilization within the memory hierarchy and estimate the impact of memory accesses on the achieved performance. The presented approach is based on a small selection of micro-benchmarks that constantly stress individual components in the memory subsystem, ranging from caches to main memory. These workloads are used to identify hardware performance counters that provide good estimates for the utilization of individual components in the memory hierarchy. However, since access latencies can be interleaved with computing instructions, a high utilization of the memory hierarchy does not necessarily result in low performance. We therefore also investigate which stall counters provide good estimates for the number of cycles that are actually spent waiting for the memory hierarchy.

Keywords

hardware performance counters, benchmarking, performance analysis

ICPE'17, April 22 - 26, 2017, L'Aquila, Italy

ISBN 978-1-4503-4404-3/17/04…$15.00

DOI: *http://dx.doi.org/10.1145/3030207.3030223*

1. INTRODUCTION

High performance computing (HPC) is an indispensable tool that is required to obtain new insights in many scientific disciplines. While HPC systems are getting more and more powerful from year to year, scientific applications typically are not able to fully utilize this potential. Utilization levels of 10% and lower are not uncommon [19].

Modern microprocessors feature multiple levels of cache—small and fast buffers for frequently accessed data—that improve the performance of memory accesses. Still, memory access latencies often account for a significant portion of the average cycles per instruction [4]. Since caches and the memory interface can be shared between multiple cores of a processor, the contention of shared resources can limit the scalability of parallel applications. Moreover, the physical memory is typically distributed among multiple processors of a system, leading to varying performance depending on the distance to the accessed data. These non-uniform memory access (NUMA) characteristics influence the performance and scalability of parallel applications [15]. The performance of memory accesses is also affected by the cache coherence protocols [18]. Overall, the complexity of contemporary computer systems results in various potential bottlenecks that can cause application performance penalties.

Our understanding of the application performance can be improved by determining the limitations that are caused by the various components of the memory hierarchy. This can be achieved with the help of performance monitoring units (PMUs), which are included in many modern processors. Using PMUs to detect memory related performance issues is common practice [5, 13, 25, 26]; [10, Appendix B.3]. However, the available events are often specific to a certain processor generation and their meaning is not always obvious. Two things are required to decide if the observed performance counter event rates are significant: The attribution of events to the utilization of individual components, and reference values for peak event rates.

In this paper we contribute a portable approach for the identification of hardware performance events that can be used to detect performance problems that are caused by memory accesses. We describe a method to derive metrics for the utilization of individual components in the memory subsystem and the memory-boundedness of applications. The applicability of our methodology is demonstrated on a contemporary dual-socket server system.

2. BACKGROUND AND RELATED WORK

Figure 1 depicts the basic structure of contemporary multi-processor systems. Typically each core has dedicated level-one caches. Per-core level-two caches also are a widely-used feature. In contrast, last level cache and memory interface are typically shared by multiple cores. Each level in the memory hierarchy has different characteristics. The level-one caches support high data rates and have very low latencies. However, their capacity is very low. Each further level in the memory hierarchy provides higher capacities along with lower data rates and higher latencies. The compute nodes of HPC systems often contain multiple processors in order to increase the compute power and memory capacity. However, the communication via the processor interconnects results in a higher latency for remote cache and memory accesses and the data rates of the links are limited [17].

Caches are transparent for the software, i.e., they are no directly addressable memory. Instead, caches contain duplicates of small blocks from main memory. Which data is placed in the caches is determined by the hardware at runtime based on the observed memory accesses. The used heuristics include keeping data that has already been accessed in cache in anticipation of further accesses to the same locations as well as various techniques that recognize regular access patterns and prefetch data that will probably be accessed soon. Furthermore, multiple copies of the same memory location can exist concurrently in different caches. Cache coherence protocols ensure that writes performed by one core eventually become visible to all cores in order to maintain the impression of a single copy. However, this often requires that cached data is invalidated. Due to the inherent unpredictability of the replacement, prefetch, and coherence mechanisms it is virtually impossible to determine the cache usage of an application using static code analysis.

Caches significantly reduce the average memory latency. However, time consuming main memory accesses are not avoided entirely and cache accesses, especially in case of large last level caches, also have considerable latencies. Therefore, processors used in HPC (excluding accelerators) typically have well-developed out-of-order execution capabilities to fill the remaining waiting times with useful work. Furthermore, the load-store units comprise multiple load and store buffers in order to facilitate multiple concurrently outstanding requests instead of processing memory accesses one after another. A sufficient number of concurrently outstanding requests is a prerequisite for achieving high data rates [14]. The caches typically support multiple outstanding misses as well to avoid being blocked by a single cache miss. The ability to continue the program execution while memory accesses are pending can significantly improve the performance. However, this and the overlapping of multiple accesses further complicates the assessment of memory related performance losses as only a fraction of the individual access latencies causes delays in the execution.

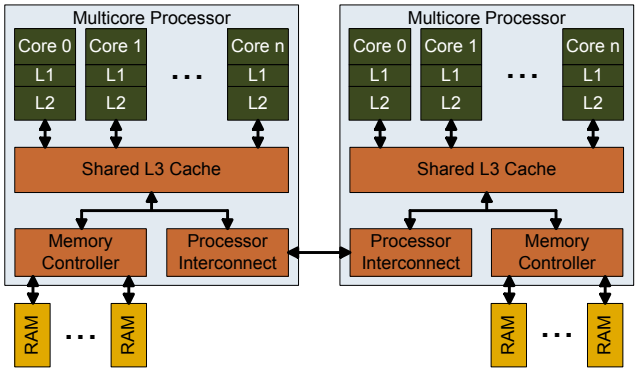


Figure 1: Schematic representation of contemporary multi-processor systems. Each processor typically contains multiple cores as well as various shared resources. The processors are typically connected to each other via point-to-point connections.

The achievable application performance can be limited by various components. Performance monitoring units (PMUs) that collect information about the usage of these components are included in many contemporary processors [2, Section 2.7]; [11, Vol. 3, Section 18.1]. Typically, each core has a number of dedicated PMUs that can be used to count events from this core, e.g., the number of executed instructions. Multi-core processors often implement additional PMUs that monitor shared resources [2, Section 2.7.2]; [8]. Typically, each PMU contains two registers, the control register and the performance monitoring counter (PMC) register. The former is used to *specify* the events that are recorded, the latter incorporates an *accumulator* that is incremented with every occurrence of an event. A widely-used tool for recording performance counter data is the Performance API (PAPI) [24], which defines a standardized interface for accessing PMUs of various processor architectures. Many performance measurement tools, e.g., *HPCToolkit* [1], and *Score-P* [12], use it to record performance counters in addition to information about the application behavior.

PMUs are primarily intended for verification purposes [27]. However, they can be used to detect performance issues as well [10, Appendix B]. This has also been studied extensively by the scientific community: Eranian [5] shows that PMUs can in principle be used to detect performance problems, although no systematic analysis of the available events is included in his study. Treibig et al. [25] describe common bottlenecks of specific applications and how they can be detected with PMUs. Their approach requires events that correctly represent the bandwidth utilization in order to diagnose memory related performance problems. Levinthal [13] and Yasin [26] present procedures that identify bottlenecks in the memory hierarchy using PMUs. However, their studies do not include any verification whether the PMUs count the selected events accurately. Yoo et al. [27] use micro-benchmarks to correlate known performance pathologies, e.g., performing random accesses on large data sets, with PMU events on Westmere-EP and Sandy Bridge HE test systems. The approach does not only allow to detect potential performance issues, it also categorizes them. However, it is inherently limited to the known types of problems. While also relying on micro-benchmarks, our approach is different as we focus on the potential bottlenecks in the hardware instead of unfavorable properties of the software. Palomares lists PMC events that correlate to cache and memory traffic in [20, Table 5.3]. However, the given events do not distinguish read and write accesses. Furthermore, the listed DRAM events for the Sandy Bridge, Ivy Bridge, and Haswell architectures only measure the DRAM accesses per package. In our work, we also investigate if the core PMUs can be used to measure the bandwidth usage of individual cores.

3. METHODOLOGY

Each component in the memory hierarchy can limit the achievable application performance, either due to limited bandwidths or due to access latencies that stall the execution. Therefore, the utilization level of the available bandwidths as well as the percentage of cycles that is spent waiting for the memory hierarchy are good indicators for memory related performance problems. In this section we describe how these metrics can be derived from performance counters. Performance counters are architecture specific, i.e., the evaluation of their significance has to be repeated for every new micro-architecture. Therefore, we propose a portable approach based on a small selection of micro-benchmarks. We demonstrate its applicability on a dual socket Haswell-EP system. A similar study for the Sandy Bridge micro-architecture is presented in [16, Section 5.3].

3.1 Bandwidth Utilization

Memory intensive applications with sequential or otherwise regular memory access patterns are typically bandwidth-bound. As the required memory addresses are predictable, there are enough concurrent requests to fill the load or store buffers and the request queues for accesses to lower levels in the memory hierarchy. This leads to a high utilization of the data paths. The bandwidth utilizations within the memory subsystem are easy to understand metrics that indicate if the application behaves as expected.

In order to determine the utilization level for each component, one needs to know the respective peak bandwidth as well as the bandwidth usage of the application. A component's theoretical peak bandwidth can be calculated by multiplying the width of the corresponding data paths with the data rates (operating frequencies), which are typically documented by the hardware vendors. However, the practically achievable bandwidths can be significantly lower [17]. Micro-benchmarks that stress individual components provide much better reference values for the achievable performance. The achieved bandwidth usage of applications needs to be recorded at runtime. Performance monitoring units provide information about the utilization of various components. However, they typically count accesses to certain levels in the memory hierarchy or packages that are transferred between processors instead of directly measuring the used bandwidth. The amount of data that is transferred per reported event is not necessarily known. Furthermore, prefetchers can request data before it is actually accessed, which may disguise memory accesses. Therefore, it can be difficult to judge if the observed event rates, e.g., for cache misses, are indicative of a performance problem.

Our methodology requires bandwidth benchmarks that perform sequential read and write accesses, use configurable data set sizes, and support NUMA-aware memory allocation. Such benchmarks can be used to identify performance counters that correlate with the utilization of individual components and determine their respective maximal event rates, which represent an utilization of 100 %. The proposed workflow comprises the following steps:

1. Identify events that show high event rates if data is transferred between the core and the L1 cache. Load and store instructions with multiple widths are used in this step to test if the usage of SIMD instructions can be recognized.
2. For each further level in the memory hierarchy: identify events that show high event rates if data is read from or written back there. This step is performed using the widest load and store instructions that are available. It is important to identify events that consider the read for ownership (RFO) requests in case of writes in order to capture all transfers to the L1 cache.
3. Allocate memory from another NUMA node in order to identify events that show high event rates if data is located in remote memory.
4. Determine the overlap between the identified counters, i.e., check if events that correlate well with the number of accesses to a certain location include other accesses as well. If possible, use additional PMU events to compensate the overlap in order to measure the number of accesses independently for every location.
5. Derive upper bounds for the event rates of the identified counters based on the measured performance and the observed number of events per memory access.

There are per core limits for the achievable bandwidths for each level in the memory hierarchy as well as limited aggregate bandwidths for the shared resources [22]. Therefore, the analysis is performed once using a single core and once using all cores that share a resource in order to determine the respective upper bounds for the associated hardware performance counters.

3.2 Memory Related Stalls

Applications with irregular memory access patterns, e.g., dereferencing chains of pointers when traversing linked lists or trees, may not generate enough concurrently outstanding requests to fully utilize the data paths. Nevertheless, such applications are clearly limited by the performance of memory accesses as well. Therefore, considering the bandwidth utilization is not sufficient to detect all memory related performance issues. However, multiplying the number of accesses to each location with the respective latency would overestimate the total waiting time in the majority of cases. Furthermore, a high bandwidth utilization is not necessarily a performance problem. As long as the out-of-order engine can keep the execution units busy with useful work, using all the available bandwidth does not pose a bottleneck. Therefore, we also search for performance counters that measure memory related waiting times, i.e., periods without any useful computation, in order to quantify the performance impact of memory accesses.

A low instruction throughput without being bandwidth bound indicates that there are high latency operations. This can however be caused by slow arithmetic instructions like *div* or *sqrt* as well as high memory accesses latencies. Fortunately, performance counters for various types of stall cycles also are a widely-used feature in contemporary processors. With our methodology it is possible to check if memory related stalls can be distinguished from other stall reasons with the existing PMU events. Furthermore, it can be checked if stalls that are caused by high access latencies and stalls between consequtive data transfers in bandwidth limited scenarios can be differentiated.

Our methodology requires a pointer-chasing benchmark that supports configurable data set sizes. This benchmark needs to be modified in order to be able to distinguish memory related stalls from other stall reasons. In a first modifica-

tion, independent arithmetic instructions have to be added between the memory accesses. In another modified version, the arithmetic operations have to be added in a way that results in a single dependency chain that comprises all operations, e.g., by repeated additions of 0 or multiplications by 1. The workflow to identify meaningful counters for memory related stall cycles comprises the following steps:

1. Use standard latency benchmark to identify stall counters that comprise all latency related stalls
2. Use modified pointer-chaising benchmarks to check if there are counters that distinguish memory related stalls from other stall reasons. In case of independent operations between the loads the runtime stays constant. A suitable counter for memory related stalls needs to deduct the cycles that perform computations from the reported number of stalls. If there is a single dependency chain, it should report the same values as for the unmodified latency benchmark.
3. Check if there are additional counters that correlate with the number of total stall cycles reported for the bandwidth benchmarks but do not cover the stalls during the latency benchmark.

If useful counters are found in step two, they can be used to check if delays caused by memory accesses account for a significant portion of the runtime. Depending on the outcome of step three, the memory-bound fraction can possibly be subdivided into latency-bound and bandwidth-bound parts.

3.3 Benchmarks

We use *x86-membench* [16, Chapter 3], which is particularly suitable to stress individual components in the memory subsystem of systems with 64 bit x86 processors. Furthermore, x86-membench can record hardware performance counters in addition to the latency and bandwidth measurements via an integrated PAPI instrumentation. The constant workloads in combination with the integrated hardware monitoring are used to identify performance counters that provide good estimates for the utilization of the individual components.

The correlation of hardware performance counters with the utilization of the memory hierarchy is evaluated using the load and store variants of *x86-membench's* throughput kernel [16, Section 3.5.4]. These benchmarks measure the achievable bandwidths for different levels in the memory hierarchy by repeatedly accessing a buffer. The size of the buffer determines the level in the memory hierarchy that is evaluated. Memory can be allocated from specific NUMA nodes. In order to determine suitable performance counters for the bandwidth that is consumed per core the throughput benchmark has to be configured as follows[1]:

- BENCHIT_KERNEL_CPU_LIST is set to "0", which restricts the measurement to CPU 0.
- A CPU from another processor that is directly connected to the processor that contains CPU 0 is entered in BENCHIT_KERNEL_MEM_BIND.
- BENCHIT_KERNEL_MIN is set to 50% of the L1 data cache size or lower.
- BENCHIT_KERNEL_MAX is set to at least ten times the LLC size.
- BENCHIT_KERNEL_ALLOC is initially set to "L" (localalloc) in order to evaluate the local memory hierarchy. It is changed to "B" (bind-to-core) in order to examine remote memory accesses.
- BENCHIT_KERNEL_INSTRUCTION is configured to perform loads or stores with different widths as required, e.g., "avx_load_pd" for 256 bit loads.

In order to investigate shared resources in multi-core processors, the benchmark configuration is changed as follows:

- All CPUs that belong to the first socket are entered in BENCHIT_KERNEL_CPU_LIST.
- BENCHIT_KERNEL_MEM_BIND contains all CPUs from another socket that is directly connected to the first socket.

The counter that most accurately represents the delay caused by memory accesses can be identified with a slightly adapted version of *x86-membench's* latency benchmark [16, Section 3.5.1]. This benchmark repeatedly dereferences a pointer in the register *RBX*. It represents a worst case scenario with only one outstanding memory request at a time, a hardly predictable access pattern, and no other instructions between the memory accesses. In order to investigate the effect of overlapping memory accesses and computation on the number of reported stall cycles, arithmetic instructions are added between the loads. There are two versions of the modified latency benchmark—one that adds multiplications without data dependencies to the workload and one with multiplications that are part of the dependency chain [16, Section 5.3.2]. In the former case the added multiplications use different registers (R8 – R15) and can therefore be reordered around the memory accesses. In the latter case the pointer itself is repeatedly multiplied by 1. Since all operations form a single dependency chain, they cannot be reordered. The parameters have to be configured as follows:

- one CPU from every processor is entered in BENCHIT_KERNEL_CPU_LIST
- settings for BENCHIT_KERNEL_MIN and BENCHIT_KERNEL_MAX are identical to the bandwidth measurements
- BENCHIT_KERNEL_ALLOC is set to "L" (localalloc)

4. EVALUATION OF HASWELL-EP

We conduct our experiments on a Bull SAS bullx R421 E4 system, which is described in Table 1. As depicted in Figure 2, each processor contains twelve cores with dedicated L1

Table 1: Dual socket Haswell-EP test system

Processors	2x Intel Xeon E5-2680 v3
Cores / threads	2x 12 / 2x 24
Core clock	2.5 GHz (2.1 GHz AVX freq. [9])
Uncore clock	up to 3.0 GHz
L1 / L2 cache	2x 32 KiB / 256 KiB per core
L3 cache	30 MiB per chip
Memory	128 GiB (8x 16 GiB) PC4-2133P-R
QPI speed	9.6 GT/s (38.4 GB/s)
Operating System	Ubuntu 16.04 LTS, kernel 4.4.0-21-generic

[1] defaults are used for the remaining parameters, see *https://fusionforge.zih.tu-dresden.de/plugins/mediawiki/wiki/benchit/index.php/X86membench*

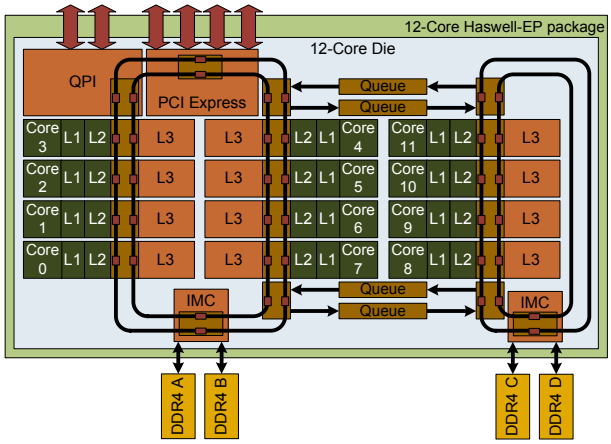


Figure 2: Xeon E5-2680 v3 package

and L2 caches as well as various shared resources. Each core has a dedicated performance monitoring unit that monitors the activity of the execution units, cache hits and misses, and so-called offcore requests, which supply data that is not found in the core's local L1 and L2 cache. Furthermore, there are multiple PMUs for the shared resources, which reside in the so-called *uncore* [8, Figure 1-2]. The L3 cache is partitioned into twelve slices. Each slice has a dedicated caching agent (CA) with an associated performance monitoring unit called C-Box. Memory accesses are monitored by the home agent (HA) counters that observe memory accesses on the coherence protocol level as well as the integrated memory (IMC) counters that record information related to the actual DRAM accesses. Since each processor contains two memory controllers, there are two HA and two IMC PMUs per processor. Data transfers between the processors are covered by the QPI counters—separately for each link.

We are using PAPI version 5.4.3 to access the PMUs. The recorded events are configured via x86-membench's BENCHIT_KERNEL_PAPI parameter. We use the event names as they are reported by the papi_native_avail command.

Haswell is an aggressive out-of-order micro-architecture [10, Section 2.2]. The large reorder window enables the processor cores to continue executing instructions while multiple outstanding memory requests are processed. However, main memory accesses that are not anticipated by the hardware prefetchers can take several hundreds of cycles and the bandwidths supported by the memory hierarchy are limited [18]. This eventually stalls the execution once a required resource, e.g., the load or store buffers, is fully used.

4.1 Bandwidth Usage of a Single Core

Since uncore counters record aggregate performance data that can hardly be attributed to actions of a single core, the evaluation is restricted to the core counters. However, there are still numerous memory related events.

The event *perf::L1-DCACHE-LOADS* counts all load instructions. This can be used to determine how many times the load ports are used. However, no distinction is made between loads of different widths. As long as the data set fits into the L1 cache, the number of reported *perf::L1-DCACHE-LOAD-MISSES* is close to zero. For larger data sets the number of events per load instruction depends on their width. One out of eight 64 bit loads generates a miss event. 128 bit loads cause a miss on every fourth access. In case of 256 bit wide loads every second load instruction misses the L1 cache. This means that the *perf::L1-DCACHE-LOAD-MISSES* increases by one for every cache line that is transferred to the L1 cache. Writes to lower cache levels also generate one load miss per cache line, i.e., RFO requests are included in the measurement. Therefore, the *perf::L1-DCACHE-LOAD-MISSES* provide a good estimate for the number of cache lines that are brought into the L1 cache. Analogous events for write accesses are available as well. The event *perf::L1-DCACHE-STORES* counts all write accesses. The event *perf::L1-DCACHE-STORE-MISSES* is not defined by the Linux kernel and is therefore not functional. However, the number of cache lines that are written back to lower levels of the memory hierarchy can be counted using the *L2_TRANS:L1D_WB*.

The remaining challenge is to find events that distinguish accesses to different memory hierarchy levels and local from remote memory accesses. As depicted in Figure 3, the *MEM_LOAD_UOPS_RETIRED* events are not suitable to determine the origin of the data in case of sequential loads. Half of the accesses hit the L1 cache or the line fill buffers (LFB). This can be explained by the fact that the benchmark performs two loads per cache line. Therefore, only every second access causes a data transfer from a lower level. The sum of the reported *L2_HIT*, *L3_HIT*, and *L3_MISS* events is equal to the number of *perf::L1-DCACHE-LOAD-MISSES*. However, the individual values do not correlate well with the number of cache lines that are delivered from the L2 cache, L3 cache, and main memory, respectively. The *L2_HIT* sub-event shows a non-negligible number of false positives if data is read from the L3 cache and main memory. Likewise, sequential memory accesses also cause a high number of *L3_HIT* events. Thus, using the *L3_MISS* events can severely underestimate the number of memory accesses. The discrepancy between the values reported by the performance counters and the actual number of accesses is presumably caused by the hardware prefetchers. Furthermore, reads that are caused by read for ownership (RFO) requests are not covered by *MEM_LOAD_UOPS_RE-TIRED*.

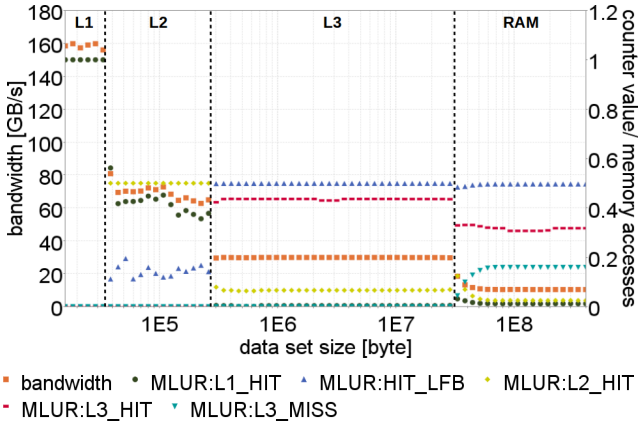


Figure 3: Date source according to MEM_LOAD_UOPS_RETIRED (MLUR) events: This event distinguishes load operations with respect to the origin of the accessed data. However, it does not correctly represent the number of data transfers from the individual levels in the memory hierarchy.

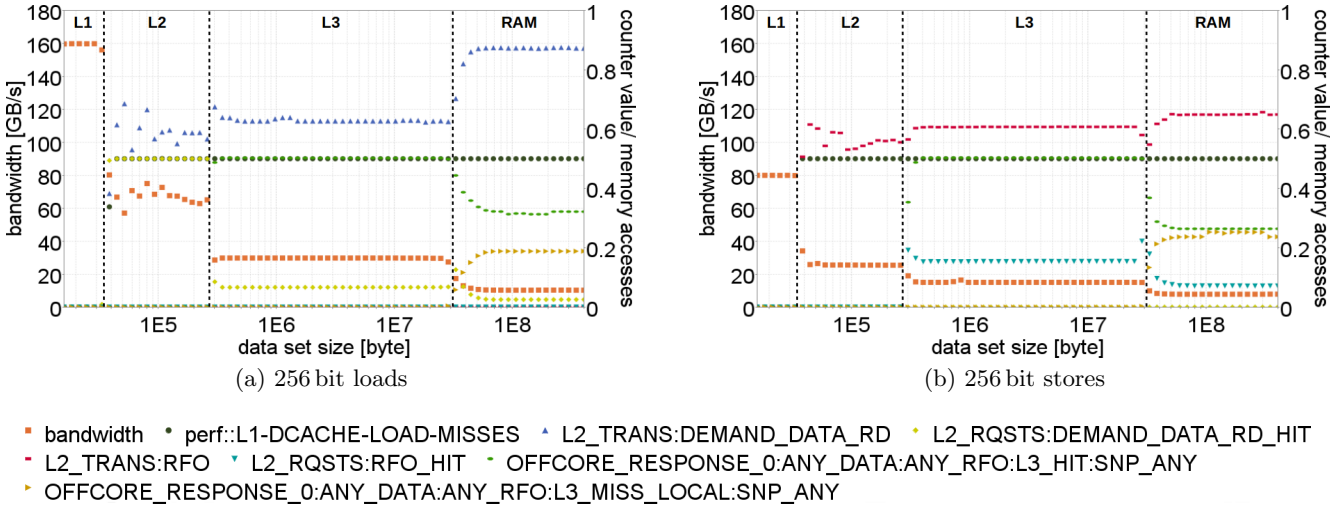
Figure 4 depicts performance counter readings that subdivide the L1 misses according to the data's prior location within the memory hierarchy. If the data is located in the L2 cache, loads cause *L2_TRANS:DEMAND_DATA_RD* and *L2_RQSTS:DEMAND_DATA_RD_HIT* events while stores generate *L2_TRANS:RFO* events. The event rates are close to the number of lines requested by the L1 cache as reported by *perf::L1-DCACHE-LOAD-MISSES*, i.e., approximately one event per accessed cache line is recorded. Surprisingly, the *L2_RQSTS:RFO_HIT* event does not record any RFO requests as long as all data fits into the L2 cache. In contrast, *L2_RQSTS:ALL_RFO − L2_RQSTS:RFO_MISS* correctly represents the number of RFOs that hit the L2 cache (not depicted). For data sets that exceed the L2 capacity the values reported by *L2_TRANS* and *L2_RQSTS* differ significantly. Apparently, the sub-events of L2_TRANS also capture accesses to lower levels of the memory hierarchy. However, the values are significantly higher than the corresponding number of misses in the L1 cache, especially in case of loads. This indicates that additional cache lines from the L2 cache are requested by the L1 cache's prefetchers while waiting for the data from lower cache levels. The L2_RQSTS readings seem to only consider cache lines that are actually delivered by the L2 cache. However, as it is the case for the *MEM_LOAD_UOPS_RETIRED* events, a significant number of false positives are reported when data is streamed from lower levels in the memory hierarchy.

The *OFFCORE_RESPONSE* events observe requests that miss in the L2 cache and provide numerous filters to isolate data transfers from a certain location. Events are specified in the following format: *OFFCORE_RESPONSE_{0|1}<request type><response type>* where the *response type* is either *ANY_RESPONSE* or *<supplier><snoop>*. The *request type* is set to *ANY_DATA:ANY_RFO* in order to include read only requests as well as the RFOs caused by stores. The *supplier* field in the *response type* can be used to distinguish L3 cache and main memory accesses. If it is set to *L3_HIT*, all L3 accesses that hit cache lines in state *Modified*, *Exclusive*, *Shared*, or *Forward* are considered. The number of cache lines requested from local main memory can be counted using *L3_MISS_LOCAL* as supplier. Remote accesses can be counted via *L3_MISS_REMOTE_HOP{0|1|2P}* (not depicted). The *SNP_ANY* setting used for the *snoop* field includes cache lines that are forwarded from other caches in the measurement (not relevant in the used benchmark). *SNP_MISS:SNP_NO_FWD: SNP_NOT_NEEDED* can help to exclude forwarded cache lines from the measurement. However, *SNP_ANY* is the most cautious choice for measuring the utilization since on-chip transfers include a L3 lookup. Furthermore, cache lines that are forwarded from caches in other processors are typically delivered from memory as well or involve a directory lookup in the home node.

If data is delivered by the L3 cache, there are as many *L3_HIT* events as there are *L1-DCACHE-LOAD-MISSES*. In contrast, the number of *L3_MISS_LOCAL* events does not match the number of L1 cache misses if data is delivered from main memory. Therefore, the proportion of main memory accesses is severely underestimated by the *OFFCORE_RESPONSE* events. However, the sum of the L3 hit and L3 miss events is very close to the number of L1 misses in both cases, so the number of cache line transfers from the uncore to each core can be measured quite accurately.

Figure 4(b) shows a write bandwidth measurement, but the recorded events only cover the read for ownership transfers that place the data in the L1 cache prior to the modification. Indicators for the write backs are depicted in Figure 5. The total number of cache lines that are written back from the L1 cache can be counted via the *L2_TRANS:L1D_WB* event. Write backs from the L2 cache to lower levels of the memory hierarchy are represented by the values reported by *L2_TRANS:L2_WB*. Core counter events that indicate the number of write backs to main memory have not been found.

Table 2 shows the bandwidths that can be achieved by a single thread for read and write accesses to the local memory hierarchy as well as remote memory. The measured bandwidths are used to calculate the resulting number of transfers per second. The maximal event rates are derived from the performance counter readings presented in this section.



(a) 256 bit loads

(b) 256 bit stores

Figure 4: The Utilization of the L2 cache can be measured via the L2_TRANS and L2_RQSTS events. L3 cache and main memory accesses can be recorded using the OFFCORE_RESPONSE events. The correlation with the amount of accessed data is far from perfect. However, it is possible to gain important insight.

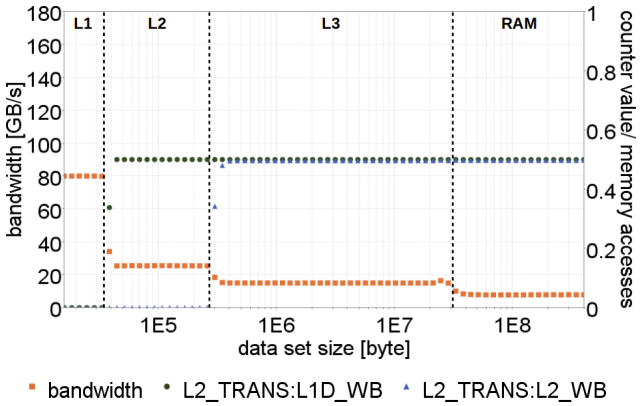


Figure 5: Write bandwidth (256 bit stores) and write back events: The L2_TRANS:L1D_WB and L2_TRANS:L2_WB events represent write backs from the L1 and L2 cache, respectively.

Most of them are identical to the number of transfers. Unfortunately, the number of transfers per second cannot be used to derive the bandwidth utilization of the L1 cache. With 64 bit instructions it is possible to reach the maximal event rates while using only 25% of the available bandwidth. However, this could still be defined as 100% load as all L1 load or store ports are active each cycle. The numbers reported by the *L2_TRANS:DEMAND_DATA_RD_HIT* and *L2_TRANS:RFO* events are higher than the actual number of requested cache lines that miss the L1 cache—presumably as some prefetcher requests are included as well. Furthermore, the *OFFCORE_RESPONSE* counters for L3 misses underestimate memory accesses. Nevertheless, the maximal event rates provide reference values for the 100% utilization of the corresponding components, which allows us to estimate the degree of capacity utilization for application runs.

4.2 Utilization of Shared Resources

The last level cache, the memory bandwidth, and the links between the processors are potential bottlenecks that may limit the application performance. In this section it is evaluated if the processor's uncore performance counters [8] can be used to measure the utilization of these shared resources. The uncore performance monitoring is implemented by multiple per-component performance monitoring units (also called "boxes") [8, Figure 1-2]. Uncore events are specified in the format: *hswep_unc_<comp>::<event name>* where *comp* selects a component, e.g. a L3 slice (C-Box), and the *event name* specifies the events that are counted. The *"hswep_unc_"* prefix is omitted here. *X86-membench* records performance counters only on one CPU and derives the event ratios by dividing the recorded number of events by the number of memory accesses performed by this CPU. This is sufficient for the collection of core counters since the workload is homogeneous, i.e., all cores would report very similar values. It also avoids conflicts between the CPUs if uncore counters are recorded as only a single CPU is accessing the uncore PMUs. However, in this case all events are attributed to a single CPU, which needs to be considered in the interpretation of the results. In case of the Xeon E5-2680 v3 processor the uncore is shared by twelve cores, i.e., the reported event ratios have to be divided by twelve to get the correct result.

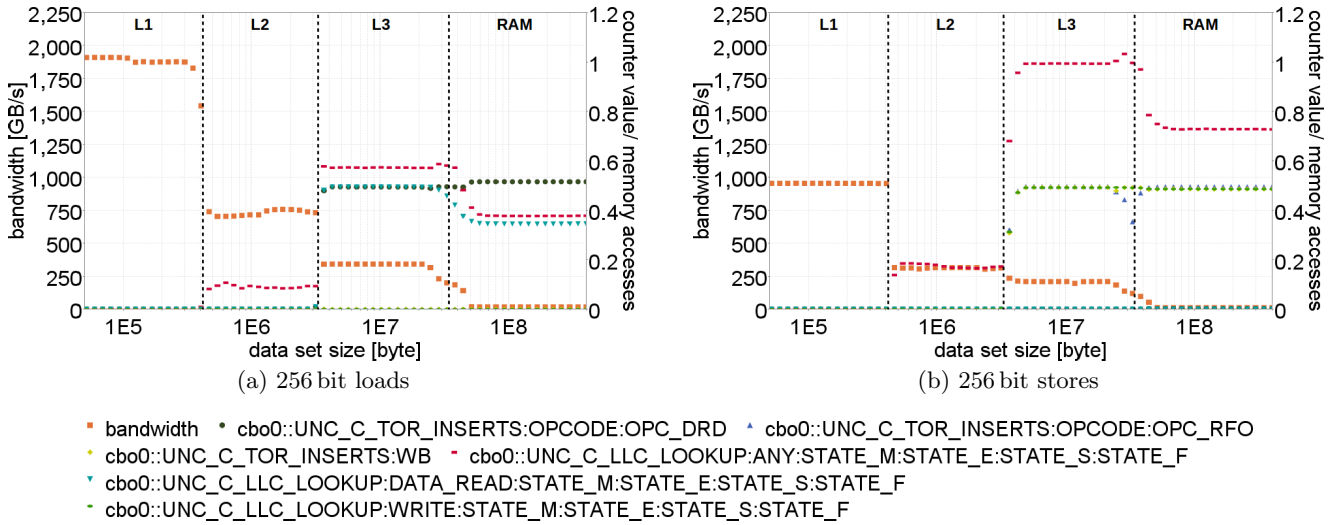
Figure 6 depicts the correlation between accesses to different levels in the memory hierarchy and C-Box events. The *UNC_C_LLC_LOOKUP* and *UNC_C_TOR_INSERTS* events can be used to measure the aggregated L3 bandwidth. As shown in Figure 6(b), RFOs are not covered by the *UNC_C_LLC_LOOKUP:DATA_READ* event. Therefore, *UNC_C_LLC_LOOKUP:ANY* has to be used in order to capture all loads, although this event also includes stores and the event rates are higher than expected in case of L2 accesses as well as reads from the L3 cache. Furthermore, most main memory accesses are also counted as LLC lookups. The *UNC_C_LLC_LOOKUP:WRITE* events correlate well with the number of cache lines written to the L3 cache. *UNC_C_TOR_INSERTS* events also count L3 accesses. In contrast to *UNC_C_LLC_LOOKUP:ANY* events, the reads (*:OPC_DRD* and *:OPC_RFO*) also include all main memory accesses, which simplifies compensating for the overlap with the DRAM counters. Unfortunately, different types of *UNC_C_TOR_INSERTS* events cannot be counted concurrently.

Reads from and writes to the main memory can be observed via the *UNC_H_REQUESTS* events in the home agent PMUs, which report one event per accessed cache line. Writes can also be counted via *UNC_H_IMC_WRITES:FULL*, which explicitly excludes partial writes. The PMUs in the inte-

Table 2: Suitable indicators for bandwidth usage per core

access type		achievable bandwidth	million transfers/s	most appropriate indicator for bandwidth utilization (including transfers from lower levels)	maximal events/s
L1D	read	159.8 GB/s	4,993 (32 byte)	perf::L1-DCACHE-LOADS	4,993 million
	write	79.9 GB/s	2,496 (32 byte)	perf::L1-DCACHE-STORES	2,496 million
L2	read	75.0 GB/s	1,171 (64 byte)	L2_TRANS:DEMAND_DATA_RD + L2_TRANS:RFO	1,546 million
	write	25.5 GB/s	398 (64 byte)	L2_TRANS:L1D_WB	398 million
L3	read	29.9 GB/s	467 (64 byte)	*OCR_L3-HIT* + *OCR_L3-MISS-loc* + *OCR_L3-MISS-rem*	467 million
	write	15.0 GB/s	277 (64 byte)	L2_TRANS:L2_WB	277 million
local DRAM	read	10.4 GB/s	162 (64 byte)	*OCR_L3-MISS-loc*	61 million
	write	7.7 GB/s	120 (64 byte)	n/a	n/a
remote DRAM	read	8.0 GB/s	125 (64 byte)	*OCR_L3-MISS-rem*	63 million
	write	5.5 GB/s	85 (64 byte)	n/a	n/a

OCR_L3-HIT = OFFCORE_RESPONSE_{0|1}:ANY_DATA:ANY_RFO:L3_HIT:SNP_ANY
OCR_L3-MISS-loc = OFFCORE_RESPONSE_{0|1}:ANY_DATA:ANY_RFO:L3_MISS_LOCAL:SNP_ANY
OCR_L3-MISS-rem = OFFCORE_RESPONSE_{0|1}:ANY_DATA:ANY_RFO:L3_MISS_REMOTE_HOP0:L3_MISS_REMOTE_HOP1 :L3_MISS_REMOTE_HOP2P:SNP_ANY



(a) 256 bit loads (b) 256 bit stores

Figure 6: Last level cache counters: All C-Boxes report very similar results, i.e., the data is evenly distributed across the L3 slices. For the purpose of clarity only the results from a single C-Box are shown. This also compensates the incorrect attribution of events caused by twelve CPUs to a single CPU as only one twelfth of the events is considered. Thus, the shown event ratios are correct.

grated memory controller (IMC) record additional DRAM specific information (precharges, refreshes, ECC errors, etc.). However, the home agent events are sufficient to measure the bandwidth utilization per socket. The traffic between the sockets can be measured separately for each of the two QPI links. The *UNC_ Q_RXL_FLITS_G1:DRS_DATA* and *UNC_Q_TXL_FLITS_ G1:DRS_DATA* events observe incoming and outgoing traffic, respectively. These events record eight flits (flow control unit—the link layer's unit of transfer [7]) per cache line. This matches the expectation of 64 bit data per flit (80 bits including CRC and control [7]). It is also possible to count the snoop requests and response caused by the coherence protocol via the *:DRS_NONDATA* subevent.

Table 3 summarizes the results of this section. If the maximal event rates are reached the respective component is 100% occupied. In contrast to the core counters, the uncore counters recognize all cache lines that are read from or written to memory. Therefore, the bandwidth in GB/s can be derived from the recorded event rates with a high accuracy.

4.3 Memory Related Stalls

In this section, we analyze if the stall counters that are available on Intel Xeon E5 v3 processors correctly represent the fraction of time that is spent waiting for the memory hierarchy. Multiple events correlate well with the execution time of the latency measurements. A selection of the available events is depicted in Figure 7. The event *CYCLE_ACTIVITY:STALLS_LDM_PENDING* perfectly matches the measured latency. The same is true for the *:CYCLES_NO_EXECUTE* subevent as well as the *UOPS_{ISSUED, EXECUTED,RETIRED}:STALL_CYCLES* events (not depicted). However, the latency benchmark does not include any operations other than the memory accesses. Therefore, it is unclear if the various events comprise other stall reasons as well. The *CYCLE_ACTIVITY:STALLS_L1D_PENDING* and *:STALLS_L2_PENDING* events report fewer stall cycles as they do not include the L1 and L2 lookup, respectively.

Figure 8 shows how the performance counter readings change if arithmetic operations are added between the loads. If independent operations are added (see Figure 8(a)), the

Table 3: Indicators for bandwidth usage per processor: The maximal event rates for the *UNC_C_ events refer to the sum of the events reported by the twelve C-Boxes (hsw_unc_cbo*). Likewise, the *UNC_H_** and *UNC_Q_** refer to the sum of both home agent (hsw_unc_ha*) and QPI Boxes (hsw_unc_qpi*), respectively.**

resource	access type	peak bandwidth	million transfers/s	most appropriate indicator for bandwidth utilization	maximal events/s
L3 cache	read	342 GB/s	5,343 (64 byte)	UNC_C_TOR_INSERTS:OPCODE:OPC_DRD / OPC_RFO*	5,279 million
	write	209 GB/s	3,265 (64 byte)	UNC_C_LLC_LOOKUP:WRITE	3,212 million
memory controller	read	63.1 GB/s	985 (64 byte)	UNC_H_REQUESTS:READS	985 million
	write	25.8 GB/s	403 (64 byte)	UNC_H_IMC_WRITES:FULL	403 million
QPI links	read	16.3 GB/s	254 (64 byte)	UNC_Q_RXL_FLITS_G1:DRS_DATA	2,032 million
	write	11.0 GB/s	171 (64 byte)	UNC_Q_TXL_FLITS_G1:DRS_DATA	1,368 million

*: cannot be measured together since the same filter register is used [8, Table 2-40], alternatively: UNC_C_LLC_LOOKUP:ANY − UNC_C_LLC_LOOKUP:WRITE, but this does not include all DRAM accesses that pass through the L3 cache

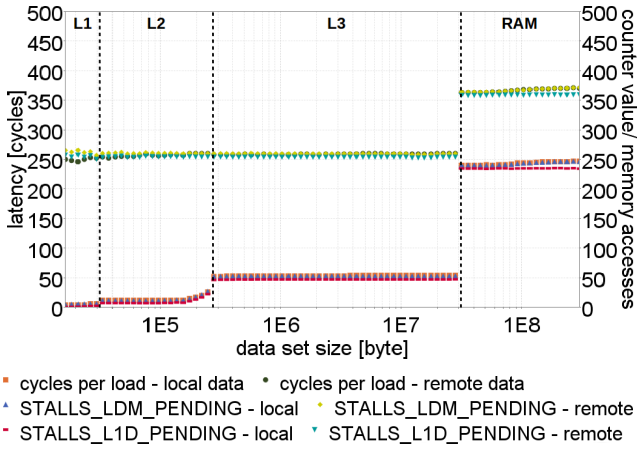
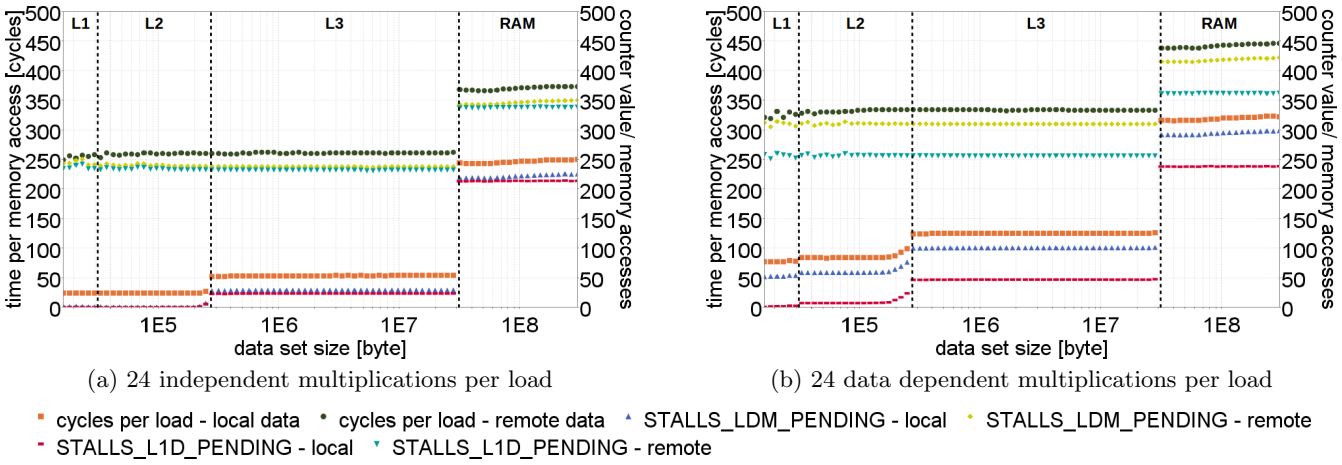


Figure 7: CYCLE_ACTIVITY events for latency benchmark: The pending loads and outstanding misses correlate well with the measured latency.

multiplications fill the gaps between the loads. Thus, the execution time remains constant if the latency is higher than the time required for the additional arithmetic operations. A useful indicator for the cycles lost due to memory accesses should only include the cycles that cannot be used for any other operations. This is the case for *CYCLE_ACTIVITY:CYCLES_NO_EXECUTE* and *:STALLS_LDM_PENDING*. They both report 29 instead of 53 cycles per memory access for data sets that fit into the L3 cache while the execution time is 53 cycles per load instruction with and without the multiplications. The 24 cycle reduction is identical to the number of operations added (throughput of 1 instruction per cycle [10, Table C-17]). If the multiplications are part of the dependency chain (see Figure 8(b))), the situation changes. In this case, the arithmetic operations depend on the result of their respective predecessor. Thus, the instruction latency (3 cycles [10, Table C-17]) also contributes to the execution time. These delays are clearly not caused by memory accesses and should therefore not be counted as memory related stalls. The results of an ideal memory stall counter would be identical to the latency measurements from Figure 7. However, such a counter does not exist. The *CYCLE_ACTIVITY:STALLS_LDM_PENDING / CYCLES_NO_EXECUTE* and *UOPS_* {*ISSUED,EXECUTED,RETIRED*}*:STALL_CYCLES* events report an increased number of stall cycles in this scenario, i.e., they include the delays caused by data dependencies between the multiplications. Therefore, they have to be excluded as indicators for memory related stalls. Only the *STALLS_L1D_PENDING* event looks promising. For accesses to the local L2 and L3 cache, the reported number of stalls is very close to the measured latency. The values reported for local and remote memory accesses are reasonably close as well. However, accesses to the L1 cache are not included.

There are two flavors of memory-boundedness—latency bound and bandwidth bound. In the former case the data waited upon is required to continue execution, i.e., the following instructions have a data dependency from the requested data. In the latter case independent instructions are available, but the data paths are used to their capacity, which restricts the processing speed of memory accesses. It is important to distinguish both cases as there are different optimization strategies for them, e.g., adding prefetch instructions in latency bound code or introducing cache blocking in bandwidth bound programs. The out-of-order execution cores are decoupled from the memory hierarchy via load and store buffers [10, Section 2.2.4.1]. Furthermore, several request queues exist that handle outstanding requests at various levels in the memory hierarchy. Bandwidth bound applications—which issue many independent requests—tend to fully utilize the available request buffers. Therefore, a high utilization ratio of the request queues is an indicator for bandwidth-boundedness.

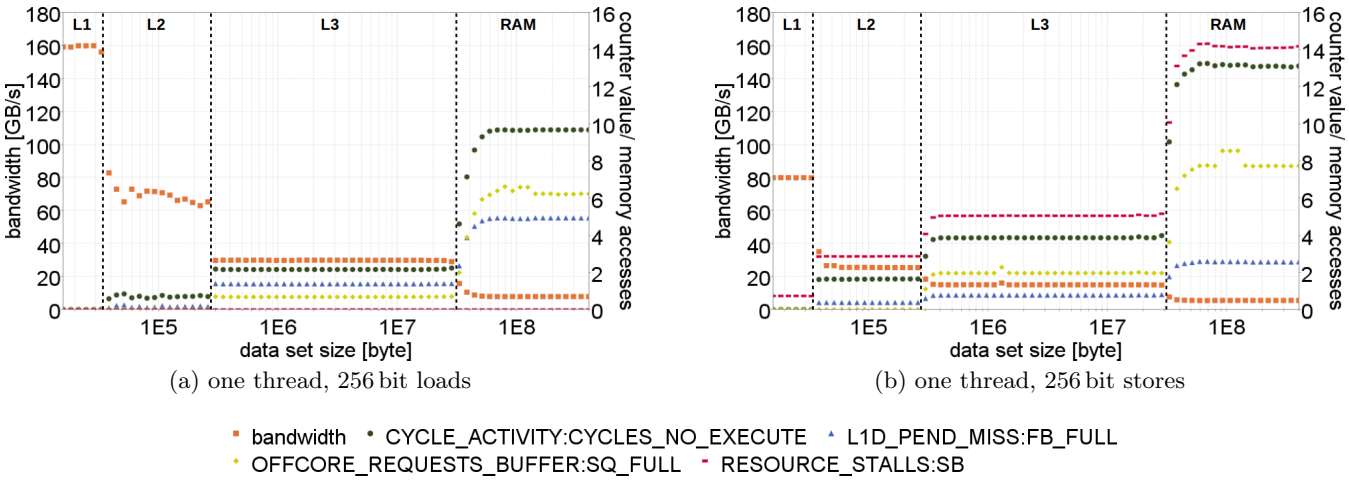
Figure 9 shows the event rates reported by several stall counters for bandwidth measurements. All stalls can be attributed to the memory accesses as the measurement routines do not include any other instructions. Therefore, the event *CYCLE_ACTIVITY:CYCLES_NO_EXECUTE* is used as a reference. The remaining events show no significant event rates for the latency measurement. Thus, they can be used to detect bandwidth-boundedness.



(a) 24 independent multiplications per load

(b) 24 data dependent multiplications per load

Figure 8: Correlation of stall cycles with adapted latency benchmark: Both events deduct the cycles required for performing the calculations from the reported number of stall cycles (left). However, the STALLS_LDM_PENDING event also includes delays caused by data dependencies (right).



(a) one thread, 256 bit loads

(b) one thread, 256 bit stores

Figure 9: Indicators for bandwidth-boundedness: In contrast to latency-bound scenarios, bandwidth-bound applications issue multiple concurrent requests. Typically, the cores can issue requests faster than they can be serviced. The execution is stalled eventually when the required buffers are filled.

Unfortunately, none of the events count the number of stall cycles accurately. *L1D_ PEND_MISS:FB_FULL* and *OFFCORE_REQUESTS_BUF-FER:SQ_FULL* do not cover bandwidth bound accesses in the L2 cache, but show high event rates in case of misses in the L2 cache. Unfortunately, even their sum is lower than the total number of stall cycles caused by the resulting L3 and memory accesses. However, it is the best available estimate for bandwidth bound loads. The event *RESOURCE_STALLS:SB* does not capture load accesses but correlates well with the stall cycles caused by stores. Unfortunately, it also overlaps with *CYCLE_ACTIVITY:STALLS_ L1D_PENDING* in case of mixed memory accesses (not depicted). The event *CYCLE_ACTIVITY:STALLS_L1D_ PENDING*, for example, includes cycles that are stalled for other reasons if there are loads outstanding at the time.

Based on these observations presented above, the stall cycles can be decomposed as shown in Figure 10. The *memory bound* fraction is determined as the maximum of the load related (*CYCLE_ACTIVITY:STALLS_L1D_PENDING*) and the store related (*RESOURCE_STALLS:SB*) stall cycles due to the potential overlap in mixed workloads. However, this can lead to an underestimation of the memory-boundedness

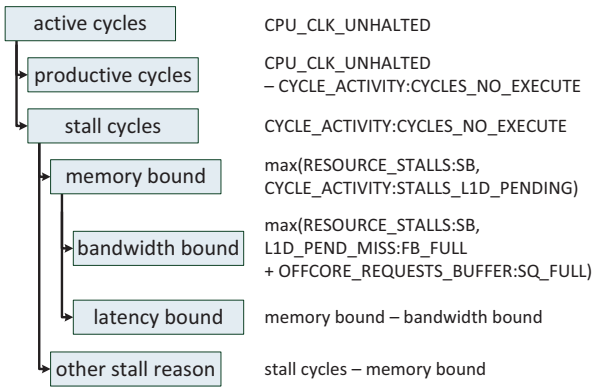


Figure 10: Decomposition of stall cycles

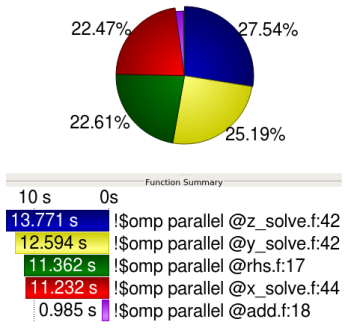
if a measurement interval includes discrete load bound and store bound phases. The *bandwidth bound* metric also uses the maximum as the events that capture loads also include RFOs caused by stores as depicted in Figure 9(b). The miss-prediction of branches is not considered here, i.e., "productive cycles" does not mean that the processed instructions are on the correct path. Ineffective speculation can be detected as described in [10, Appendix B.3.2]. Front-end stalls—including instruction cache misses—are also not covered by the approach presented here.

5. APPLICABILITY FOR ENERGY EFFICIENCY OPTIMIZATION

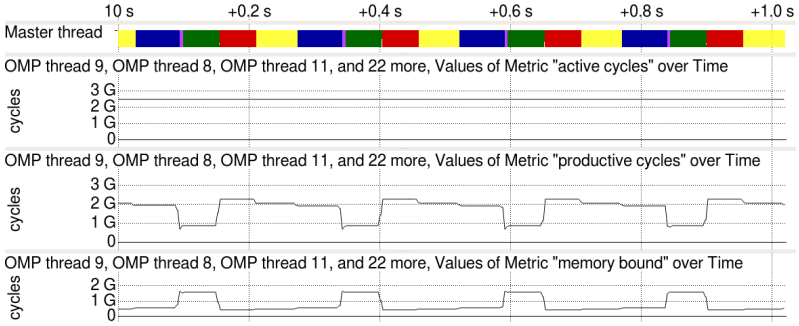
One possible application of the findings presented so far are energy efficiency optimizations. Researchers use the detection of memory boundedness to determine program phases where power-saving mechanisms can be used. Based on the assumption that the memory performance is not linked to the processor frequency, they use dynamic voltage and frequency scaling (DVFS) to slow down processor cores [23]. However, this approach is not viable for all processor architectures [22]. We used our model to optimize the OpenMP parallel NAS benchmark bt [3] in version 3.3.1.

We gather the metrics that are described in Figure 10 by using Score-P. A profile of the executed parallel regions is depicted in Figure 11(a). An excerpt of the trace is visualized in Figure 11(b). During the execution of the parallel regions defined in `add.f` and `rhs.f` the processor cores spend less than 50 % of their time in productive cycles. The remaining cycles are mostly stalls due to memory bandwidth limitation. Thus, these regions can be optimized using dynamic voltage and frequency scaling.

We used the Score-P substrate plugin interface to call the optimization library `libadapt` [21]. This library is able to optimize the hardware software environment based on predefined rules that are passed via a configuration file. In this file, we stated that the frequency should be reduced to 1300 MHz when the parallel region in `add.h` is started and that the frequency should be reset when the parallel region in `rhs.f` is left.



(a) Runtime share and absolute runtimes of parallel regions



(b) Four compute iterations and respective metrics "active cycles", "productive cycles", and "bandwidth bound cycles".

Figure 11: VAMPIR visualization of the OpenMP parallel NAS benchmark *bt* at reference frequency. The parallel regions in *add.f* and *rhs.f* have a high share of non-productive bandwidth bound cycles.

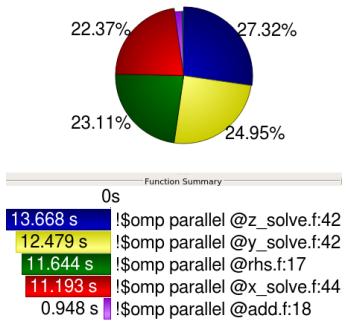
The resulting runtime profile and the recorded metrics are depicted in Figure 12. To verify the optimization results, we collect power consumption information from the test system using a ZES-ZIMMER LMG 450 power meter. This highly accurate device is attached to the power plug of the server and returns power consumption samples every 50 ms [6]. The average power consumption while executing the unoptimized version of the benchmark is 284 Watts. After reducing the frequency in the selected parallel regions, the overall average power consumption is reduced to 271 Watts, which is a reduction of 4.6 % based on the reference measurement. The runtime is increased by 0.2 %. However, the optimization only targets 25 % of the runtime and does not change the frequency of the uncore, which represents a significant share of the overall power consumption of the processor. Thus, the potential for energy efficiency optimizations with DVFS is limited for the given application.

6. CONCLUSIONS

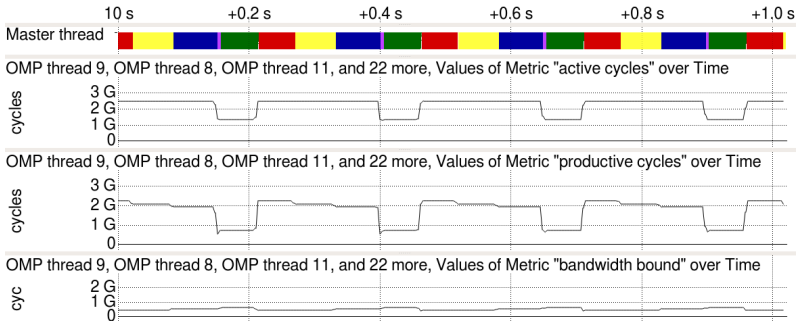
Knowing the maximal achievable performance of individual components in the memory hierarchy is an essential prerequisite to detect memory related performance issues. In order to determine the impact of the memory hierarchy on the achieved application performance, the waiting times caused by memory accesses have to be measured at runtime. This creates several challenges: While the performance degradation due to resource limitations is often reflected by certain hardware events, raw performance counter values are usually difficult to understand. Furthermore, modern processors support numerous events that monitor memory accesses. Selecting the relevant ones is not a trivial task. The fact that many events do not work as one might expect further complicates the performance analysis task.

To tackle these challenges, we present a portable methodology that derives meaningful metrics for the resource utilization from hardware performance counters. The practically achievable bandwidths (e.g., via QPI) can be lower than the theoretical peak performance. Thus, even if suitable events for measuring the achieved bandwidth do exist, the resource utilization can be severely underestimated if the theoretical peak performance is used as a reference. We therefore base our utilization metrics on the achievable bandwidth, which we measure with highly optimized micro-benchmarks. Furthermore, we present a novel approach to verify if the various stall events that can be counted by the core PMUs are able to distinguish memory related stall cycles from other stall reasons. We also describe how events that differentiate bandwidth bound and latency bound stall cycles can be identified.

We demonstrate the applicability of the approach on a Haswell-EP based system. We show that the available core counters can be used as indicator for a high utilization, but are not very useful for accurately measuring the bandwidth usage of shared resources. In contrast, the uncore counters provide events that measure the bandwidth usage quite ac-



(a) Runtime share and absolute runtimes of parallel regions



(b) Four compute iterations and respective metrics "active cycles", "productive cycles", and "bandwidth bound cycles".

Figure 12: DVFS optimization of *add.f* and *rhs.f* where a frequency of 1300 MHz is applied. The overall cycles and the number of bandwidth bound cycles are reduced by approx. 50 % during these functions.

curately. We show that the CYCLE_ACTIVITY:STALLS _LDM_PENDING event, which is intended to detect memory related stalls, also includes stalls that are caused by data dependencies between register-only operations. It is therefore not applicable to reliably detect memory bound program phases. Based on our verified set of performance counters, the adequate combination of multiple counter values and a visualization of the performance counter data, we demonstrate that complex memory performance problems can be found by the non-expert user.

As our experiments have shown, some performance counter events do not deliver the expected results. Therefore, it cannot be recommended to rely on performance counter readings without validating that they correlate with the characteristic that should be measured. *X86-membench* is ideally suited to perform such validations. The presented workflow is portable to other architectures. However, our benchmarks only support the x86 instruction set at the moment. The results obtained on one architecture cannot easily be transferred to other architectures as the set of available events as well as their definition and functionality can be different. Analyzing a new processor architecture currently is a time consuming task with many manual steps. The automatic detection of (potentially) useful counters and the generation of the corresponding metrics is subject to future work.

Acknowledgments

This work has been funded in a part by the European Union's Horizon 2020 program in the READEX project under grant agreement number 671657 and by the Bundesministerium für Bildung und Forschung via the research project Score-E (BMBF 01IH13001).

References

[1] L. Adhianto et al. Hpctoolkit: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 2010. DOI: 10.1002/cpe.1553.

[2] AMD. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors*, Jan 2013. Publication # 42301, Revision: 3.14.

[3] D. H. Bailey et. al. The nas parallel benchmarks—summary and preliminary results. In *ACM/IEEE Conference on Supercomputing*, 1991. DOI: 10.1145/125826.125925.

[4] L. A. Barroso et al. Memory system characterization of commercial workloads. *SIGARCH Comput. Archit. News*, 1998. DOI: 10.1145/279361.279363.

[5] S. Eranian. What can performance counters do for memory subsystem analysis? In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, 2008. DOI: 10.1145/1353522.1353531.

[6] D. Hackenberg et al. Power measurement techniques on standard compute nodes: A quantitative comparison. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013. DOI: 10.1109/ispass.2013.6557170.

[7] Intel. *An Introduction to the Intel® QuickPath Interconnect*, 1 2009.

[8] Intel. *Intel® Xeon® Processor E5 and E7 v3 Family Uncore Performance Monitoring Reference Manual*, 6 2015. Document Number: 331051-002.

[9] Intel. *Intel® Xeon® Processor E5 v3 Product Families - Specification Update*, 8 2015. Reference Number: 330785-009US, Revision 009.

[10] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Jan 2016. Order Number: 248966-032.

[11] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes 1, 2A, 2B, 2C, 3A, 3B and 3C*, Apr 2016. Order Number: 325462-058US.

[12] A. Knüpfer et al. Score-P: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *International Workshop on Parallel Tools for High Performance Computing*. 2012. DOI: 10.1007/978-3-642-31476-6_7.

[13] D. Levinthal. Performance analysis guide for Intel Core i7 processor and Intel Xeon 5500 processors. Technical report, Intel, 2009.

[14] J. D. Little. A proof for the queueing formula: $L = \lambda \cdot W$. *Operations Research*, 9(3), 1961.

[15] Z. Majo and T. R. Gross. Memory system performance in a numa multicore multiprocessor. In *International Conference on Systems and Storage (SYSTOR)*, 2011. DOI: 10.1145/1987816.1987832.

[16] D. Molka. *Performance Analysis of Complex Shared Memory Systems*. PhD thesis, Technische Universität Dresden, 2017.

[17] D. Molka et al. Main memory and cache performance of intel sandy bridge and amd bulldozer. In *Workshop on Memory Systems Performance and Correctness (MSPC)*, 2014. DOI: 10.1145/2618128.2618129.

[18] D. Molka et al. Cache coherence protocol and memory performance of the intel haswell-ep architecture. In *International Conference on Parallel Processing (ICPP)*, 2015. DOI: 10.1109/ICPP.2015.83.

[19] L. Oliker et. al. A performance evaluation of the cray x1 for scientific applications. In *High Performance Computing for Computational Science (VECPAR 2004)*. 2005. DOI: 10.1007/11403937_5.

[20] V. Palomares. *Combining static and dynamic approaches to model loop performance in HPC*. PhD thesis, Université de Versailles, 2015. https://tel.archives-ouvertes.fr/tel-01293040.

[21] R. Schöne and D. Molka. Integrating performance analysis and energy efficiency optimizations in a unified environment. *Computer Science - Research and Development*, 2014. DOI: 10.1007/s00450-013-0243-7.

[22] R. Schöne et al. Memory performance at reduced cpu clock speeds: an analysis of current x86_64 processors. In *USENIX conference on Power-Aware Computing and Systems (HotPower)*, 2012.

[23] V. Spiliopoulos et al. Green governors: A framework for continuously adaptive dvfs. In *International Green Computing Conference and Workshops (IGCC)*, 2011. DOI: 10.1109/IGCC.2011.6008552.

[24] D. Terpstra et al. Collecting performance data with PAPI-C. In *International Workshop on Parallel Tools for High Performance Computing*, 2009. DOI: 10.1007/978-3-642-11261-4_11.

[25] J. Treibig et al. Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering. In *Euro-Par 2012: Parallel Processing Workshops*. 2013. DOI: 10.1007/978-3-642-36949-0_50.

[26] A. Yasin. A top-down method for performance analysis and counters architecture. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014. DOI: 10.1109/ISPASS.2014.6844459.

[27] W. Yoo et al. Adp: Automated diagnosis of performance pathologies using hardware events. In *Proceedings of the 12th ACM SIGMETRICS/ PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, 2012. DOI: 10.1145/2254756.2254791.