Karlsruher Institut für Technologie

Fine-Grained Estimation of Memory Bandwidth Utilization

Masterarbeit
von

Florian Larysch

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuender Mitarbeiter:	Dipl.-Inform. Marius Hillenbrand

Bearbeitungszeit: 14. September 2015 – 13. März 2016

The usefulness of a model is not what it can explain, but what it can’t. [. . .] Your strength as a rationalist is your ability to be more confused by fiction than by reality. If you are equally good at explaining any outcome, you have zero knowledge.

— *Eliezer Yudkowsky*

DECLARATION

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 13. März 2016

Florian Larysch

ABSTRACT

Main memory bandwidth is a shared resource among applications running on a system. Thus, the behavior of individual applications can have performance implications for other applications running on the same system. This makes main memory bandwidth utilization a interesting characteristic of applications. Furthermore, the patterns of utilization matter: Does an application use a constant amount of bandwidth or does it cause bursts of high utilization separated by phases of low utilization, for example?

As memory accesses are usually invisible to the operating system, measuring the actual memory access behavior of applications in a live system is difficult. In this thesis, we develop a system for capturing main memory bandwidth consumption traces with a high temporal resolution on commodity hardware. To build such a system, we analyze existing hardware mechanisms for directly or indirectly monitoring memory access, such as Intel Memory Bandwidth Monitoring, Performance Counters and Precise Event-Based Sampling (PEBS). We pay special attention to achieving a high resolution while maintaining a low overhead and implement a prototype system based on PEBS.

We evaluate our implementation using both synthetic and real-world benchmarks and find that our system does indeed reduce the incurred worst-case overhead compared to traditional approaches by about 40%. We also find that the reduced overhead of our system increases its accuracy by lowering interference with the measured application.

CONTENTS

1 INTRODUCTION 1
2 BACKGROUND AND RELATED WORK 3
  2.1 The Architecture of the Skylake CPU Family 3
  2.2 Related Work 6
3 ANALYSIS 9
  3.1 Criteria 9
  3.2 Mechanism Selection 10
  3.3 Experimental Verification 13
  3.4 Further Experiments 17
  3.5 Conclusion 21
4 DESIGN 23
  4.1 Design Goals 23
  4.2 Design Overview 24
  4.3 Design Rationale 25
  4.4 PEBS Configuration 25
  4.5 PEBS Handler 26
  4.6 Postprocessing 26
  4.7 Ring Buffer 28
  4.8 Trace Consumer 28
  4.9 Conversion Between Overflow-Based and Sampling-Based Representations 29
5 IMPLEMENTATION 31
  5.1 Choice of Operating System 31
  5.2 Operating System Integration 31
  5.3 PEBS Configuration 32
  5.4 Performance Counter Configuration 33
  5.5 PEBS Handler 33
  5.6 User Space Interface 34
  5.7 Trace Consumer 34
6 EVALUATION 37
  6.1 Experimental Setup 37
  6.2 Baseline Software-Based Approach 37
  6.3 Accuracy 38
  6.4 Overhead 44
  6.5 Accuracy Implications of Overhead 44
  6.6 Achievable Throughput 47
7 CONCLUSION 49
  7.1 Future Work 50

I APPENDIX 51

BIBLIOGRAPHY 53

1 INTRODUCTION

Main memory bandwidth is a shared resource like CPU time or IO bandwidth, but while application consumption of the latter can be readily monitored by the operating system itself (because for both scheduling decisions and IO accesses, control usually passes from the application to the operating system kernel), memory bandwidth consumption is much harder to account for, as most memory accesses (other than page faults and similar exceptions) are handled in hardware, without involvement of the operating system.

Nevertheless, the memory bandwidth consumption of an application is an interesting characteristic, as it directly influences the performance of other applications: Not only does an application use up a part of the finite resource that is the available memory bandwidth, but the access patterns it produces over time also affect the queuing and scheduling of memory accesses by the memory controller, thereby possibly influencing other applications in a more complex fashion.

This thesis aims to evaluate the feasibility of using hardware performance monitoring mechanisms to precisely measure the amount of main memory bandwidth consumed by a particular application, where "precision" means precisely measuring the amount of data transferred, but more importantly also means precisely determining the temporal characteristics of such accesses. We hope that this will enable further study of the interactions between applications and the memory subsystem and interference between applications caused by their memory access patterns.

To that end, we develop a bandwidth monitoring system based on the introduction of time stamps into the *Precise Event-Based Sampling* (PEBS) mechanism of the Intel Skylake generation of CPUs and evaluate whether it yields an improvement over previous approaches.

The remainder of this work is structured as follows: Chapter 2 presents an overview of the underlying technologies and related work. Chapter 3 reevaluates assumptions made in previous work about how to measure memory bandwidth consumption using performance counters. In Chapter 4, we propose an improved mechanism for measuring memory bandwidth consumption and describe its implementation in Chapter 5. We evaluate our new mechanism in Chapter 6. Chapter 7 concludes this work and outlines future work.

2 BACKGROUND AND RELATED WORK

In this chapter, we introduce the concepts and technologies which are relevant for understanding the work done in this thesis. We also outline the scientific context of this thesis by reviewing related work.

As the main contribution of this thesis is based on advancements in the current generation of Intel CPUs (called *Skylake*), we will focus on presenting its architecture in this section. However, as detailed documentation is scarce, we will occasionally make assumptions based on published information about previous generations.

2.1 THE ARCHITECTURE OF THE SKYLAKE CPU FAMILY

The Skylake microarchitecture is a multi-core/multiprocessor architecture with out-of-order-execution and symmetric multithreading (SMT, also known as *Hyperthreading*) support [10]. Since the departure of the *Front Side Bus* and the *Northbridge* and their replacement by the *QuickPath Interconnect* (QPI) and integrated memory controllers (IMCs) in each CPU in the *Nehalem* generation of processors, Intel CPUs have effectively become a NUMA (non-uniform memory access) architecture, where the access to main memory connected to a processor's integrated memory controller occurs without the intervention of other CPUs, while requests to remote memory must first traverse a QPI link to another CPU [2].

Each core contains an L2 cache which is shared between instructions and data, and dedicated L1 caches for instructions and data each. Additionally, each core contributes a *slice* of memory to the L3/last-level cache (LLC), which is shared both between all cores on a die and the graphics system [15]. All transfers between each of the cores, the shared LLC and the *system agent* (representing the external interface of the CPU and containing the IMC, PCIe and QPI controllers) are mediated via a ring bus. All the subsystems of the processor which lie outside the individual cores are collectively called the *uncore*.

As an out-of-order architecture, the cores do not necessarily execute instructions in the same order they occur in the program and might even speculatively execute branches, only to discard these results should the branch not be taken. Thus, it is important to distinguish between instructions starting execution (being *dispatched*) and actually finishing and having observable effects (*retiring*). This also implies that later instructions might retire before earlier instructions if there is no dependency between them and no explicit reordering

barrier is inserted. As this process happens transparently, it is difficult to reason about the temporal sequence of events within a CPU.

2.1.1 *Performance Counters*

Modern Intel CPUs contain *Performance Monitoring Units* (PMUs) for measuring the performance and behavior of the different subsystems of the processor. PMUs contain sets of *Performance Counters* (PMCs), which can either be configured to count various events, such as cache hits or branch mispredictions or are hardwired to count a specific event. These counters are exposed to the system software as *Model-Specific Registers* (MSRs), which can be accessed using the `rdmsr` and `wrmsr` instructions. Additionally, some of these counters can be configured to cause an interrupt once they overflow.

In this thesis, we mainly use the on-core PMU, which deals with performance-monitoring events caused by an individual core. These counters can be also configured to only count events in specific execution modes, such as ring 0 or ring 3. Crucially, the overflow interrupts generated by the performance counters contained in this PMU are always delivered locally, which means it is not possible to route those interrupts to another core.

On the Skylake microarchitecture, each core contains 8 (or 4, if SMT is enabled) 48-bit performance counters, which can be freely configured to count almost any event (PEBS-capable events only work in the first four PMCs, even if PEBS is not being used, see below for an explanation of PEBS).

The available events are divided into *architectural* and non-architectural performance events. Architectural events are abstract events which ideally behave the same way across microarchitectures, whereas non-architectural events are specific to each individual microarchitecture generation.

Performance counters are usually used to find and analyze bottlenecks in user space applications. This is done as follows:

1. The operating system configures a performance counter to increment whenever a specific event occurs in ring 3 and preloads the counter with a value, such that it will overflow after a number of events (called the *threshold*) have occurred.

2. Control is transferred to the application.

3. If the counter overflows, an interrupt is generated. The interrupt handler records the overflow and also captures some of the state of the executing application, such as its current instruction pointer.

4. The interrupt handler once again preloads the performance counter to overflow after a number of events and returns control to the application.

5. If the application has used up its time slice or has to stop executing for another reason, the operating system stores the current value of the performance counter to be able to restore it once the application resumes execution.

Note that as there is a small but variable delay between the overflow occurring and the interrupt handler commencing execution, there will usually be a certain overshoot of events within the counter which has overflowed, which needs to be taken into account when calculating the next value to be loaded into the performance counter. Thus, the interrupt handler also reads the current value of the performance counter and uses it as an offset for calculating the next value to be loaded into the counter.

The above algorithm is also the approach taken by the *perf* subsystem in the Linux kernel, which is the standard high-level interface for performance measurements and profiling in Linux.

While this approach works, it suffers from two problems: Firstly, as the measurement framework has to run on the same core as the application that is being analyzed, there is a positive relationship between achieving higher accuracy (by using a lower threshold) and increasing the slowdown of the measured application, because a larger fraction of time is being spent handling the increased number of counter overflows. Also, the measurement process itself increasingly influences the core's state, for example by putting additional pressure on the cache, thus distorting the measurements.

Secondly, when we are interested in the precise location in an application's code which caused an event, such as a branch misprediction, to occur, the aforementioned delay before the overflow interrupt occurs causes the instruction pointer to advance an unpredictable amount (called *skid*), before the sampling within the interrupt handler can take place.

The second problem was solved by the introduction of *Precise Event-Based Sampling* (PEBS) in the Core microarchitecture. While not its stated goal, it also alleviates some aspects of the first problem by reducing the overhead incurred by each overflow.

2.1.2 *PEBS*

PEBS is a hardware and microcode mechanism which implements the previously described algorithm for sampling the machine state at the time of a PMC overflow without the involvement of the operating system. The operating system only designates a buffer in memory and configures reload values for the counters of interest. Once

PEBS is enabled, a microcode assist takes care of snapshotting the various registers and resetting the performance counter to the configured value. For each overflow event, a PEBS record is stored to the configured buffer. Only when this buffer exceeds a configurable fill level, an interrupt is signaled and the operating system will process the collected records.

The main advantage of this approach is that, because the entire process is under microcode control, asynchronicity inside the processor can be more easily accounted for, allowing PEBS to pinpoint the instruction which caused the event to occur with a constant skid of one instruction.

As PEBS focuses on events that are tightly causally related to the instruction stream, only a few performance monitoring events which mostly deal with the retirement of instructions are compatible with it.

The number of elements in a PEBS record has grown over time [11]. Crucially for this work, as of the Skylake microarchitecture, these records also contain the value of the *Time-Stamp Counter* (TSC), which is a clock with a high resolution and constant rate, at the time the PEBS assist was invoked.

2.1.3 *Prefetchers*

Prefetchers are a mechanism by which the processor recognizes memory access patterns and tries to reduce memory access latency by automatically fetching data from the predicted next access locations into the cache. As we will want to deduce memory accesses causally related to application behavior from cache misses, prefetchers can add a considerable distortion to our measurements if the workload lends itself to prefetching. Thus, it will be useful to disable prefetchers, which is possible on Intel CPUs using the MSR `0x1A4` [16].

2.2 RELATED WORK

One way to analyze the memory access behavior of applications is simulation. Popular simulators such as QEMU[3] and gem5[7] suffer from slowdown and inaccuracy. While QEMU is designed to minimize the slowdown it causes when simulating a machine, it sacrifices accuracy by only trying to be indistinguishable from a real machine from the point of view of the application running inside of it, not replicating the precise behavior of the memory hierarchy, for example. This is reflected in its name, as it can be more readily described as an emulator, rather than as a simulator. Gem5, on the other hand, tries to be an accurate simulator with respect to the selected CPU model and memory hierarchy. However, this makes it very slow, typically causing a slowdown of several orders of magnitude. Hybrid approaches

such as MARSS[5] exist, which combine precise simulation with fast emulation where such precision is not needed (for example, when forwarding to an interesting point within a benchmark run).

We are, however, interested in analyzing the behavior of applications running on actual machines and replicating the exact environment inside a simulator is difficult. Furthermore, no models are available for modern CPU families, and probably never will be, as many aspects of the behavior of these machines are kept secret.

In live systems, main memory bandwidth measurement has been explicitly done as a part of other works, but they focused on building resource allocation policies on top of mechanisms for memory bandwidth measurement and throttling: Both Bellosa[4] and Yun et al.[17] used performance monitoring to count cache misses, from which they inferred the amount of read traffic towards main memory (thus ignoring memory writes and DMA) and used this information to throttle applications which violate pre-set limits to achieve better performance guarantees for real-time systems.

While our approach also monitors cache misses and is thus also limited to read traffic, we focus not on resource allocation policies, but on improving the process of measurement itself. We do this by first analyzing the behavior of performance counters more rigorously than done in previous work and then design a mechanism which enables precise memory bandwidth measurements on a modern CPU architecture without causing excessive interference.

3 ANALYSIS

In this chapter, we analyze which hardware mechanisms enable us to measure consumed main memory bandwidth and could be used as a basis for a high-resolution tracing mechanism.

First, in Section 3.1, we establish some criteria which encompass the goals for the system we are going to develop and which will enable us to judge the usefulness of the available base mechanisms. We then discuss the various mechanisms with respect to these criteria and select a candidate in Section 3.2. Finally, in Section 3.3, we describe the experiments which we used to verify that this mechanism does indeed help us to measure memory bandwidth and conclude our analysis in Section 3.5.

3.1 CRITERIA

There are various hardware mechanisms available which could be used for measuring memory bandwidth. However, these mechanisms have very different characteristics, which make them useful for some goals, but not for others. To guide our selection of a mechanism, we establish a set of criteria which capture the characteristics which are important for our goal:

- Precision: The values reported by the mechanism should closely mirror the actual memory traffic caused by an application.
- Temporal accuracy: There should be a way to precisely deduce the temporal pattern of memory accesses.
- Specificity: The mechanism should be usable in a way that will allow us to analyze the memory access behavior of a single application, that is, we should be able to filter for individual applications / cores, rather than observing the aggregated memory traffic of the whole system.
- Availability: Hardware implementing the mechanism should be readily available.
- Low interference and overhead: The measurement mechanism should not interfere with the measured application too strongly, possibly distorting the measurements and slowing down the application.

3.2 MECHANISM SELECTION

We now discuss the available mechanisms with respect to these criteria.

3.2.1 *Intel MBM*

Intel *Memory Bandwidth Monitoring* (MBM) is a mechanism present in a few Intel server processors, which, in cooperation with the operating system scheduler, allows individual threads to be tagged with Resource Monitoring IDs (RMIDs), which can then be used to measure the main memory bandwidth consumed by those threads.

MBM is ideal insofar that it is a mechanism which is dedicated to measuring memory bandwidth. Because of that, we would expect a high precision and specificity from it. However, MBM only allows the operating system to query for the currently consumed memory bandwidth by reading from a MSR, rather than supplying a hardware mechanism which automatically generates traces, which reduces the achievable temporal accuracy. Also, processors supporting MBM are not readily available: We were unable to acquire any such processor and thus were unable to evaluate its feasibility for achieving our goal.

3.2.2 *Mechanisms Based on Performance Counters*

Previous work (see Section 2.2) has used cache-related performance counters as an intermediary for determining the memory bandwidth consumed by an application.

Modern x86 processors include facilities that allow for timestamped tracing of performance counter overflows, allowing us to achieve higher temporal accuracy than with manually recording timestamps once an overflow occurs: Lightweight Profiling (LWP) by AMD and Precise Event-Based Sampling (PEBS) by Intel.

3.2.2.1 *PEBS*

PEBS was already described in Section 2.1.2. As of the Skylake microarchitecture, each PEBS record includes the value of the timestamp counter at the time of the overflow that triggered the PEBS assist, which should allow for a high temporal accuracy.

3.2.2.2 *LWP*

AMD's Lightweight Profiling (LWP [13]) was introduced with the Bulldozer microarchitecture and is a similar mechanism to PEBS. As the records written by LWP are much more compact than those written by PEBS (32 bytes vs. 200 bytes), the memory traffic (and thus

the interference) caused by LWP should be much lower than the one caused by PEBS.

However, while we were able to acquire hardware which supports LWP in general, the implementation present within that processor did not support tracing cache-related events. Furthermore, AMD discontinued support for LWP with the upcoming Zen microarchitecture. Thus, this analysis and the remaining work done as part of this thesis focus on exploiting PEBS on Skylake processors to achieve memory bandwidth tracing.

In the rest of this chapter, we will analyze whether using performance counters to measure consumed main memory bandwidth is feasible on the Skylake processor family, as no such analysis is currently available: MemGuard [17], the most recent work in this area, incorporates memory bandwidth measurements through performance counters, but has only been tested on Core2Quad and Sandy Bridge processors, which, while recent, are based on different microarchitectures and thus might exhibit different behavior than Skylake. Similar concerns apply to other publications which cover older processor architectures. Additionally, all of these publications take the correspondence between their chosen performance monitoring event and memory accesses as a given and do not provide a detailed analysis of that assumption.

3.2.3 *Performance Monitoring Event Selection*

A modern processor contains many different performance counters [11], but only a small fraction are related to memory accesses. Based on the above criteria, we can considerably narrow down the list of events we need to evaluate.

Off-core/Uncore events within the memory controller or the CBos (Cache-Boxes), which manage the LLC accesses, are very precise, but not very specific. For example, the memory controller can count actual main memory accesses, but the information which core caused the access to happen is not available as a filter. Furthermore, no off-core event is compatible with PEBS and could thus only serve as a comparison during the evaluation of other events, but will not be usable in the PEBS-based mechanism. Additionally, at the time work on this thesis was begun, only the desktop processors of the Skylake family were available, which have severely reduced uncore monitoring compared to the server processors. Even now, documentation about uncore performance monitoring on Skylake remains scarce. Thus, we mostly exclude uncore performance monitoring from our further considerations.

On-core events are very specific, as they only occur for transactions that are relevant to each core. However, there is still some room for error. For example, an event counting responses from the uncore to

a core might trigger after a context switch to another application has happened in cases of high response latency. Thus, we should favor events which occur as early as possible. This, however, affects precision. If we simply counted all memory access instructions as soon as they are dispatched, we would not know whether they hit a cache or actually caused a main memory transaction to occur.

Considering all on-core events for their ability to track main-memory accesses, we are left with the following events (from here on we refer to events using their official names as given in the Intel documentation):

- `OFFCORE_REQUESTS.L3_MISS_DEMAND_DATA_RD`
- The `OFF_CORE_RESPONSE` events
- The architectural `LONGEST_LAT_CACHE.MISS` event
- `MEM_LOAD_RETIRED.L3_MISS`

While some other events could be used in combination to calculate memory bandwidth, only overflows of individual counters can trigger PEBS and each counter can only track one event, so we focus on individual events instead.

As our initial experiments showed that `LONGEST_LAT_CACHE.MISS` and `OFFCORE_REQUESTS.L3_MISS_DEMAND_DATA_RD` behaved similarly, we excluded the latter from further analysis, as the former is more commonly used for monitoring last-level cache misses and would thus benefit more from being included in our analysis. The group of events represented by `OFF_CORE_RESPONSE` is interesting, as it affords fine-grained control about which transactions are counted, for example allowing us to selectively monitor prefetcher traffic, if desired. However, this event miscounted and also exhibited spurious activity, which caused us to drop it from further consideration.

As `MEM_LOAD_RETIRED.L3_MISS` is the only event in the list which is supported by PEBS, it will be the one we have to select in our new approach. This event counts the retired instructions where at least one micro-operation (*uop*) caused an L3 miss. This means that this event will not count speculative executions if they do not eventually retire, causing us to undercount when speculative loads get aborted. Additionally, some instructions contain more than one memory uop [9] which might cause more than one L3 miss, only one of which would be counted.

We now evaluate this event for its precision. As MemGuard uses `LONGEST_LAT_CACHE.MISS` on most microarchitectures, we include it in the evaluation for comparison. Note that there has been an erratum concerning Skylake ([1], SKD019), recommending against the use of the latter event, claiming that it might miscount cache misses, but this erratum has since been withdrawn and is "under consideration" by Intel, according to personal communication.

3.3 EXPERIMENTAL VERIFICATION

As the official documentation often does not explain what an event actually means and, judging from previous errata, the performance monitoring implementation within processors tends to contain many bugs, we choose an experimental approach for verifying that prospective events do indeed correspond to memory traffic.

To be able to analyze the behavior of various performance monitoring events during our initial experiments, we developed a user space tool which samples the values of performance counter registers at regular intervals. This was easier to implement than the usual overflow-based approach (see Section 2.1.1) and also has the advantage of only creating a constant disturbance in the program under observation, rather than one that varies with the frequency of events, as in the overflow-based case. We reuse this tool for the following experiments.

As the `rdmsr` instruction, which is used to read the performance counters, is privileged, we need kernel support for sampling the performance counters. While this could be done using the `msr` module for the Linux kernel, this module only affords reading a single MSR at a time. As we also wanted to analyze correlations between the various events, all PMCs should be sampled at the same point in time. To this end, we used a customized version of the `msr` module.

To clearly see the behavior of the different events, we use a synthetic benchmark, adapted from *pmbw* [6], which allocates a chunk of memory and then linearly reads from successively larger areas from within this chunk in a loop such that the total number of bytes read for each size remains the same. This serves to exercise different parts of the cache hierarchy, so we can also observe the behavior of events for memory accesses which solely miss the lower-order caches. If an event were to also trigger for those, it would not be useful for main memory bandwidth monitoring even if it worked correctly in the later parts of the benchmark.

As we do not know whether any event accurately represents memory traffic, we need to rely on another way to measure memory bandwidth for comparison. While we could measure the achieved read bandwidth from within our benchmark, this value represents the bandwidth visible to the application, which means it is distorted by the presence of caches and only begins to approximate main memory bandwidth once the size of the work area increases beyond the capacity of the last-level cache.

However, the memory controller also supports rudimentary monitoring [14] via the memory-mapped `DRAM_DATA_READS` and `DRAM_DATA_WRITES` registers, which count the number of 64-byte read and write accesses, respectively. While these counters are not core-specific and thus count all the memory traffic of the system, we can still derive

a correlation, as the system will be otherwise idle during a benchmark run.

As each cache miss does itself correspond to a 64-byte cache line fetch and memory transactions are also counted in multiples of 64 bytes, we expect an occurrence of the cache miss event to also cause a single increment of the read transaction counter in the memory controller. Thus, the values reported by the memory controller and those from the performance monitoring unit should merely differ by a constant offset.

3.3.1 *Experimental Setup*

The previously described synthetic benchmark is run on an Intel Core i5-6600K system running Linux 4.2. The first performance counter, PMC_0, counts `MEM_LOAD_RETIRED.L3_MISS` events while PMC_1 counts `LONGEST_LAT_CACHE.MISS` events for comparison. We disabled overflow interrupts, as well as any Linux subsystem that interacts with performance monitoring. The measurement tool samples both counters as well as `DRAM_DATA_READS` at a frequency of 1 kHz.

We disabled Hyperthreading and any kind of CPU frequency scaling. We also disable prefetchers, as those are not accounted for by either last-level cache event. The fourth core has been isolated using the `isolcpu=3` kernel parameter and only the benchmark is run there using `taskset`.

3.3.2 *Results*

Figure 1 shows a capture of the complete benchmark run. The plot has been annotated with the respective work area sizes (represented as $\log_2(\text{areasize})$) that were being used during each time interval. One can see that both events do indeed correlate with read memory accesses as reported by the memory controller. The stepwise increase of the memory read rate can be explained by taking into account that, depending on the replacement policy of the LLC, a portion of the loads can still be answered from the LLC until the area size gets sufficiently large.

Note that the benchmark initially writes some data to the whole allocated memory area to ensure that it is actually allocated by the operating system, which also causes memory reads to happen, explaining the bump at 2200 ms.

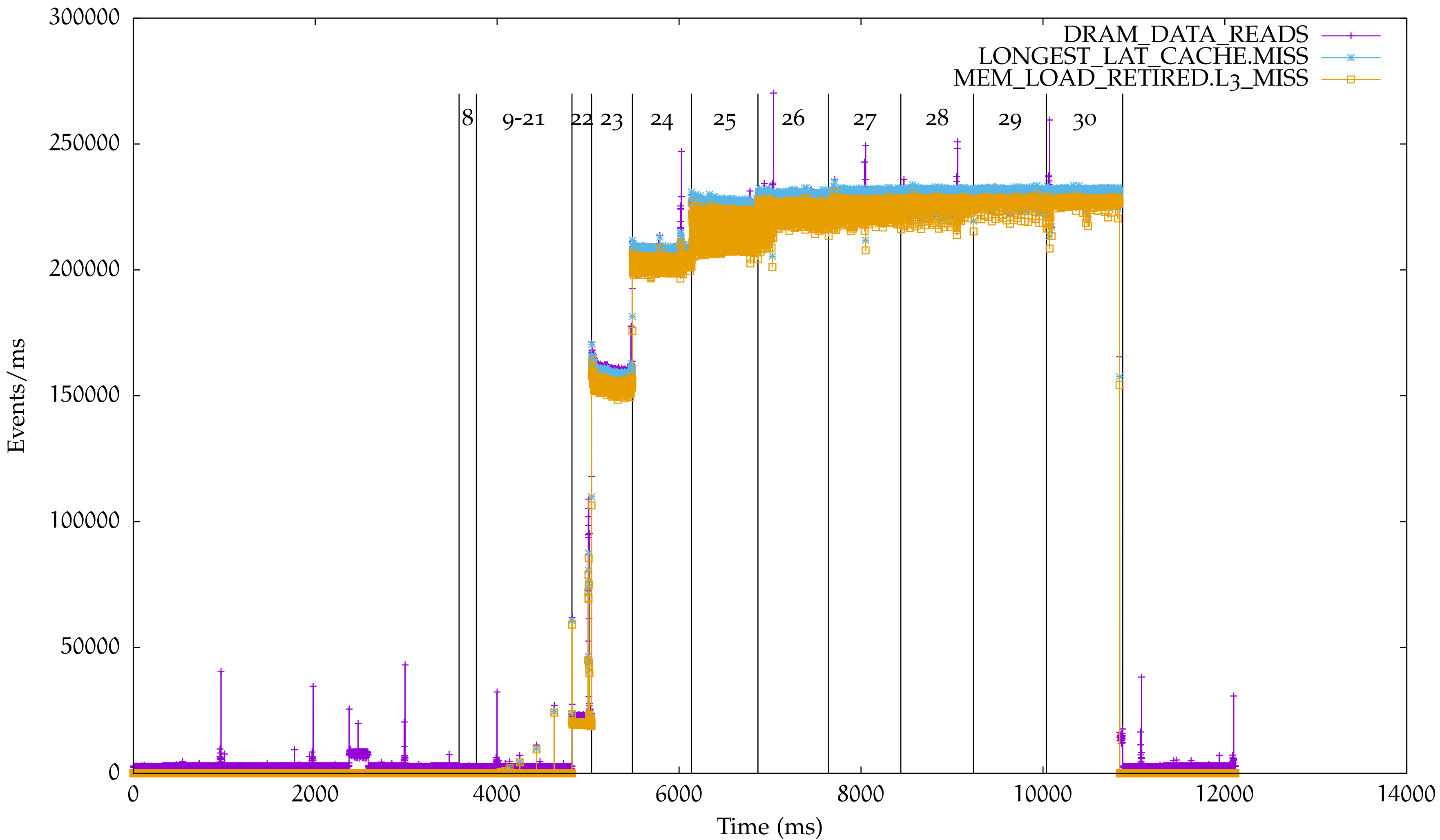

Figure 1: A complete benchmark run, annotations show $\log_2(\text{areasize})$.

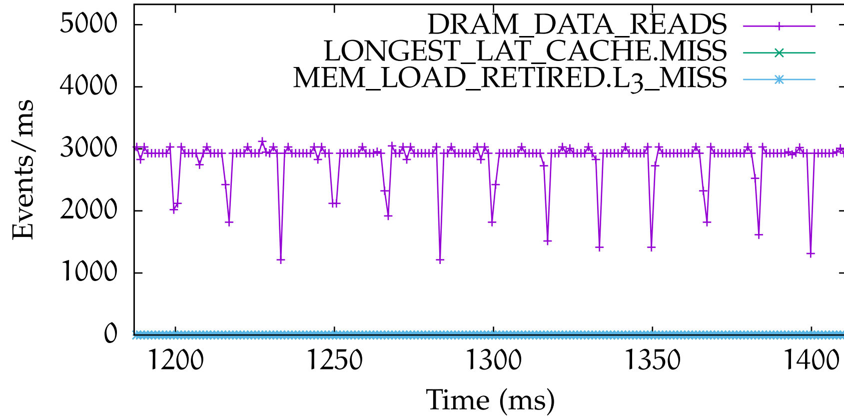

Figure 2: Detailed view of the idle portion of the benchmark run.

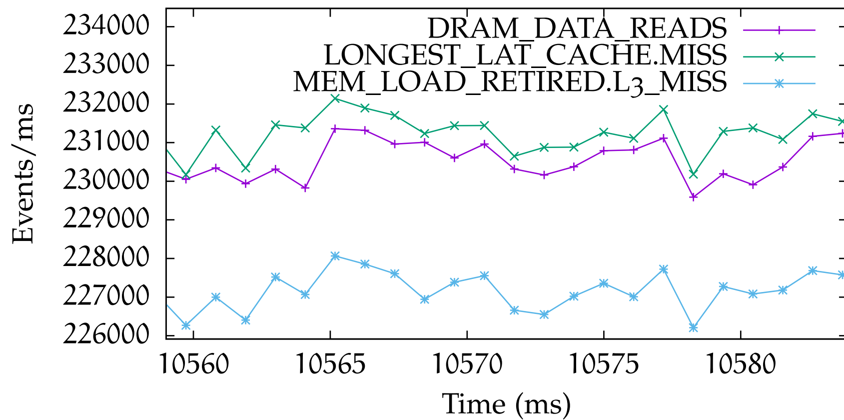

Figure 3: Detailed view of the main memory portion of the benchmark run.

Figures 2 and 3 show a detailed view of the same benchmark run during the idle phase and while accessing main memory, respectively. Figure 2 shows a relatively constant idle load of 3000 main memory reads per millisecond; as this already includes the load generated by the measurement framework and this load is independent of the benchmark, we expect it to carry over to the active part of the benchmark.

Figure 3 supports our assumption that both performance monitoring events correlate to main memory transactions caused by last-level cache misses. The offset of 3000 events per millisecond carries over as well. Curiously, the event rate of `LONGEST_LAT_CACHE.MISS` is higher than the rate of main memory transactions, which should not be possible and might be related to the erratum recommending against the use of this event.

To analyze the correspondence of `MEM_LOAD_RETIRED.L3_MISS` and `DRAM_DATA_READS` from a statistical viewpoint, we look at the difference between those values at each point in time during the active part

run	event count	equivalent memory traffic
1	1672675739	99.699 GiB
2	1672674122	99.699 GiB
3	1673078111	99.723 GiB

Table 1: MEM_LOAD_RETIRED.L3_MISS event counts after reading 100 GiB.

of the benchmark, that is, we subtract the number of retired memory loads which missed L3 from the number of reads reported by the memory controller. This yields a mean difference which is close to the idle values reported by the memory controller, the relative error being well below 1%. The maximum difference can be very high but this is caused by spurious activity elsewhere in the system which influences the values reported by the memory controller. In the 95th percentile of the difference, the error still is only about 1%.

To gain a better understanding of the predictive power of this event, we run a linear regression on MEM_LOAD_RETIRED.L3_MISS with respect to DRAM_DATA_READS, which yields an r^2 value of 0.9997, indicating that we can indeed approximate the number of memory read transactions caused by an application by monitoring this event.

3.4 further experiments

To further characterize the behavior of the MEM_LOAD_RETIRED.L3_MISS event, we run some additional experiments.

3.4.1 *Lower Bound on the Error*

Exactly assessing the accuracy of the event is difficult, as we usually have no a priori knowledge of the amount of cache misses a benchmark will cause. To remedy this, we now run a benchmark which sequentially reads from a one GiB area of memory in 64 byte increments, thus causing one cache miss per read. The large size of the memory area ensures that no caches will contain the address which is accessed next. This process is repeated 100 times, so a total of 100 GiB is transferred from main memory.

To determine a lower bound on the error introduced by measuring memory accesses with this event, we ignore the temporal behavior of the event and only compare the counter values before and after each run of the above benchmark.

Table 1 shows the results of a number of benchmark runs. We notice some variance, but it is low enough (COV = 0.014%) to be attributed to the inevitable nondeterminism present within the system. We can see that the event consistently undercounts by about 0.3%. The rea-

sons for this are not clear, but we expect this to be caused by load instructions that were dispatched by the benchmark, but retire after the processor has changed its execution mode either to ring 0 or system management mode, causing them not to be counted because the counter is programmed to only count ring 3/user-level events.

3.4.2 *Behavior on Split Cache Line Reads*

The description of the `MEM_LOAD_RETIRED.L3_MISS` event says that it counts instructions which caused a cache miss, not cache misses. This suggests that a situation where a single instruction causes multiple cache misses would cause us to miscount the number of cache misses and in turn lead to a wrong estimate of the subsequent main memory transactions.

X86 supports unaligned memory accesses. Thus, one such situation can be caused by unaligned loads which touch multiple cache lines.

Figure 4 shows a variant of the previous benchmark which reads 512 MiB from a one GiB area five times for each alignment relative to a 64 byte boundary using a word-sized (8 bytes) read instruction. As we are now interested in the behavior over time, we again use the sampling mechanism as described in Section 3.3.1. We can see that once part of the read crosses a cache line boundary, the event rate drops off sharply to a fifteenth of the usual rate. The architectural LLC miss event is not affected.

The Intel Optimization Reference Manual[10] says:

> An access to data unaligned on 64-byte boundary leads to two memory accesses and requires several µops to be executed (instead of one). Accesses that span 64-byte boundaries are likely to incur a large performance penalty, the cost of each stall generally are greater on machines with longer pipelines.

This suggests that an unaligned read gets split into two aligned read uops, only one of which will cause an event to be counted once the whole instruction retires. However, even in this case we would not expect the event count to drop so drastically in our benchmark, as the steady state of a sequential read operation would still only cause one cache miss per instruction.

However, cache line splits are rare enough not to cause excessive inaccuracies: An experiment showed that the ratio between `MEM_INST_RETIRED.SPLIT_LOADS` (counting memory instructions which involve split loads) and `MEM_INST_RETIRED.ALL_LOADS` (counting all load instructions) in the memory-intensive SPEC `lbm` benchmark[17] is lower than 1 in 350000.

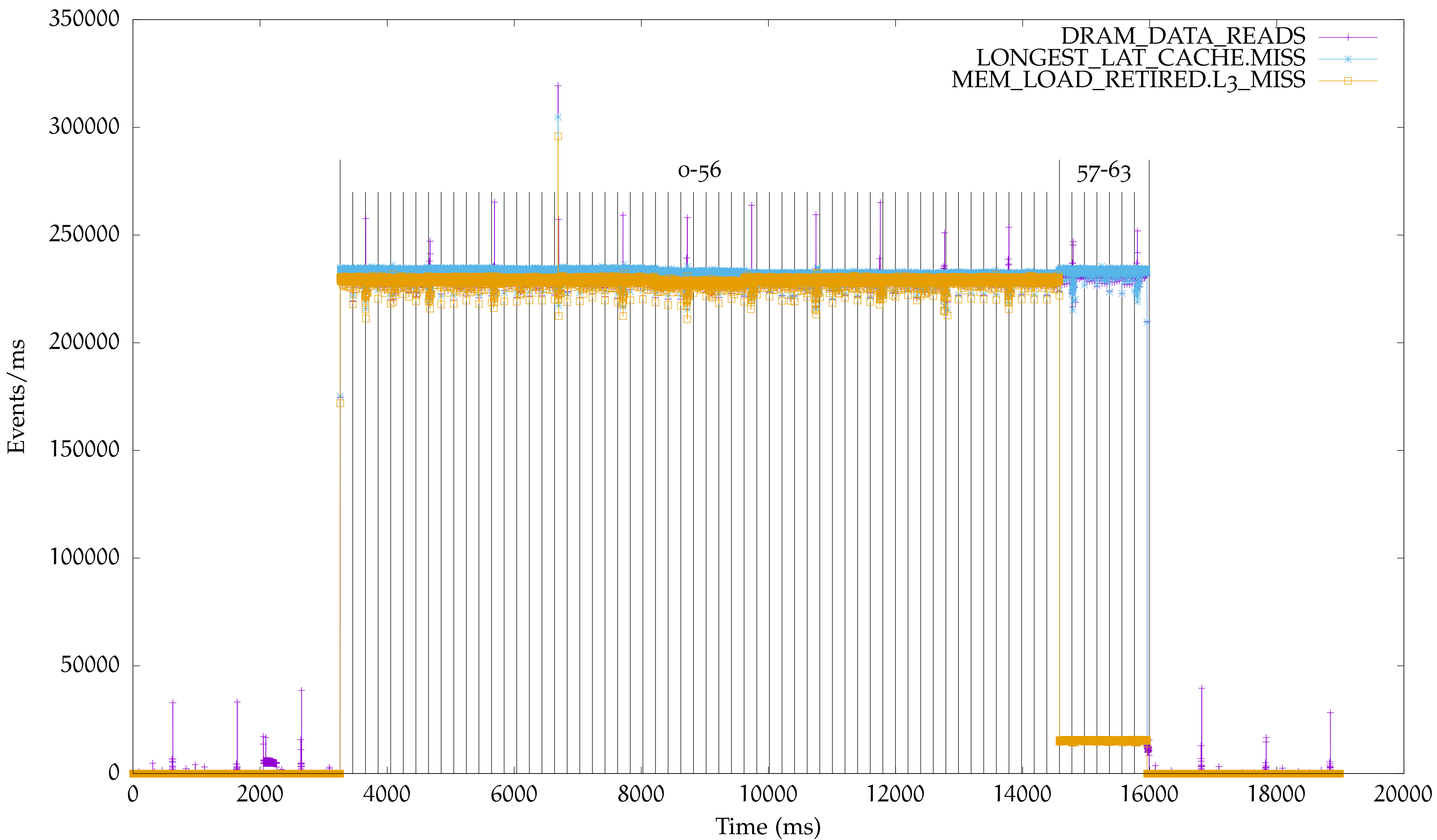

Figure 4: Counter behavior with respect to cache-line alignment. Annotations the show offset within a cache-line.

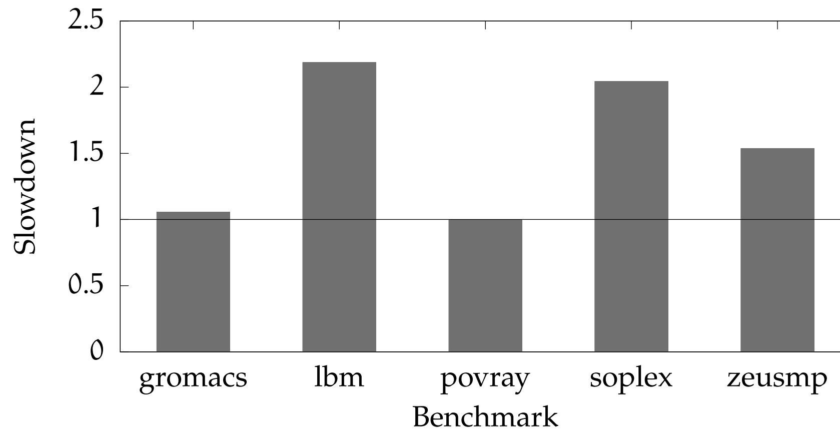

Figure 5: Slowdown caused by disabling prefetchers when running Spec benchmarks.

3.4.3 *Effects of Prefetchers*

In all our previous experiments, we followed the precedent set by previous works and disabled prefetchers. While doing so is certainly necessary for reaching our goal of very precise measurements, as prefetchers represent an unpredictable external factor in these measurements, it comes at a performance penalty for certain applications. While both effects are going to be very pronounced for simple synthetic benchmarks as the ones previously used, how strong are they going to be in practice?

To answer this question, we ran a selection of benchmarks from the Spec CPU2006 suite of benchmarks: `lbm`, `zeusmp` and `soplex`, which are relatively memory-intensive, and `gromacs` and `povray`, which are not [17], both with enabled prefetchers and without. We then compared both the run time and the total number of last-level cache miss events that occurred for each benchmark and prefetcher mode.

We find that prefetchers generally increase performance measured as the total run time of a benchmark in memory-intensive benchmarks, having little effect in the others (Figure 5). Curiously, `povray` runs marginally faster with disabled prefetchers.

However, enabling prefetchers does indeed reduce the accuracy of our memory bandwidth measurements: In each case, the event count drops by more than half, in the case of `gromacs` and `lbm` it drops by multiple orders of magnitude (Figure 6).

We thus conclude that disabling prefetchers, while undesirable for performance reasons, is nevertheless necessary when using last-level cache miss events for memory bandwidth monitoring.

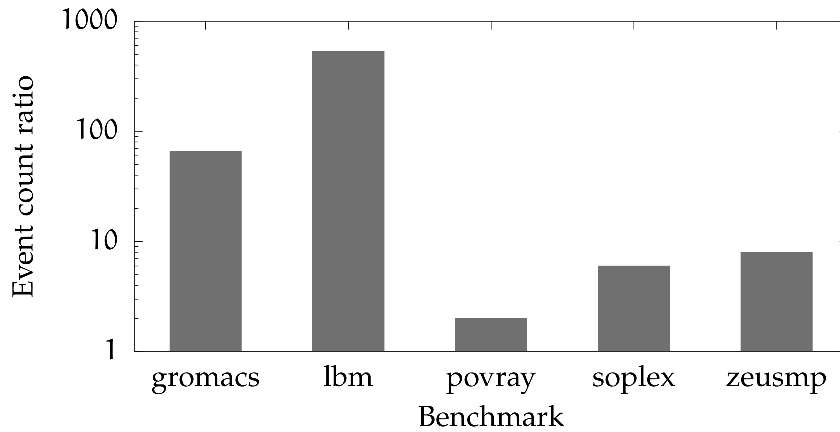

Figure 6: Number of events counted when running with prefetchers disabled divided by the number of events counted when running with prefetchers enabled (logarithmic scale).

3.5 CONCLUSION

While far from ideal, using `MEM_LOAD_RETIRED.L3_MISS` in combination with PEBS is currently our only option for implementing hardware-assisted memory bandwidth tracing. We hope that Intel might one day extend the PEBS mechanism to be usable with more or even all available performance monitoring events; this would not only suit our purpose better, but would probably also benefit those who use the performance monitoring subsystem for classical profiling.

4 DESIGN

We now propose a design for a framework in which we can apply PEBS for measuring memory bandwidth. As we also develop this framework with the intention that it will enable detailed study of the interactions between an application and the memory subsystem rather than to be used as a data source for an automated mechanism such as MemGuard, we settle for a design which will allow us to capture memory bandwidth traces in a live system for later analysis. Figure 7 shows a high-level overview of the proposed design. After summarizing our design goals in Section 4.1 we give a rough overview of our design in Section 4.2. In Section 4.3, we explain the reasons for some of the high-level design decisions. Finally, we describe each of the components of our design in more detail.

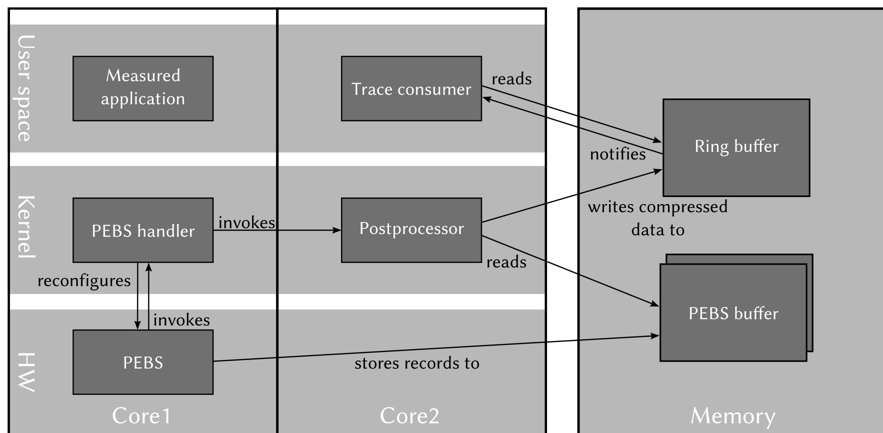

Figure 7: Design overview

4.1 DESIGN GOALS

In Section 3.1, we already defined some criteria which helped us select an appropriate hardware mechanism as the basis for our mechanism. The goals for our design reflect the same ideas: Our design should enable precise reporting of the measured memory bandwidth and the temporal accuracy should not suffer from the decisions made as part of this design.

As we also want to disturb the measured application as little as possible (low overhead), we would ideally completely separate the components involved in the measurement process from the measured application. However, this is not possible on multiple levels. As explained in Chapter 2, on-core performance counters can only be han-

dled on the same core, either using overflow interrupts or using PEBS; PEBS buffer-full interrupts are also handled locally. Thus, some disturbance is inevitable. Furthermore, as both the PEBS assist and our measurement framework will have to write to memory, they will put pressure on the caches and consume bandwidth on the on-chip interconnect. Our evaluation system only contains a single CPU with a single memory controller. While the memory controller supports multiple channels, it transparently maps addresses to both channels in an interleaved manner, which means that the memory accesses from the measurement framework will have to compete with the memory accesses of the measured application. While these effects are unavoidable, we should try to minimize their consequences.

As PEBS might not always be the preferred mechanism for measuring memory bandwidth, we should design our mechanism in a way that can be easily adapted to a new mechanism, should one arise. As PEBS itself is more generic than how we use it in the course of this thesis, it would also be desirable if our mechanism could be reused for other applications as well.

Our design should also be simple in the sense that it avoids unnecessary complexity.

4.2 DESIGN OVERVIEW

Our design requires a multi-core CPU and tries to divert as much work as possible from the core running the application to minimize interference. While there is no reason why it could not support multiplexing among multiple applications running on a single core, we chose not to implement this for the sake of simplicity. Thus, all user space activity on the measured core is accumulated. However, this is not a problem for our later evaluation, as we can force the kernel to only run the process of interest on that core. No part of our design precludes implementing multiplexing by integrating it with the operating system scheduler, should the need for such functionality some day arise.

While we only use a single core for running measurements in this thesis, we still symmetrically set up the measurement mechanism on each core to maintain flexibility and avoid special cases.

We use the PEBS mechanism to acquire traces, which stores individual PEBS records to a buffer in memory whenever a performance counter overflows. Once this buffer fills up, an interrupt is signaled on the same core, causing the PEBS handler to be executed. The handler reconfigures PEBS to use another buffer, sends a inter-processor interrupt (IPI) to invoke a postprocessor for the current buffer on another core and then immediately returns.

As we are only interested in when each overflow occurred, the trace postprocessor iterates over all PEBS records, extracting only the

timestamp field of each fixed-size record and applying some compression to further reduce the size of the generated data. This data is then exposed to a consumer running in user space via a lockless ring buffer. To relieve the consumer from having to continuously poll the ring buffer for new entries, we also include a notification mechanism, which allows the application to sleep until entries are available.

4.3 DESIGN RATIONALE

Other designs than the above are certainly possible. For example, one could directly map the PEBS buffers to user space and do all the processing there, which would save at least one copy operation. However, our main goal is achieving low overhead. For this, it is necessary that filled PEBS buffers are processed swiftly to avoid having to halt the application when no buffer space is available or having to drop records. Waking up a specific user space application and notifying it of which buffer to process would take longer than remaining in-kernel, as our proposed design does. The complexity with regard to locking would also be much higher.

Another variation could have been drop the user space component altogether, simply storing the collected traces by calling file system operations directly from the kernel code. However, doing this within Linux, the operating system chose to implement our design on, would have incurred much additional complexity. Additionally, it would have limited the flexibility of our approach. For example, should we decide that storing records to disk was not desirable anymore for efficiency or other reasons, switching an user space application to stream this data over a high-speed network interface would be easy; modifying an in-kernel implementation would be much more difficult.

In summary, we favor a design which is more simple, modular and obviously correct, rather than one which would theoretically be more efficient. Our experiments conducted during implementing this design by enabling and disabling the whole processing chain show that our design incurs next to no additional overhead to the one incurred by PEBS itself. Additionally, our design has already been used as a basis for another mechanism developed within our working group which analyzes load latencies. This would have been more difficult to integrate, if not for the loose coupling of these components. Thus, we believe these design decisions to having been sound.

4.4 PEBS CONFIGURATION

We use PEBS to acquire events via the PMC_0 performance counter. To remain generic and reduce complexity, this counter is completely con-

figured from user space, although in this thesis, we will always use the `MEM_LOAD_RETIRED.L3_MISS` event, as determined in Chapter 6.

PEBS allows for the configuration of an *interrupt threshold*, which is the number of records contained within the PEBS buffer after which an interrupt is raised. This is distinct from the size of the buffer, as there is a delay between this threshold being reached and the interrupt handler starting to execute, during which more records might be written. If not for a separate limit, these records would have to be discarded. We thus configure this threshold a little lower than the capacity of the PEBS buffer.

4.5 PEBS HANDLER

Whenever the PEBS buffer fills up to the interrupt threshold, an interrupt is raised and the PEBS interrupt handler is invoked.

Classically, all processing of PEBS records happens within this handler, as PEBS was not primarily designed to reduce overhead, but to enable precise snapshotting of the machine state whenever a performance counter overflows. However, as we want to minimize the overhead our approach incurs, we chose to deviate from the classical approach: We use a double buffering scheme, where we allocate two equally sized buffers for containing the PEBS records when initializing our framework and only switch the active buffer in the PEBS interrupt handler, submitting the now-frozen full buffer to a separate core for processing using an inter-processor interrupt.

While the Intel manual [11] only describes configuring the location of the PEBS record buffer while PEBS itself is disabled, reconfiguring it from the interrupt handler does seem to work in practice.

As the buffer postprocessor runs asynchronously from the core where the measurement is running, it could happen that the currently active buffer fills up and needs to be replaced before the postprocessor has finished processing the other buffer. To detect this situation, we use locks for each buffer. The only ways to handle this situation are dropping the contents of the active buffer or waiting for the postprocessor to finish. For our implementation, we decided that causing some additional overhead is a smaller problem than losing data. Thus, the PEBS handler will spin until the second buffer is available.

While using a buffering scheme with more than two buffers is possible, we decided against doing so, as two buffers turn out to generally suffice in practice and adding more buffers would just further increase the cache footprint of this mechanism.

4.6 POSTPROCESSING

The postprocessor fulfills the function that is traditionally implemented within the PEBS interrupt handler: It iterates over the PEBS record

0	64	127
EFLAGS	EIP	
EAX	EBX	
ECX	EDX	
ESI	EDI	
EBP	ESP	
R8	R9	
R10	R11	
R12	R13	
R14	R15	
IA32_PERF_GLOBAL_STATUS	Data Linear Address	
Data Source Encoding	Latency value	
EventingIP	TX Abort Information	
TSC		

Figure 8: Structure of a PEBS record as generated by the Skylake microarchitecture.

buffer, extracting the data of interest. To distribute the load, a core's postprocessor is run on the core numerically preceding it, modulo the number of cores.

Figure 8 shows the record format as of the Skylake microarchitecture. While some of the other fields might have interesting applications as well, the field which is the most useful for the purposes of this thesis is the TSC value.

However, the full width 64 bits is not necessary for exposing TSC values to the subsequent components in this design. As the absolute value of the TSC at the time a measurement is started is effectively random, we just store the difference between the TSC values of subsequent records. These deltas fit into 32 bits, especially at the low overflow thresholds we are interested in. Thus, we only store unsigned 32 bit deltas to the ring buffer. Should a delta not fit into 32 bits, the postprocessor will cap it to the maximum representable value. However, this does not happen in practice except during idle phases. Thus, the accuracy of our system is not affected.

Should the ring buffer not be able to accept another delta because it has reached its capacity, the postprocessor aborts and drops the contents of the whole PEBS buffer to give the consumer a chance to recover.

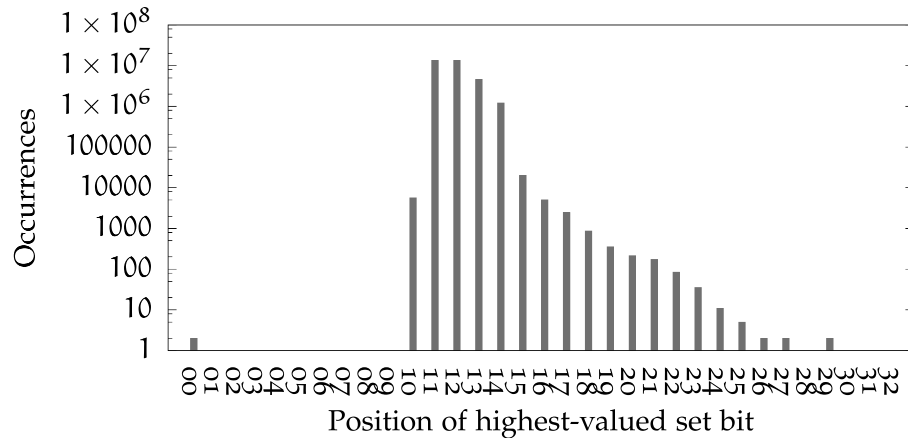

Figure 9: Distribution of timestamp deltas in a representative trace, on a logarithmic scale. The deltas are grouped by the number of the highest-valued set bit.

4.7 RING BUFFER

To transfer data to the trace consumer, our design uses a ring buffer in shared memory between the kernel and the trace consumer. There is one such buffer per core, which is only written to by the postprocessor assigned to that core and only read from by at most one consumer. Thus, the implementation of this buffer can be lockless, which simplifies the interface.

To prevent the consumer from having to poll the state of the ring buffer continuously, it also provides a notification mechanism, using which the consumer can block during periods of low traffic, to conserve CPU time.

4.8 TRACE CONSUMER

The trace consumer is a user space program which attaches to the ring buffer for a specific core and consumes the data this buffer provides. In our implementation, the consumer simply writes this data to disk, but one can imagine other variants, which, for example, provide real time analysis of the gathered data.

To further reduce the size used by each datum and thus reduce the necessary bandwidth to store the generated traces, we apply an additional step of lossless compression. For this, we use a simplistic variable length encoding scheme, which, while consuming little CPU time, still yields a compression of about 50%, compared to our initial experiments with the LZ4 compression algorithm [8], which only achieved a compression to 66% of the original size, while also consuming too much CPU time for streaming data without dropouts.

We chose the parameters for this compression scheme based on statistics gathered from the traces we commonly encountered during

testing our implementation (Figure 9): More than 99% of all entries can be contained within 15 bit. Thus, we use the following representation, based on 16 bit words:

```
0x0000 - 0x7fff: 0b0xxxxxxxxxxxxxxx

0x8000 - 0x7ffffffe: 0b1xxxxxxxxxxxxxxx 0bxxxxxxxxxxxxxxxx

0x7fffffff-0xffffffff: 0xffff 0xffff 0xxxxx 0xxxxx
```

This means that entries below 2^{15} are stored literally in one 16 bit word and entries below $2^{31} - 1$ are stored as two words, where the extended length is signaled by the highest valued bit being set in the first word. The remaining entries are stored literally in two words, prefixed by two all-ones escape words.

4.9 CONVERSION BETWEEN OVERFLOW-BASED AND SAMPLING-BASED REPRESENTATIONS

Our approach records the time stamps of when a specific event, namely the overflow of a performance counter, occurs. This is fundamentally different from approaches such as the one used in Chapter 3, where the value of a performance counter is sampled at regular points in time: While an overflow-based approach with a threshold of 1 yields equivalent data to a sampling-based approach with a sufficiently high sample rate, the outputs of both approaches diverge as the threshold and sampling interval get higher.

As representing the data as the number of events over time is more natural for manual inspection, we need a way to convert the time stamp traces generated by our approach to such a representation. This can be achieved by binning:

1. Select the desired equivalent sampling interval (t_s).
2. Divide the time interval covered by the trace into intervals of length t_s.
3. For each such interval: Count the number of trace time stamps within this interval and multiply this number by the threshold that was used when capturing the trace.

The end result of this process is a trace which is similar to one captured by sampling a performance counter at regular intervals, but where the number of events at each point in time is quantized to multiples of the threshold.

5 IMPLEMENTATION

In this chapter we describe the most important aspects of our implementation of the design proposed in Chapter 4.

This chapter is structured as follows: We justify the choice of Linux as the basis for our implementation in Section 5.1 and give some considerations on how we chose to integrate our design with it in Section 5.2. In Section 5.3 and Section 5.4, we describe the configuration of PEBS and the corresponding Performance Counters, respectively. We give details about the implementation of the PEBS interrupt handler in Section 5.5. Finally, we explain the interface between the kernel module and the user space components in Section 5.6 and detail the implementation of the trace consumer in Section 5.7.

5.1 CHOICE OF OPERATING SYSTEM

We chose to implement our design as an extension to the Linux kernel, for multiple reasons: Linux is an open-source operating system, making it easy to extend and modify compared to closed-source operating systems, such as Microsoft Windows. Compared to other open-source operating systems, Linux is also popular in cloud applications and scientific computing, an area where a system such as ours might be applied for analyzing application behavior. Finally, Linux was the only open-source operating system supporting the Skylake generation of Intel CPUs and the associated chipset at the time we implemented our design.

5.2 OPERATING SYSTEM INTEGRATION

We implemented our mechanism as a kernel module instead by modifying the kernel itself. While this requires some more work, it also affords easier development, as code modifications do not require a reboot, and eases distribution, as the module can be independently compiled on another system without requiring a kernel patch to be applied, which would require a rebuild of the whole kernel.

While Linux already contains the perf subsystem for performance monitoring, which includes tracing facilities and also supports PEBS, we decided against basing our implementation on perf, as it incurs a lot of complexity and includes some mechanisms, such as automatically adjusting overflow thresholds depending on the event load caused by the measured application, which are useful for classical performance analysis, but not for implementing our design. Addition-

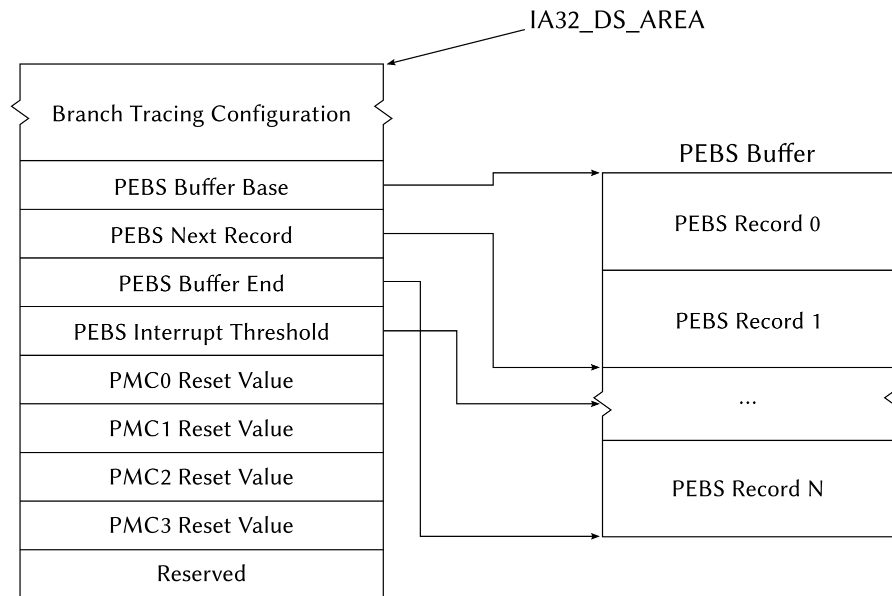

Figure 10: Layout of the Debug Store Area

ally, the PEBS support is very limited, only supporting buffers which can contain exactly one record. Again, this is good enough for using PEBS to accurately capture the state of an application that is being profiled, but does not allow for the implementation of our design.

While it would have been possible to extend perf to be sufficiently generic for implementing our design, this would neither have been worthwhile for the purposes this thesis, nor would it have been an efficient use of the time allotted.

The downside of this approach is that using perf while our implementation is active will yield unpredictable results.

5.3 PEBS CONFIGURATION

PEBS is configured via the Debug Store (DS) Area, which is a per-CPU in-memory data structure, which also contains the configuration for other tracing mechanisms, such as branch tracing. The location of this area is communicated to the CPU using the `IA32_DS_AREA` MSR. Figure 10 shows the layout of the Debug Store Area: *PEBS Buffer Base* contains the address of the PEBS buffer and *PEBS Buffer End* contains the address of the first byte after the end of the PEBS buffer. *PEBS Next Record* contains the address where the next PEBS record will be stored. If this pointer exceeds the value stored in *PEBS Interrupt Threshold*, an interrupt is raised. Whenever a performance counter overflows and PEBS is active for that counter, PEBS resets the counter's value to the reset value specified in the Debug Store Area.

The perf subsystem already configures a DS area on boot. As our implementation needs to control the contents of the DS area, it allocates a new one and only modifies the `IA32_DS_AREA` MSR to point

to the new structure, instead of modifying the existing one. This has the advantage that we only need to restore the original `IA32_DS_AREA` value for deactivating our mechanism.

Our design requires two PEBS buffers to be preallocated per core, where the size of the buffers represents a tradeoff between reducing the number of PEBS overflow interrupts over time (large buffers), and reducing the cache footprint caused by these buffers (small buffers). Additionally, larger buffers incur a longer delay between the time a record is written to the buffer by the PEBS assist and the time we can analyze its contents, but this latency is not a concern when just capturing traces. The PEBS buffer size can be changed via a kernel module parameter, but we found a default value of 3 MiB (corresponding to 15728 PEBS records) to minimize the incurred slowdown in our benchmarks.

5.4 PERFORMANCE COUNTER CONFIGURATION

As the configuration of performance counters is independent of the configuration and handling of PEBS, our design proposes not to include facilities for configuring performance counters in the PEBS kernel module. Instead, it expects them to be set up externally.

In our implementation, we do this by configuring the `PERFEVTSEL` MSRs, which control the individual performance counters, using user space scripts. To access these MSRs, we use the `msr` kernel module and the `wrmsr` command line tool.

In our setup, we only count user space (ring 3) events to avoid measuring kernel activity, which also includes activity caused by our measurement framework, by setting bit 16 (`USR`), but not bit 17 (`OS`), which would enable ring 0 counting, in the `PERFEVTSEL` register.

5.5 PEBS HANDLER

PEBS interrupts are delivered on the same core via the per-core interrupt controller, the Local APIC (LAPIC). The Local Vector Table Performance Counter Register (LVTPC) of the APIC determines how the interrupt is delivered by determining the delivery mode (normal or non-maskable interrupt, for example) and the vector number to which the interrupt is delivered. Similarly to the DS area, we save the original value of the LVTPC and restore it afterwards.

One complication with our approach is that Linux does not expose an interface to register new interrupt vectors in the Interrupt Descriptor Table (IDT) directly. As our implementation is a kernel module, we cannot simply patch the code that initially fills the IDT. While the unprivileged `SIDT` instruction would reveal the linear address of the IDT, Linux creates a read-only mapping of the IDT at a fixed linear address and stores this address in the IDT location register to avoid

leaking kernel addresses and to prevent the modification of IDT entries (which is precisely what we need to do) to limit the scope of security vulnerabilities in the kernel. Thus, similarly to the `pmu-tools` package [12], we resort to resolving the address of the IDT using the optional *kallsyms* symbol lookup mechanism, making running a kernel built with the `CONFIG_KALLSYMS_ALL` option mandatory. We can then patch the IDT and register our interrupt handler.

The interrupt handler itself exchanges the buffers as described in Section 4.5, using spin locks to prevent concurrent buffer access with the postprocessor. The postprocessor is then invoked on another core using the asynchronous Linux SMP remote function call interface.

5.6 USER SPACE INTERFACE

The user space interface exposed by our implementation has two facets, which are both exposed via the *debugfs* virtual filesystem: A control and statistics interface, and the per-core ring buffers.

The statistics interface exposes per-core statistics of the number of PEBS buffers and records processed, the number of ring buffer overflows and the number of PEBS records dropped as a result, and the number of deltas which exceeded a 32 bit range.

As PEBS buffers are usually only processed when they are full, it might be the case that, at the end of a measurement run, there are still outstanding records in the currently active PEBS buffer. Thus, the control interface also exposes a file that, when written to, forces the active buffer to be exchanged and processed, whether or not it is completely filled.

Each ring buffer is represented by a single debugfs file which can be memory-mapped using the `mmap()` system call. This allows the transfer of data from the kernel to user space without any additional system calls, which is necessary to achieve a sufficiently high throughput when streaming the event traces. To allow the consumer application to go to sleep when no data is pending and avoid wasting CPU cycles on polling the buffer, the file also implements support for the `poll()` system call, which blocks the application until new data is inserted into the buffer.

Again, buffer sizing represents a tradeoff, but for the ring buffer, 256 KiB, corresponding to the contents of about four PEBS buffers, suffice not to cause any overflows in practice.

5.7 TRACE CONSUMER

The trace consumer of our implementation compresses the trace entries it receives and stores them to disk for later inspection. However, some optimizations were necessary to make it fast enough to

store traces without creating overflows in the preceding buffers of the pipeline.

Firstly, it uses the `sched_setscheduler()` syscall to put itself into the FIFO realtime scheduling class, preventing it from being interrupted by any ordinary process. Secondly, it reserves enough space for the trace using `fallocate()` and then maps that file to memory using `mmap()`. Both operations make it difficult to dynamically resize the file during runtime, so the user has to guess an appropriate size beforehand. Should the file turn out to be too small, the consumer logs a message to that effect and drops further incoming data. If the file is too large, it can be truncated afterwards.

The consumer polls the ring buffer, consuming elements until no further ones are available. If the ring buffer remains empty after multiple attempts to read from it, the consumer uses `poll()` to block until new entries appear.

Ideally, the consumer is placed on the same core that runs the post-processor, to exploit L2 and L1 caches when reading from the ring buffer. While this could be forced using CPU affinity, the CPU migration mechanism of the Linux kernel automatically places the consumer on the correct core.

6 EVALUATION

In this chapter, we evaluate our previously developed mechanism with respect to our main design goals: Accuracy, and low overhead.

This chapter is structured as follows: First, we describe our experimental setup in Section 6.1 and a software-only approach previously developed in our working group in Section 6.2. This approach will serve as a basis for evaluating whether our new mechanism actually yields an improvement over classical mechanisms for measuring main memory bandwidth consumption.

We then proceed to evaluate the accuracy of our approach in Section 6.3 and the overhead in Section 6.4. We describe a relationship between overhead and accuracy in Section 6.5. Finally, we quantify the maximally achievable throughput of our solution in Section 6.6.

6.1 EXPERIMENTAL SETUP

We run all of the following measurements on a system with a Intel Core i5-6600K processor. Both memory channels are in use and are each populated by two 8 GiB DDR4 SDRAM modules. For trace recording, we use a 250 GB SATA 6.0 Gb/s SSD with a nominal write rate of 500 megabytes per second, formatted with the ext4 file system.

We disabled any kind of CPU frequency scaling such as *SpeedStep* and *TurboBoost* as a precaution against unpredictable processing speed changes during the benchmark runs. We also disable prefetchers, in accordance with Section 3.4.3. As hyperthreading has been known to cause issues with respect to performance monitoring[1], we also turned it off. Of the four available physical cores, the fourth has been isolated using the `isolcpu=3` kernel parameter and we only run the benchmark there using `taskset`.

6.2 BASELINE SOFTWARE-BASED APPROACH

The earlier approach developed in our working group (henceforth called `soft-tsc`) also uses performance monitoring events to measure memory bandwidth. However, contrary to our new approach (`pebs-extract`), it does not use a hardware assisted tracing mechanism which also includes timestamps, but rather implements timestamping manually using overflow interrupts. It is also implemented as an extension to the Linux kernel and will thus be easily comparable to our new implementation. More precisely, it works as follows:

A trace buffer is preallocated per core and exposed to user space. User space communicates the desired event threshold to the kernel and then configures a performance counter as desired, enabling overflow interrupts. It then once preloads the counter so that it will overflow after `threshold` events and enables the counter.

Once an overflow happens, an interrupt is signaled. The interrupt handler reads the current TSC value and stores it to the trace buffer. It then resets the performance counter value and returns from the interrupt.

While this approach is straightforward and does not require any hardware support beyond basic performance monitoring, we expect it to have a number of disadvantages:

OVERHEAD As each performance counter overflow causes an interrupt and a switch from user to kernel mode, a high overhead is incurred, which rises with decreasing threshold values.

ACCURACY We expect the high interrupt load to distort the measurements more strongly than our approach.

6.3 ACCURACY

To check whether our PEBS-based approach (called `pebs-extract`) has retained the accuracy of the `MEM_LOAD_RETIRED.L3_MISS` event as determined in Section 3.4.1, we reuse the benchmark from that section, which causes 100 GiB of read traffic from main memory to occur. The read loop in this benchmark is unrolled, so that 32 load instructions are issued before a branch instruction occurs to ensure the pipeline is filled with loads most of the time.

We use the first performance counter, PMC_0, for PEBS, but also configure the same event to PMC_1 without activating PEBS on this counter, for comparison. For this experiment, we disabled any post-processing of PEBS records, that is, the PEBS handler only increments a record counter in the statistics interface, resets the write index within the PEBS buffer to zero and immediately returns. We do this to be able to argue more clearly that the effects we observe are due to PEBS itself and not due to our implementation. However, according to further experiments not detailed here, enabling postprocessing does not change the behavior we observe in the following sections.

To count the original number of events seen by `pebs-extract`, we count the total number of PEBS records written and multiply it by the configured event threshold, after which such a record is written. Running the above benchmark at various thresholds, we observe that the number of events reported by PEBS drops from 96% of the expected amount at a threshold of 1000 to 37% at a threshold of 10. Furthermore, the number of events reported by PEBS differs from

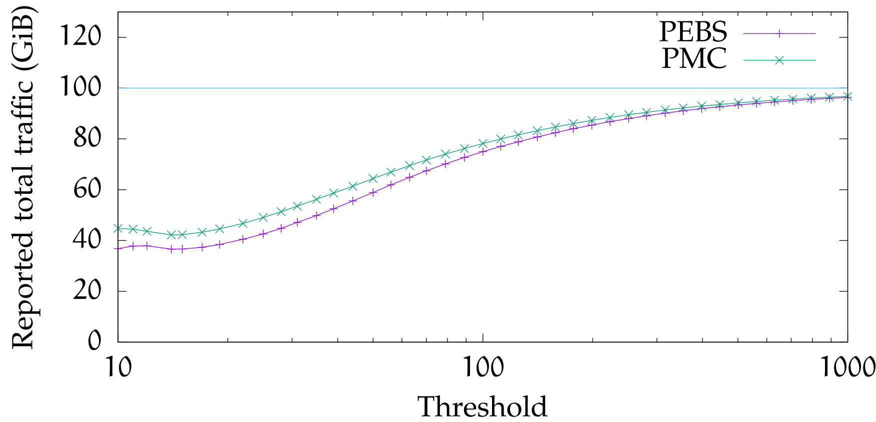

Figure 11: Inaccuracies incurred by using PEBS when measuring 100 GiB of reads.

the number of events as counted by the regular performance counter running in parallel, which also loses events (Figure 11).

Ideally, all values should be at 100 GB, but as Section 3.4.1 has demonstrated, a minor error of about 0.3% is to be expected. However, the results we have now obtained differ by more than 3% in the best case and by more than 60% in the worst case, at the lowest threshold.

6.3.1 *Discussion*

Clearly, the cause of this inaccuracies must lie within PEBS itself: The PEBS handler gets called only seldomly and does almost no work, as postprocessing has been disabled.

The Intel Software Development Manual says:

> When the hardware needs the assistance of microcode to deal with some event, the machine takes an assist. [. . .] Assists clear the entire machine of ops before they begin and are costly. ([11], Section 18.13.6)

This means that loads which have been dispatched but not retired at the point the PEBS assist activates will be aborted and reissued once the assist completes. This alone would not cause us to miscount events, as those loads will just complete after being reissued. However, as the original loads already triggered memory fetches, those complete even if the load instruction itself is subsequently aborted. This means that the requested data is already present in the cache hierarchy when the load instructions are reissued. These load instructions will then not cause an L3 miss anymore, causing us to undercount.

Indeed, we observe most of the missing L3 misses as L1 hits: If we run the same benchmark, but additionally count the number of

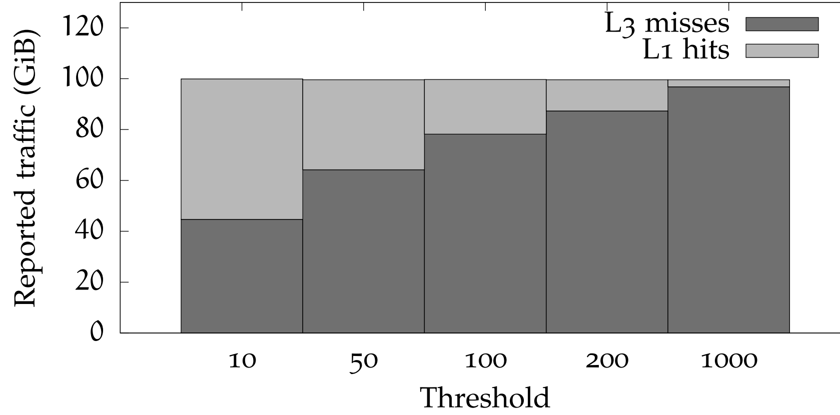

Figure 12: Migration of L3 miss events towards L1 hits at lower PEBS thresholds in the 100 GiB read benchmark.

THRESHOLD	PEBS	PMC1
PEBS disabled	–	1677914879 (= 100%)
10	78.33%	100%
50	92.94%	100%
100	95.75%	100%
200	97.87%	100%
1000	99.54%	100%

Table 2: Number of MEM_INST_RETIRED.ALL_LOADS events caused by the 100 GiB read benchmark, using the raw event count when run with PEBS disabled as a reference.

L1 hits using the MEM_LOAD_RETIRED.L1_HIT event in PMC2, the error virtually disappears, supporting this assumption (Figure 12).

While this explains the reduction of event counts of the raw L3 miss event in Figure 11, it is still unclear why the PEBS-based mechanism counts even fewer events than the simple counter. Running the same benchmark, but measuring an event which is not influenced by the above cache hierarchy effects shows this more clearly. In Table 2, we show the results of measuring the event MEM_INST_RETIRED.ALL_LOADS, which represents all retired loads, regardless of which element of the memory hierarchy supplied the data. While this is not useful for measuring main memory bandwidth in a real application, it demonstrates that PEBS still looses events relative to the raw count captured in PMC1, which does not lose events in this setup. Thus, we are observing an additional effect, which is independent of the one just described.

In Section 2.1.1 we described the algorithm for using performance counters with overflow interrupts. A crucial part of that algorithm

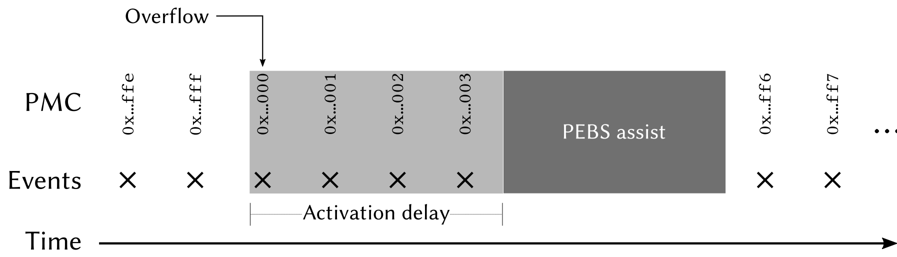

Figure 13: Activation delay. In this illustration, PEBS is configured for a threshold of 10.

was to read the current value of the performance counter from within the interrupt handler, as events continue to occur and get counted until the interrupt handler code finally begins executing. Presumably, something similar happens with PEBS (Figure 13): According to [11] (Section 18.4.2), PEBS only gets "armed" when a counter overflow happens; the "PEBS event", which will then cause the PEBS assist to execute, gets issued when the next underlying event occurs. We assume that there additionally is a small delay between the PEBS event being issued and the assist actually starting execution. Unlike the previously described algorithm based on overflow interrupts however, the PEBS assist does neither record the current value of the performance counter at the time of its execution, nor does it apply this value as an offset to the value the counter is reset to at the end of the assist. Thus, the events which occur during this *activation delay* are irretrievably lost.

6.3.2 *Upper Bound on Lost Events*

Both the loss incurred by the machine clear caused by the PEBS assist and the loss incurred by the activation delay are dependent on the measured application: Both the number of loads in flight at the time the PEBS assist activates, for the first effect, and the rate of retirements during the activation delay in the second case are variable. However, in both cases, the effect gets worse with a higher rate of loads which will miss L3. Thus, the above benchmark should represent a worst case, as nearly all its instructions are loads that will miss L3, allowing us to deduce an upper bound for the incurred error.

When the event rate is constant, we can model both effects as causing the loss of a constant number l of events whenever the event count reaches the threshold t. During each run of the benchmark, approximately the same volume v of events should occur causing a activations of the PEBS assist. Then,

$$\frac{v - a * l}{t} = a$$

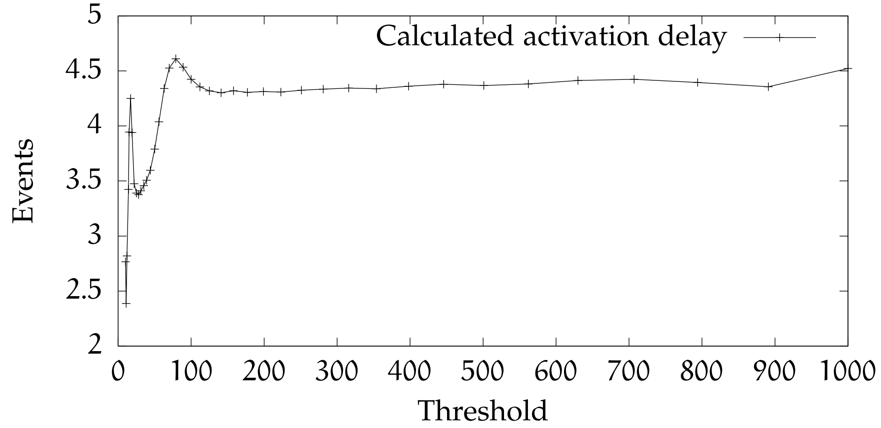

Figure 14: Estimated activation delay from our worst-case benchmark.

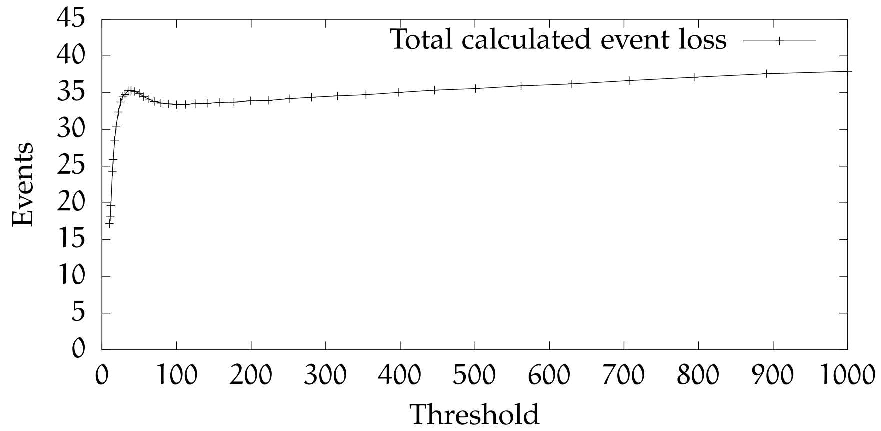

Figure 15: Estimated total loss per PEBS assist activation in our worst-case benchmark.

holds, which allows us to calculate the loss as

$$l = \frac{v}{a} - t,$$

where a is the number of activations that has actually been observed.

The results from the experiment in Table 2 allow us to estimate the number of events lost to the activation delay. Figure 14 shows the results of this estimation with more data points within the same threshold range. While our model breaks down for low thresholds, it yields a stable activation delay of 4 to 5 for higher thresholds.

Similarly, the number of events lost to the combination of both effects can be calculated from the data gathered in the first experiment, beginning at 17 and averaging out at about 35 events for higher thresholds (Figure 15). There is a slight linear upwards trend which continues at higher thresholds; however, this is primarily caused by the small loss incurred by the event itself (see Section 3.4.1). Clearly, the effect caused by the machine clear dominates.

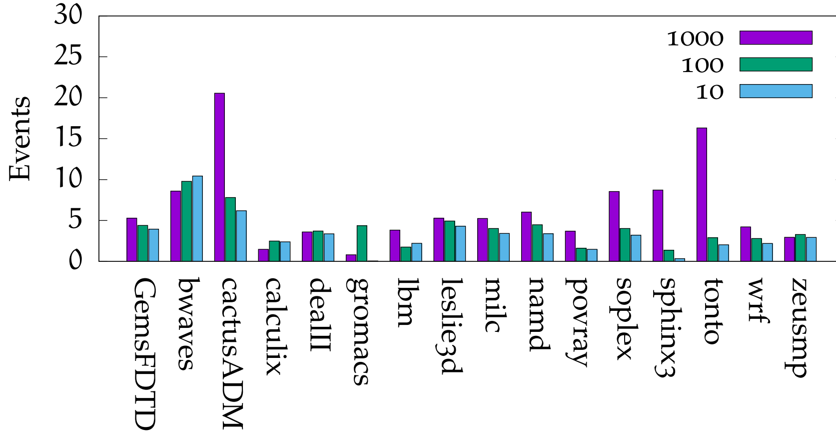

Figure 16: Total losses per PEBS assist activation in SPEC benchmarks at different thresholds.

To see whether this is indeed an upper bound in practical applications, we run a similar analysis using the SPECFP 2006 set of benchmarks (the `gamess` benchmark from that suite failed to run correctly on our machine and was thus excluded). As we have no a priori value for the number of main memory accesses caused by these benchmarks, we first run them without PEBS, counting the number of L3 misses using a regular performance counter to establish a baseline. Then, we run each benchmark with PEBS and apply the above formula.

We find that the worst-case loss encountered by these benchmarks are indeed less than the ones incurred by our assumed worst-case benchmark, reaching only 20 events in the `cactusADM` benchmark (Figure 16).

These inevitable effects limit the usefulness of our approach and make it mostly useless for our initial intent of precisely capturing the memory access characteristics of an application. However, we now compare it to `soft-tsc` and find that the latter suffers from similar problems.

6.3.3 *Comparison to the software approach*

When running our synthetic benchmark under `soft-tsc`, we also observe a loss of events.

`soft-tsc` samples the current performance counter value after interrupt entry has completed and thus can avoid the loss incurred by the activation delay. We can confirm this by comparing the value it reports to the value reported by an independent performance counter running in parallel, noting that they match closely.

However, similar to PEBS, we also observe a migration of L3 misses to L1 hits with `soft-tsc` at lower thresholds, virtually identical in quantity to the one observed with PEBS (see Figure 12).

Thus, our approach incurs only a slight additional loss compared to `soft-tsc`, caused by the activation delay. We now compare both approaches with regard to their overhead and find that our approach significantly improves upon `soft-tsc`.

6.4 OVERHEAD

To evaluate the overhead of our approach under realistic circumstances, we again use benchmarks from the SPEC suite. At various thresholds, we compare the run time of each benchmark when observed by our mechanism to the run time of the benchmark when run unobserved. For comparison, we repeat the same measurements with `soft-tsc`.

We find that there is virtually no difference between both approaches at a threshold of 1000, but that the `soft-tsc` causes a significantly higher overhead at lower thresholds, up to a slowdown of 150% compared to a slowdown of 50% incurred by `pebs-extract` at threshold 10 for the `bwaves` and `milc` benchmarks (see Figure 17). While the effects described in Section 6.3 skew this comparison in favor to our new approach by losing events and thus causing fewer invocations of the PEBS assist and the PEBS interrupt handler, our approach still yields a net gain when factoring in these effects: In the case of `lbm` at threshold 10, `soft-tsc` only counted 13% more counter overflows than the PEBS-based solution, but still was 50% slower. Thus, our approach affords lower thresholds, which means better accuracy, at the same slowdown or a lower slowdown at a similar accuracy when compared to a software-only approach.

6.5 ACCURACY IMPLICATIONS OF OVERHEAD

The overhead incurred per counter overflow with a particular mechanism also influences the accuracy of that mechanism: As the number of overflows per unit of time varies, so does the incurred overhead and as such, the slowdown of the application. This results in a compression or dilation of different phases in a trace. Figure 18 shows the access patterns of a benchmark which alternates between phases of high and low L3 miss rates as recorded by both `pebs-extract` and `soft-tsc`, scaled in time to show corresponding phases. One can both see that the benchmark takes longer to run with `soft-tsc` and that the phases of high event rates are stretched compared to `pebs-extract`, which has a lower per-overflow overhead.

Furthermore, `soft-tsc` causes an interrupt per counter overflow compared to `pebs-extract`, which only causes an interrupt per PEBS

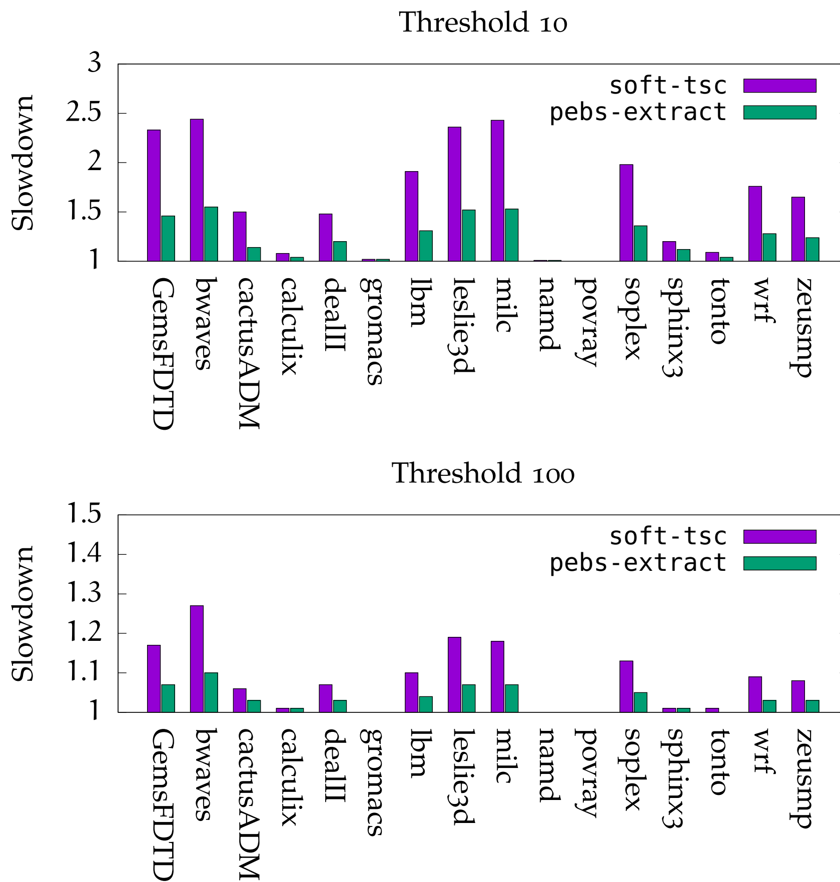

Figure 17: Slowdown of Spec benchmarks caused by `soft-tsc` and our solution at different thresholds. Threshold 1000 is not plotted, as no significant slowdown occurs with either approach.

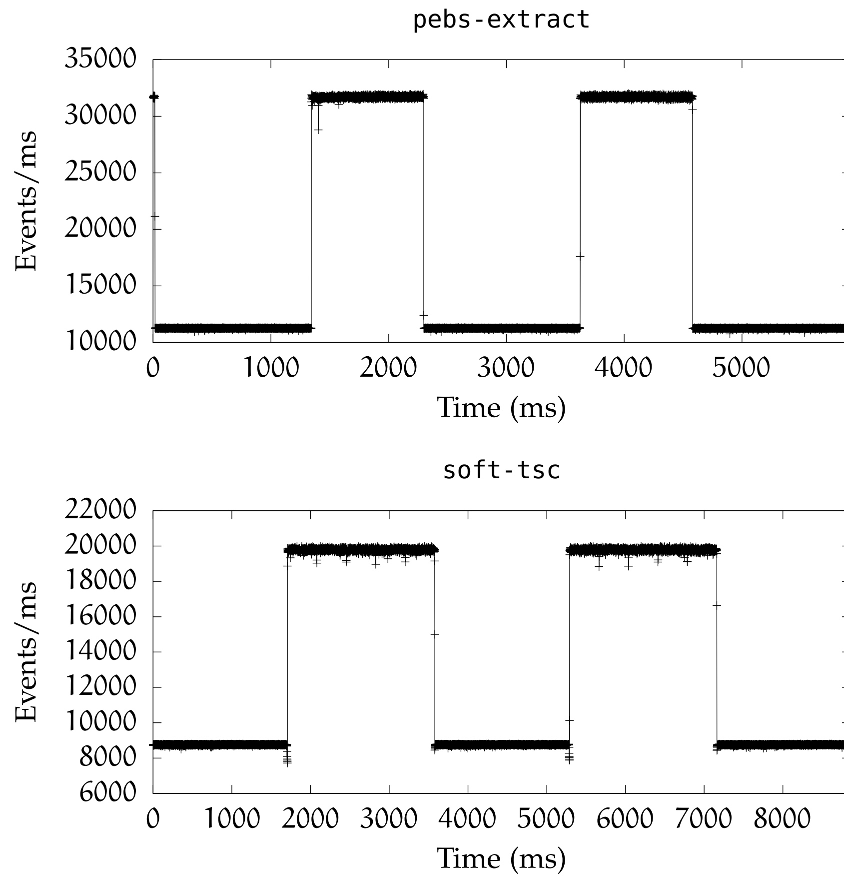

Figure 18: Stretching of phases of high event rates with `soft-tsc` compared to `pebs-extract`, at threshold 10.

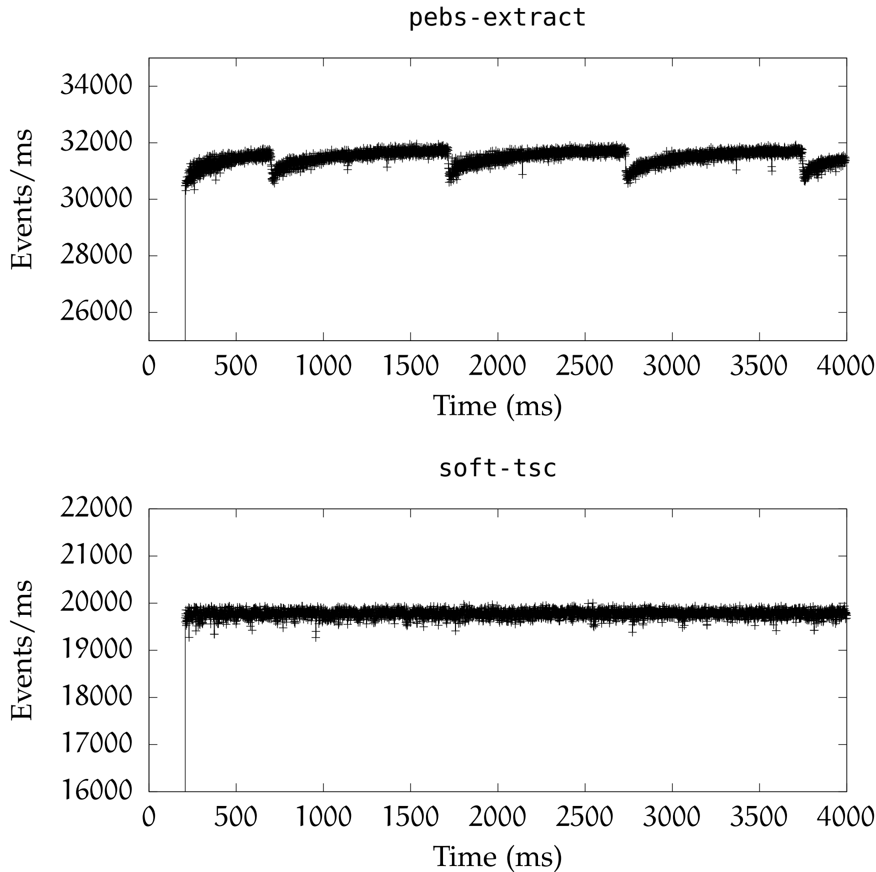

Figure 19: Artificial IPI interference in the 100 GiB read benchmark, which is not visible with `soft-tsc`.

buffer overflow. To demonstrate the relevance of this difference, we run the 100 GiB read benchmark but additionally set up another core to cause a short burst of inter-processor-interrupts to the core running the application every second.

Figure 19 shows traces of this setup as captured by either mechanism. One can see regular drops in the event rate in the trace captured by `pebs-extract`, but not in the one captured by `soft-tsc`, where this effect remains invisible.

6.6 ACHIEVABLE THROUGHPUT

As our new solution—contrary to `soft-tsc`—also contains a mechanism for streaming traces and storing them to disk, we also need to evaluate whether this mechanism represents a bottleneck when trying to capture traces with a high data rate.

We were not able to create a situation where any workload, even at low thresholds, caused our mechanism to drop events because the trace consumer could not keep up with storing them to disk.

However, this is in part a result of the aforementioned event loss caused by PEBS. To be able to determine a theoretical maximum event rate, we modified our approach to simulate varying event rates by replacing the PEBS overflow interrupt as a trigger for postprocessing by a high-resolution timer and modified the postprocessor to place artificial deltas in the ring buffer. This enabled us to evaluate the feasibility of our postprocessing chain without having to create a workload which would cause a sufficiently high load.

We found that, with our experimental setup, our implementation can easily handle 4000 PEBS buffer-full interrupts per second, corresponding to an event rate of about $630 * 10^6$ events per second at a threshold of 10. If each event represents a LLC cache miss, this would be sufficient for monitoring transfer rates of up to 37 GiB per second, which is close to the theoretical maximum of 37.5 GiB per second of a DDR4 dual-channel system. However, in practice, our evaluation system only reaches a throughput of about 32 GiB per second in the best case, even with prefetchers enabled, as measured by pmbw [6].

Thus, our proposed solution achieves sufficient throughput for all practical scenarios.

7 CONCLUSION

Main memory bandwidth is a shared resource which is more difficult to account for than resources under the explicit management of the operating system, such as CPU time or IO bandwidth. Nevertheless, the total memory bandwidth is limited and applications can thus influence each other through this bottleneck.

In this thesis, we developed a system for capturing fine-grained memory bandwidth consumption traces for applications in a live system with a focus on achieving a high accuracy while maintaining low overhead and interference. To that end, we first analyzed the available technologies, settling on Precise Event-Based Sampling (PEBS) in combination with last-level cache miss counters as the basis for our system. We also determined that the assumption that last-level cache miss counters can be used to measure the rate of main memory accesses still broadly holds on modern CPU architectures: Split cache line reads are a cause of undercounting, but they are rare enough not to cause significant errors. However, disabling prefetchers is necessary to achieve accurate measurements and does often reduce application performance significantly.

We proposed a modular design based on PEBS for efficiently streaming counter overflow timestamp traces and detailed our Linux-based implementation.

The evaluation of this implementation revealed both a fundamental issue with the PEBS mechanism and with using last-level cache miss counters in combination with both PEBS and classical approaches based on overflow interrupts: On the one hand, PEBS does not sample the current value of a performance counter when executing the PEBS assist, which causes it to lose events due to the PEBS activation delay. On the other hand, both the activation of the PEBS assist and a classical counter overflow interrupt abort pending loads, causing a migration of L3 misses towards L1 hits.

Based on these observations, we conclude that the currently available hardware mechanisms are not sufficient for reaching our goal of capturing fine-grained main memory bandwidth traces. However, we were able to demonstrate that hardware-assisted tracing does indeed help reduce the overhead such a mechanism incurs: At threshold 10, our solution reduced the overhead by 20% on average and never increased the overhead.

We also demonstrated that lower overhead improves the accuracy by virtue of causing less interference with the measured application during periods of high application activity and that our approach

also reveals application behavior which was previously masked by the high interrupt load of classical interrupt-based approaches.

7.1 FUTURE WORK

Whether accurate high-resolution memory bandwidth measurement will become feasibly hinges on the development of better hardware support for doing so. Intel Memory Bandwidth Monitoring (MBM) seems to be a step in the right direction; however, it does not include any kind of tracing support, requiring manual sampling. For future hardware generations, a combination of a mechanism such as PEBS with MBM would be promising. Extending PEBS records to include a sample of the counter value at the time the assist activates would avoid the loss of events incurred by the PEBS activation delay.

Monitoring memory write bandwidth using performance monitoring remains largely unexplored because the relationship between cache events and actual memory traffic is more complex. Investigating this relationship would also be worthwhile, although all approaches based on performance counters will ultimately suffer from similar issues as the ones detailed in this work.

Part I

APPENDIX

BIBLIOGRAPHY

[1] *6th Generation Intel® Core™ Processor Family Specification Update*. Intel. Aug. 2015.

[2] *An Introduction to the Intel QuickPath Interconnect*. Intel.

[3] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator." In: *USENIX Annual Technical Conference, FREENIX Track*. 2005, pp. 41–46.

[4] Frank Bellosa. *Process Cruise Control: Throttling Memory Access in a Soft Real-Time Environment*. Tech. rep. TR-I4-97-02. July 1997. URL: http://os.itec.kit.edu/.

[5] SUNY Binghamton. "MARSSx86—Micro-ARchitectural and System Simulator for x86-based Systems." In: *State University of New York at Binghamton, Binghamton, NY* (2013).

[6] T Bingmann. *Parallel Memory Bandwidth Benchmark*. 2013. URL: http://panthema.net/2013/pmbw.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. "The gem5 simulator." In: *ACM SIGARCH Computer Architecture News* 39.2 (2011), pp. 1–7.

[8] Yann Collet. *Lz4: Extremely fast compression algorithm*. 2013. URL: http://lz4.org/.

[9] Agner Fog. "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs." In: *Denmark (Lyngby): Technical University of Denmark* (2016).

[10] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel. Sept. 2015.

[11] *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 3: System Programming Guide*. Intel. June 2015.

[12] Andi Kleen. *pmu-tools*. URL: https://github.com/andikleen/pmu-tools.

[13] *Lightweight Profiling Specification*. AMD. Aug. 2010.

[14] *Monitoring Integrated Memory Controller Requests in the 2nd, 3rd and 4th generation Intel® Core™ processors*. Intel. Mar. 2013.

[15] *The Compute Architecture of Intel® Processor Graphics Gen9*. Intel.

[16] Vish Viswanathan. *Disclosure of H/W prefetcher control on some Intel processors*. Sept. 2014. URL: https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.

[17] Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui Sha. "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms." In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. Apr. 2013, pp. 55–64. DOI: 10.1109/RTAS.2013.6531079.