Investigating Cache Parameters of x86 Family Processors

Vlastimil Babka and Petr Tůma

Department of Software Engineering
Faculty of Mathematics and Physics, Charles University
Malostranské náměstí 25, Prague 1, 118 00, Czech Republic
`{vlastimil.babka|petr.tuma}@dsrg.mff.cuni.cz`

Abstract. The excellent performance of the contemporary x86 processors is partially due to the complexity of their memory architecture, which therefore plays a role in performance engineering efforts. Unfortunately, the detailed parameters of the memory architecture are often not easily available, which makes it difficult to design experiments and evaluate results when the memory architecture is involved. To remedy this lack of information, we present experiments that investigate detailed parameters of the memory architecture, focusing on such information that is typically not available elsewhere.

1 Introduction

The memory architecture of the x86 processor family has evolved over more than a quarter of a century – by all standards, an ample time to achieve considerable complexity. Equipped with advanced features such as translation buffers and memory caches, the architecture represents an essential contribution to the overall performance of the contemporary x86 family processors. As such, it is a natural target of performance engineering efforts, ranging from software performance modeling to computing kernel optimizations.

Among such efforts is the investigation of the performance related effects caused by sharing of the memory architecture among multiple software components, carried out within the framework of the Q-ImPrESS project[1]. The Q-ImPrESS project aims to deliver a comprehensive framework for multicriterial quality of service modeling in the context of software service development. The investigation, necessary to achieve a reasonable modeling precision, is based on evaluating a series of experiments that subject the memory architecture to various workloads.

In order to design and evaluate the experiments, a detailed information about the memory architecture exercised by the workloads is required. Lack of information about features such as hardware prefetching, associativity or inclusivity

[1] This work is supported by the European Union under the ICT priority of the Seventh Research Framework Program contract FP7-215013 and by the Czech Academy of Sciences project 1ET400300504.

could result in naive experiment designs, where the workload behavior does not really target the intended part of the memory architecture, or in naive experiment evaluations, where incidental interference between various parts of the memory architecture is interpreted as the workload performance.

Within the Q-ImPrESS project, we have carried out multiple experiments on both AMD and Intel processors. Surprisingly, the documentation provided by both vendors for their processors has turned out to be somewhat less complete and correct than necessary – some features of the memory architecture are only presented in a general manner applicable to an entire family of processors, other details are buried among hundreds of pages of assorted optimization guidelines. To overcome the lack of detailed information, we have constructed additional experiments intended specifically to investigate the parameters of the memory architecture. These experiments are the topic of this paper.

We believe that the experiments investigating the parameters of the memory architecture can prove useful to other researchers – some performance relevant aspects of the memory architecture are extremely sensitive to minute details, which makes the investigation tedious and error prone. We present both an overview of some of the more interesting experiments and an overview of the framework used to execute the experiments – Section 2 focuses on the parameters of the translation buffers, Section 3 focuses on the parameters of the memory caches, Section 4 presents the framework.

After a careful consideration, we have decided against providing an overview of the memory architecture of the x86 processor family. In the following, we assume familiarity with the x86 processor family on the level of the vendor supplied user guides [1,2], or at least on the general programmer level [3].

1.1 Experimental Platforms

For the experiments, we have chosen two platforms that represent common servers with both Intel and AMD processors, further referred to as Intel Server and AMD Server.

Intel Server A server configuration with an Intel processor is represented by the Dell PowerEdge 1955 machine, equipped with two Quad-Core Intel Xeon CPU E5345 2.33 GHz (Family 6 Model 15 Stepping 11) processors with internal 32 KB L1 caches and 4 MB L2 caches, and 8 GB Hynix FBD DDR2-667 synchronous memory connected via Intel 5000P memory controller.

AMD Server A server configuration with an AMD processor is represented by the Dell PowerEdge SC1435 machine, equipped with two Quad-Core AMD Opteron 2356 2.3 GHz (Family 16 model 2 stepping 3) processors with internal 64 KB L1 caches, 512 KB L2 caches and 2 MB L3 caches, integrated memory controller with 16 GB DDR2-667 unbuffered, ECC, synchronous memory.

To collect the timing information, the RDTSC processor instruction is used. In addition to the timing information, we collect the values of the performance counters for events related to the experiments using the PAPI library [4] running on top of perfctr [5]. The performance events supported by the platforms are described in [1, Appendix A.3] and [6, Section 3.14]. For overhead incurred by the measurement framework, see [7].

Although mostly irrelevant, both platforms are running Fedora Linux 8 with kernel 2.6.25.4-10.fc8.x86_64, gcc-4.1.2-33.x86_64, glibc-2.7-2.x86_64. Only 4 level paging with 4 KB pages is investigated.

1.2 Presenting Results

To illustrate the results, we typically provide plots of values such as the duration of the measured operation or the value of a performance counter, typically plotted as a dependency on one of the experiment parameters. Durations are expressed in processor clocks. On Platform Intel Server, a single clock tick corresponds to 0.429 ns. On Platform AMD Server, a single clock tick corresponds to 0.435 ns.

To capture the statistical variability of the results, we use boxplots of individual samples, or, where the duration of individual operations approaches the measurement overhead, boxplots of averages. The boxplots are scaled to fit the boxes with the whiskers, but not necessarily to fit all the outliers, which are usually not related to the experiment. Where boxplots would lead to poorly readable graphs, we use lines to plot the trimmed means.

When averages are used in a plot, the legend of the plot informs about the details. The *Avg* acronym is used to denote standard mean of the individual observations – for example, *1000 Avg* indicates that the plotted values are standard means from 1000 operations performed by the experiment. The *Trim* acronym is used to denote trimmed mean of the individual observations where 1 % of minimum and maximum observations was discarded – for example, *1000 Trim* indicates that the plotted values are trimmed means from 1000 operations performed by the experiment. The acronyms can be combined – for example, *1000 walks Avg Trim* means that observations from 1000 walks performed by the experiment were the input of a standard mean calculation, whose outputs were the input of a trimmed mean calculation, whose output is plotted.

Since the plots that use averages do not give information about the statistical variability of the results, we point out in text those few cases where the standard deviation of the results is above 0.5 processor clock cycles or 0.2 performance event counts.

2 Investigating Translation Buffers

On Platform Intel Server, the translation buffers include an instruction TLB (ITLB), two levels of data TLB (DTLB0, DTLB1), a cache of the third level paging structures (PDE cache), and a cache of the second level paging structures

(PDPTE cache). On Platform AMD Server, the translation buffers include two levels of instruction TLB (L1 ITLB, L2 ITLB), two levels of data TLB (L1 DTLB, L2 DTLB), a cache of the third level paging structures (PDE cache), a cache of the second level paging structures (PDPTE cache), and a cache of the first level paging structures (PML4TE cache). The following table summarizes the basic parameters of the translation buffers on the two platforms, with the parameters not available in vendor documentation *emphasized*.

Table 1. Translation Buffer Parameters

Buffer	**Entries**	**Associativity**	**Miss [cycles]**
Platform Intel Server			
ITLB	128	4-way	*18.5*
DTLB0	16	4-way	2
DTLB1	256	4-way	*+7*
PDE cache	*present*		*+4*
PDPTE cache	*present*		*+8*
PML4TE cache	*not present*		N/A
Platform AMD Server			
L1 ITLB	32	full	*4*
L2 ITLB	512	4-way	*+40*
L1 DTLB	48	full	*5*
L2 DTLB	512	4-way	*+35*
PDE cache	*present*		*+21*
PDPTE cache	*present*		*+21*
PML4TE cache	*present*		*+21*

We begin our translation buffers investigation by describing experiments targeted at the translation miss penalties, which are not available in vendor documentation.

2.1 Translation Miss Penalties

The experiments we perform are based on measuring durations of memory accesses using various access patterns, constructed to trigger hits and misses as necessary. Underlying the construction of the patterns is an assumption that accesses to the same address generally trigger hits, while accesses to different addresses generally trigger misses, and the choice of addresses determines which part of the memory architecture hits or misses.

Due to measurement overhead, it is not possible to measure the memory accesses alone. To minimize the distortion of the experiment results, the measured workload should perform as few additional memory accesses and additional processor instructions as possible. To achieve this, we create the access pattern in advance and store it in memory as the very data that the measured workload accesses. The access pattern forms a chain of pointers and the measured workload uses the pointer that it reads in each access as an address for the next access. The workload is illustrated in Listing 1.1.

Listing 1.1. Pointer walk workload.

```
// Variable start is initialized by an access pattern generator
uintptr_t *ptr = start;

for (int i = 0; i < loopCount; i++)
    ptr = (uintptr_t *) *ptr;
```

Experiments with instruction access use a similar workload, replacing chains of pointers with chains of jump instructions. A necessary difference from using the chains of pointers is that the chains of jump instructions must not wrap, but must contain additional instructions that control the access loop. To achieve a reasonably homogeneous workload, the access loop is partially unrolled, as presented in Listing 1.2.

Listing 1.2. Instruction walk workload.

```
// The jump_walk function contains the jump instructions
int len = loopCount / 16;

while (len --)
    jump_walk (); // The function is invoked 16 times
```

To measure the translation miss penalties, the experiments need to access addresses that miss in TLB but hit in the L1 cache. This is done by accessing addresses that map to the same associativity set in TLB but to different associativity sets in the L1 cache. With a TLB of size S and associativity A mapping pages of size P, the associativity set is selected by $\log_2(S/A)$ bits starting with bit $\log_2(P)$ of the virtual address. Similarly, with a virtually indexed L1 cache of size S and associativity A caching lines of size L, the associativity set is selected by $\log_2(S/A)$ bits starting with bit $\log_2(L)$ of the virtual address. The two groups of bits can partially overlap, making a choice of an associativity set in TLB limit the choices of an associativity set in the L1 cache. We generate an access pattern that addresses a single associativity set in TLB and chooses a random associativity set of the available sets in the L1 cache.

The code of the set collision access pattern generator is presented in Listing 1.3 and accepts these parameters:

- $numPages$ The number of different addresses to choose from.
- $numAccesses$ The number of different addresses to actually access.
- $pageStride$ The stride of accesses in units of page size.
- $accessOffset$ Offset of addresses inside pages when not randomized.
- $accessOffsetRandom$ Tells whether to randomize offsets inside pages.

Listing 1.3. Set collision access pattern generator.

```
// Create array of pointers to the allocated pages
uintptr_t **pages = new (uintptr_t *) [numPages];
for (int i = 0; i < numPages; i++)
    pages [i] = (uintptr_t *) buf + pageStride * PAGE_SIZE;

// Cache line size is considered in units of pointer size
int numOffsets = PAGE_SIZE / LINE_SIZE;

// Create array of offsets in a page
offsets = new int [numPageOffsets];
for (int i = 0 ; i < numPageOffsets ; i++)
    offsets [i] = i * cacheLineSize;

// Randomize the order of pages and offsets
random_shuffle (pages, pages + numPages);
random_shuffle (offsets, offsets + numOffsets);

// Create the pointer walk from pointers and offsets
uintptr_t *start = addresses [0];
if (accessOffsetRandom) start += offsets [0];
else start += accessOffset;

uintptr_t **ptr = (uintptr_t **) start;
for (int i = 1 ; i < numAccesses ; i++) {
    uintptr_t *next = addresses [i];
    if (accessOffsetRandom) next += offsets [i % numOffsets];
    else next += accessOffset;

    (*ptr) = next;
    ptr = (uintptr_t **) next;
}

// Wrap the pointer walk
(*ptr) = start;
delete [] pages;
```

2.2 Experiment: TLB miss penalties

For every DTLB present in the system, the experiments that determine the penalties of translation misses use the set collision pointer walk from Listing 1.1 and 1.3 with *pageStride* set to number of entries divided by associativity, *numPages* set to a value higher than associativity and *numAccesses* varying from 1 to *numPages*. When *numAccesses* is less than or equal to associativity, all accesses should hit, afterwards the accesses should start missing, depending on the replacement policy. For ITLBs, we analogically use a jump emitting version of code from Listing 1.3 with the code from Listing 1.2.

Since the plots that illustrate the results for each TLB are similar in shape, we include only representative examples and comment the results in writing. All plots are available in [7].

Starting with an example of a well documented result, we choose the experiment with DTLB0 on Platform Intel Server, which requires *pageStride* set to 4 and *numAccesses* varying from 1 to 32. The results on Fig. 1 contain both the average access duration and the counts of the related performance events. We see that the access duration increases from 3 to 5 cycles at 5 accessed pages. At the same time, the number of misses in DTLB0 (DTLB_MISSES.L0_MISS_LD events) increases from 0 to 1, but there are no DTLB1 misses (DTLB_MISSES:ANY events). The experiment therefore confirms the well documented parameters of DTLB0 such as the 4-way associativity and the miss penalty of 2 cycles [1, page A-9]. It also suggests that the replacement policy behavior approximates LRU for our access pattern.

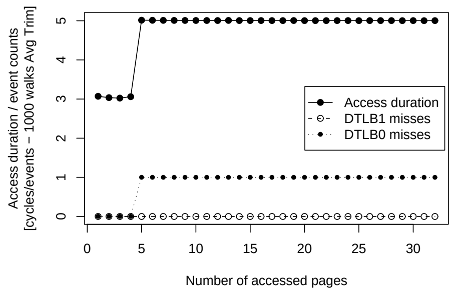

Fig. 1. DTLB0 miss penalty and related performance events on Intel Server.

Experimenting with DTLB1 on Platform Intel Server requires changing the *pageStride* parameter to 64 and yields an increase in the average access duration from 3 to 12 cycles at 5 accessed pages. Figure 2 shows the counts of the related performance events, attributing the increase to DTLB1 misses and confirming the 4-way associativity. Since there are no DTLB0 misses that would hit in the DTLB1, the figure also suggests non-exclusive policy between DTLB0 and DTLB1. The experiment therefore estimates the miss penalty, which is not available in vendor documentation, at 7 cycles. Interestingly, the counter of cycles spent in page walks (PAGE_WALKS:CYCLES events) reports only 5 cycles per access and therefore does not fully capture this penalty.

As an additional information not available in vendor documentation, we can see that exceeding the DTLB1 capacity increases the number of L1 data cache references (L1D_ALL_REF events) from 1 to 2. This suggests that page tables are cached in the L1 data cache, and that the PDE cache is present and the page table accesses hit there, since only the last level page walk step is needed.

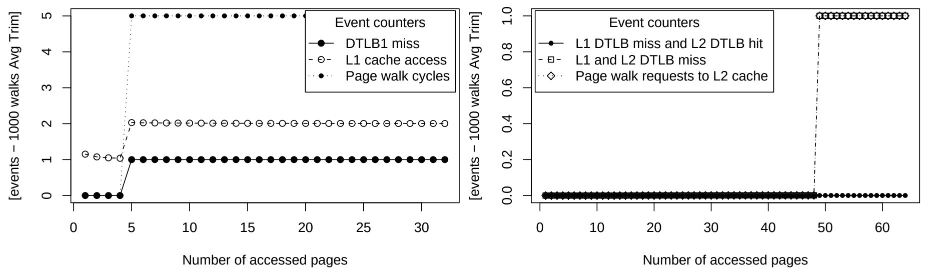

Fig. 2. Performance event counters related to L1 DTLB misses on Intel Server (left) and L2 DTLB misses on AMD Server (right).

Experimenting with L1 DTLB on Platform AMD Server requires changing *pageStride* to 1 for full associativity. The results show a change from 3 to 8 cycles at 49 accessed pages, which confirms the full associativity and 48 entries in the L1 DTLB, the replacement policy behavior approximates LRU for our access pattern. The performance counters show a change from 0 to 1 in the L1 DTLB miss and L2 DTLB hit events, the L2 DTLB miss event does not occur. The experiment therefore estimates the miss penalty, which is not available in vendor documentation, at 5 cycles. Note that the value of L1 DTLB hit counter (L1_DTLB_HIT:L1_4K_TLB_HIT) is always 1, indicating a possible problem with this counter on the particular experiment platform.

For L2 DTLB on Platform AMD Server, *pageStride* is set to 128. The results show an increase from 3 to 43 cycles at 49 accessed pages, which means that we observe L2 DTLB misses and also indicates a non-exclusive policy between L1 DTLB and L2 DTLB. The L2 associativity, however, is difficult to confirm due to full L1 associativity. The event counters on Fig. 2 show a change from 0 to 1 in the L2 miss event (L1_DTLB_AND_L2_DTLB_MISS:4K_TLB_RELOAD event). The penalty of the L2 DTLB miss is thus estimated at 35 cycles in addition to the L1 DTLB miss penalty, or 40 cycles in total.

On Platform AMD Server, the paging structures are not cached in the L1 cache. The value of the REQUESTS_TO_L2:TLB_WALK event counter shows that each L2 DTLB miss in this experiment results in one page walk step that accesses the L2 cache. This means that a PDE cache is present, as is further examined in the next experiment. Note that the problem with the value of the L1_DTLB_HIT:L1_4K_TLB_HIT event counter persists, it is always 1 even in presence of L2 DTLB misses.

2.3 Additional Translation Caches

Our experiments targeted at the translation miss penalties indicate that a TLB miss can be resolved with only one additional memory access, rather than as many accesses as there are levels in the paging structures. This means that that a cache of the third level paging structures is present on both investigated platforms, and since the presence of such additional translation caches mentioned only discussed in general terms in vendor documentation [8], we investigate these caches next.

2.4 Experiment: Extra translation buffers

With the presence of the third level paging structure cache (PDE cache) already confirmed, we focus on determining the presence of caches for the second level (PDPTE cache) and the first level (PML3TE cache).

The experiments use the set collision pointer walk from Listing 1.1 and 1.3. The *numAccesses* and *pageStride* parameters are initially set to values that make each access miss in the last level of DTLB and hit in the PDE cache. By repeatedly doubling *pageStride*, we should eventually reach a point where only a single associativity set in the PDE cache is accessed, triggering misses when *numAccesses* exceeds the associativity. This should be observed as an increase of the average access duration and an increase of the data cache access count during page walks. Eventually, the accessed memory range $pageStride \times numPages$ exceeds the 512×512 pages translated by a single third level paging structure, making the accesses map to different entries in the second level paging structure and thus different entries in the PDPTE cache, if present. Further increase of *pageStride* extends the scenario analogically to the PML4TE cache.

The change of the average access durations and the corresponding change in the data cache access count for different values of *pageStride* on Platform Intel Server are illustrated in Fig. 3. Only those values of *pageStride* that lead to different results are displayed, the results for the values that are not displayed are the same as the results for the previous value.

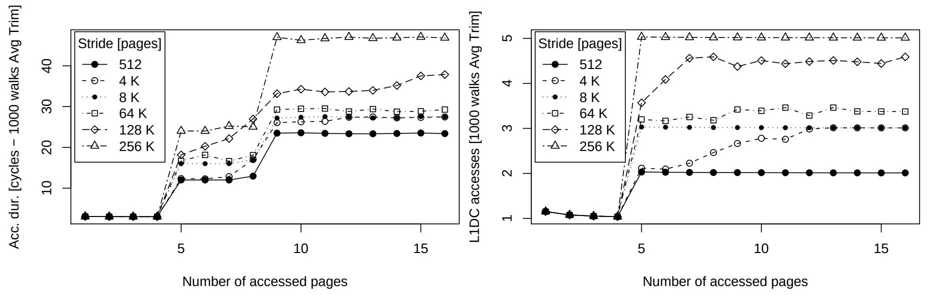

Fig. 3. Extra translation caches miss penalty (left) and related L1 data cache references events related (right) on Intel Server.

For the 512 pages stride, the average access duration changes from 3 to 12 at 5 accessed pages, which means we hit the PDE cache as in the previous experiment.

We also observe an increase of the access duration from 12 to 23 cycles and a change in the L1 cache miss (L1D_REPL event) counts from 0 to 1 at 9 accessed pages. These misses are not caused by the accessed data but by the page walks, since with this particular stride and alignment, we always read the first entry of a page table and therefore the same cache set. We see that the penalty of this miss is 11 cycles, also reflected in the value of the PAGE_WALKS:CYCLES event counter, which changes from 5 to 16. Later experiments will show that an L1 data cache miss penalty for data load on this platform is indeed 11 cycles, which means that the L1 data cache miss penalty simply adds up with the DTLB miss penalty.

As we increase the stride, we start to trigger misses in the PDE cache. With the stride of 8192 pages, which spans 16 PDE entries, and 5 or more accessed pages, the PDE cache misses on each access. The L1 data cache misses event counter indicates that there are three L1 data cache references per memory access, two of them are therefore caused by the page walk. This means that a PDP cache is also present and the PDE miss penalty is 4 cycles.

Further increasing the stride results in a gradual increase of the PDP cache misses. With the 512×512 pages stride, each access maps to a different PDP entry. At 5 accessed pages, the L1D_ALL_REF event counter increases to 5 L1 data cache references per access. This indicates that there is no PML4TE cache, since all four levels of the paging structures are traversed, and that the PDP cache has at most 4 entries. Compared to the 8192 pages stride, the PDP miss adds approximately 19 cycles per access. Out of those, 11 cycles are added by an extra L1 data cache miss, as both PDE and PTE entries miss the L1 data cache due to being mapped to the same set. The remaining 8 cycles is the cost of walking two additional levels of page tables due to the PDPTE miss.

The standard deviation of the results exceeds the limit of 0.5 cycles only when the L1 cache associativity is about to be exceeded – up to 3.5 cycles, and when the translation cache level is about to be exhausted – up to 8 cycles.

The observed access durations and the corresponding change in the data cache access count from an analogous experiment on Platform AMD Server are shown in Fig. 4. We can see that for a stride of 128 pages, we still hit the PDE cache as in the previous experiment. Strides of 512 pages and more need 2 page walk steps and thus hit the PDPTE cache. Strides of 256 K pages need 3 steps and thus hit the PML4TE cache. Finally, strides of 128 M pages need all 4 steps. The access duration increases by 21 cycles for each additional page walk step. With a 128 M stride, we see an additional penalty due to page walks triggering L2 cache misses.

The standard deviation of the results exceeds the limit of 0.5 cycles only when the L2 cache capacity is exceeded – up to 18 cycles, and when the translation cache level is about to be exhausted – up to 10 cycles.

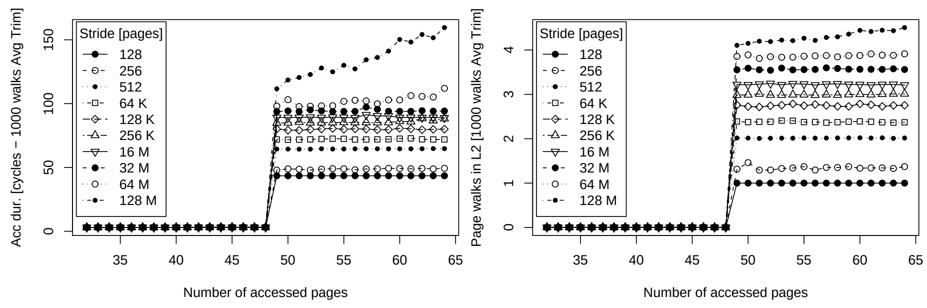

Fig. 4. Extra translation caches miss penalty (left) and related page walk requests to L2 cache (right) on AMD Server.

3 Investigating Memory Caches

On Platform Intel Server, the memory caches include an L1 instruction cache per core, an L1 data cache per core, and a shared L2 unified cache per every two cores. Both L1 caches are virtually indexed, the L2 cache is physically indexed. On Platform AMD Server, the memory caches include an L1 instruction cache per core, an L1 data cache per core, an L2 unified cache per core, and a shared L3 unified cache per every four cores. The following table summarizes the basic parameters of the memory caches on the two platforms, with the parameters not available in vendor documentation *emphasized*.

Table 2. Cache parameters

Cache	Size	Associativity	Index	Miss [cycles]
Platform Intel Server				
L1 data	32 KB	8-way	virtual	11
L1 code	32 KB	8-way	virtual	*30*[2]
L2 unified	4 MB	16-way	physical	*256-286*[3]
Platform AMD Server				
L1 data	64 KB	2-way	*virtual*	*12 random, 27-40 single set*[4]
L1 code	64 KB	2-way	*virtual*	*20 random*[5], *25 single set*[6]
L2 unified	512 KB	16-way	*physical*	*+32-35 random*[7], *+16-63 single set*
L3 unified	2 MB	32-way	*physical*	*+208 random*[8], *+159-211 single set.*

[2] Includes penalty of branch misprediction.
[3] Depends on the cache line set where misses occur. Also includes associated DTLB1 miss and L1 data cache miss due to page walk.
[4] Differs from the 9 cycles penalty stated in vendor documentation [9, page 223].
[5] Includes partial penalty of branch misprediction and L1 ITLB miss.
[6] Includes partial penalty of branch misprediction.
[7] Depends on the offset of the word accessed. Also includes penalty of L1 DTLB miss.

We begin our memory caches investigation by describing experiments targetted at the cache line sizes, which differ between vendor documentation and reported research.

3.1 Cache Line Size

The experiments we perform are still based on measuring durations of memory accesses using various access patterns in the pointer walk from Listing 1.1. To avoid the effects of hardware prefetching, we use a random access pattern generated by code from Listing 1.4. First, an array of pointers to the buffer of $allocSize$ bytes is created, with a distance of $accessStride$ bytes between two consecutive pointers. Next, the array is shuffled randomly. Finally, the array is used to create the access pattern of a length of $accessSize$ divided by $accessStride$.

Listing 1.4. Random access pattern generator.

```
// Create array of pointers in the allocated buffer
int numPtrs = allocSize / accessStride;
uintptr_t **ptrs = new (uintptr_t *)[numPtrs];
for (int i = 0; i < numPtrs; i++)
   ptrs [i] = buffer + i * accessStride;

// Randomize the order of the pointers
random_shuffle (ptrs, ptrs + numPtrs);

// Create the pointer walk from selected pointers
uintptr_t *start = ptrs [0];
uintptr_t **ptr = (uintptr_t **) start;
int numAccesses = accessSize / accessStride;
for (int i = 1; i < numAccesses; i++) {
   uintptr_t *next = ptrs [i];
   (*ptr) = next;
   ptr = (uintptr_t **) next;
}

// Wrap the pointer walk
(*ptr) = start;
delete [] ptrs;
```

3.2 Experiment: Cache line size

In order to determine the cache line size, the experiment executes a measured workload that randomly accesses half of the cache lines, interleaved with an interfering workload that randomly accesses all the cache lines. For data caches, both

[8] Includes penalty of L2 DTLB miss.

workloads use a pointer emitting version of code from Listing 1.4 to initialize the access pattern and code from Listing 1.1 to traverse the pattern. For instruction caches, both workloads use a jump emitting version of code from Listing 1.4 to initialize the access pattern and code from Listing 1.2 to traverse the pattern. The measured workload uses the smallest possible access stride, which is 8 B for 64 bit aligned pointer variables and 16 B for jump instructions. The interfering workload varies its access stride. When the stride exceeds the cache line size, the interfering workload should no longer access all cache lines, which should be observed as a decrease in the measured workload duration, compared to the situation when the interfering workload accesses all cache lines.

The results from both platforms and all cache levels and types, except the L2 cache on Platform Intel Server, show a decrease in the access duration when the access stride of the interfering workload increases from 64 B to 128 B. The counts of the related cache miss events confirm that the decrease in access duration is caused by the decrease in cache misses. Except for the L2 cache on Platform Intel Server, we can therefore conclude that the line size is 64 B for all cache levels, as stated in the vendor documentation.

Figure 5 shows the results for the L2 cache on Platform Intel Server. These results are peculiar in that they would indicate the cache line size of the L2 cache is 128 B rather than 64 B, a result that was already reported in [10]. The reason behind the observed results is the behavior of the streamer prefetcher [11, page 3-73], which causes the interfering workload to fetch two adjacent lines to the L2 cache on every miss, even though the second line is never accessed. The interfering workload with a 128 B stride thus evicts two 64 B cache lines. Figure 5 contains values of the L2 prefetch miss (L2_LINES_IN:PREFETCH) event counter collected from the interfering workload rather than the measured workload, and confirms that L2 cache misses triggered by prefetches occur.

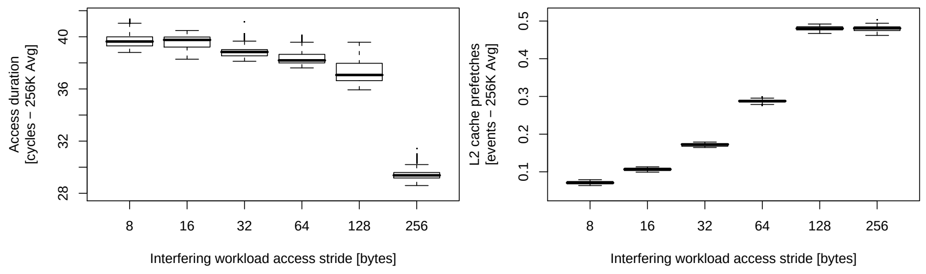

Fig. 5. The effect of interfering workload access stride on the L2 cache eviction (left); streamer prefetches triggered by the interfering workload during the L2 cache eviction on Intel Server (right).

Because the vendor documentation does not explain the exact behavior of the streamer prefetcher when fetching two adjacent lines, we have performed a slightly modified experiment to determine which two lines are fetched together.

Both workloads of the experiment access 4 MB with 256 B stride, the measured workload with offset 0 B, the interfering workload with offsets 0, 64, 128 and 192 B. The offset therefore determines whether both workloads access the same cache associativity sets or not. The offset of 0 B should always evict lines accessed by the measured code, the offset of 128 B should always avoid them. If the streamer prefetcher fetches a 128 B aligned pair of cache lines, using the 64 B offset should also evict the lines of the measured workload, while the 192 B offset should avoid them. If the streamer prefetcher fetches any pair of consecutive cache lines, using both the 64 B offset and the 192 B offset should avoid the lines of the measured workload.

The results on Fig. 6 indicate that the streamer prefetcher always fetches 128 B aligned pair of cache lines, rather than any pair of consecutive cache lines.

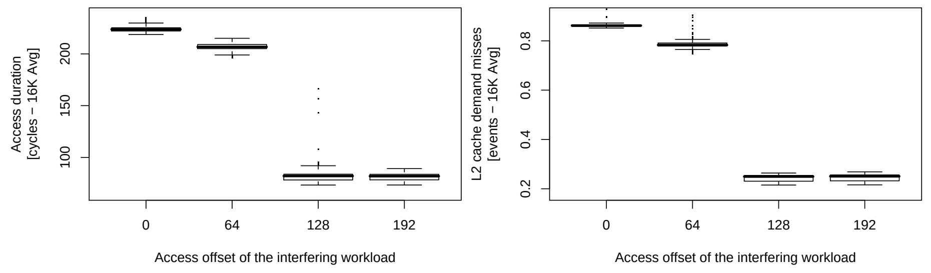

Fig. 6. Access duration (left) and L2 cache misses by accesses only (right) investigating streamer prefetch on Intel Server.

Additional experiments also show that the streamer prefetcher does not prefetch the second line of a pair when the L2 cache is saturated with another workload. Running two workloads on cores that share the cache therefore results in fewer prefetches than running the same two workloads on cores that do not share the cache.

3.3 Cache Indexing

We continue by determining whether the cache is virtually or physically indexed, since this information is also not always available in vendor documentation. Knowing whether the cache is virtually or physically indexed is essential for later experiments that determine cache miss penalties.

We again use the pointer walk code from Listing 1.1 and create the access pattern so that all accesses map to the same cache line set. To achieve this, we reuse the pointer walk initialization code from the TLB experiments on Listing 1.3, because the stride we need is always a multiple of the page size on our platforms. The difference is in that we do not use the offset randomization.

For physically indexed caches, the task of constructing the access pattern where all accesses map to the same cache line set is complicated by the fact that the cache line set is determined by physical rather than virtual address. To overcome this complication, our framework provides an allocation function that returns pages whose physical and virtual addresses are identical in the bits that determine the cache line set. This allocation function, further called colored allocation, is used in all experiments that define strides in physically indexed caches.

Note that we do not have to determine cache indexing for the L1 caches on Platform Intel Server, where the combination of 32 KB size and 8-way associativity means that an offset within a page entirely determines the cache line set.

3.4 Experiment: Cache set indexing

We measure the average access time in a set collision pointer walk from Listing 1.1 and 1.3, with the buffer allocated using either the standard allocation or the colored allocation. The number of accessed pages is selected to exceed the cache associativity. If a particular cache is virtually indexed, the results should show an increase in access duration when the number of accesses exceeds associativity for both modes of allocation. If the cache is physically indexed, there should be no increase in access duration with the standard allocation, because the stride in virtual addresses does not imply the same stride in physical addresses.

The results from Platform Intel Server show that colored allocation is needed to trigger L2 cache misses, as illustrated in Fig. 7. The L2 cache is therefore physically indexed. Without colored allocation, the standard deviation of the results grows when the L1 cache misses start occuring, staying below 3.2 cycles for 8 accessed pages and below 1 cycle for 9 and more accessed pages. Similarly with colored allocation, the standard deviation stays below 5.5 cycles for 7 and 8 accessed pages when the L1 cache starts missing, and below 10.5 cycles for 16 and 17 accessed pages when the L2 cache stats missing.

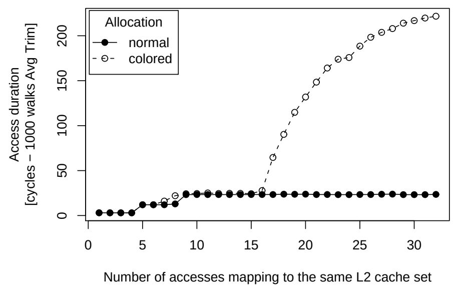

Fig. 7. Dependency of associativity misses in L2 cache on page coloring on Intel Server.

The results from Platform AMD Server on Fig. 8 also show that colored allocation is needed to trigger L2 cache misses with 19 and more accesses. Colored allocation also seems to make a difference for the L1 data cache, but values of the event counters on Fig. 8 show that the L1 data cache misses occur with both modes of allocation, the difference in the observed duration therefore should not be attributed to indexing. The standard deviation of the results exceeds the limit of 0.5 cycles for small numbers of accesses, with a maximum standard deviation of 2.1 cycles at 3 accesses.

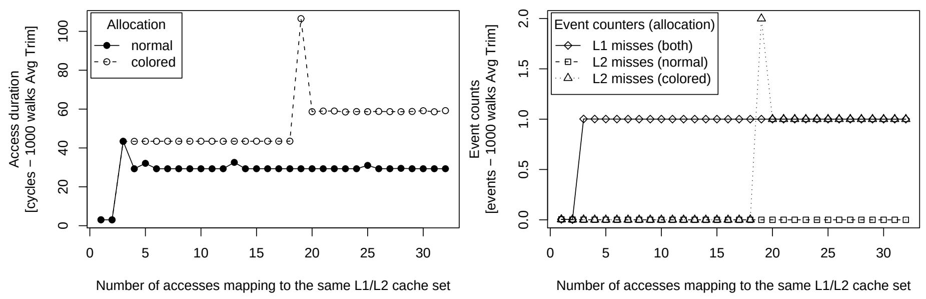

Fig. 8. Dependency of associativity misses in L1 data and L2 cache on page coloring (left) and related performance events (right) on AMD Server.

3.5 Cache Miss Penalties

Finally, we measure the memory cache miss penalties, which appear to include effects not described in vendor documentation.

3.6 Experiment: Cache miss penalties and their dependencies

The experiment determines the penalties of misses in all levels of the cache hierarchy and their possible dependency on the offset of accesses triggering the misses. We rely again on the set collision access pattern from Listing 1.1 and 1.3, increasing the number of repeatedly accessed addresses and varying the offset within a cache line to determine its influence on the access duration. The results are summarized in Table 2, more can be found in [7].

On Platform Intel Server, we observe an unexpected increase in the average access duration when about 80 different addresses mapped to the same cache line set. The increase, visible on Fig. 9, is not reflected by any of the relevant event counters. Further experiments, also illustrated on Fig. 9, reveal a difference between accessing odd and even cache line sets within a page. We see that the difference varies with the number of accessed addresses, with accesses to the even cache lines faster than odd cache lines for 32 and 64 addresses, and the other way around for 128 addresses. The standard deviation in these results is under 3 clocks, or 1 % of the values.

On Platform AMD Server, we observe an unusually high penalty for the L1 data cache miss, with an even higher peak when the number of accessed addresses

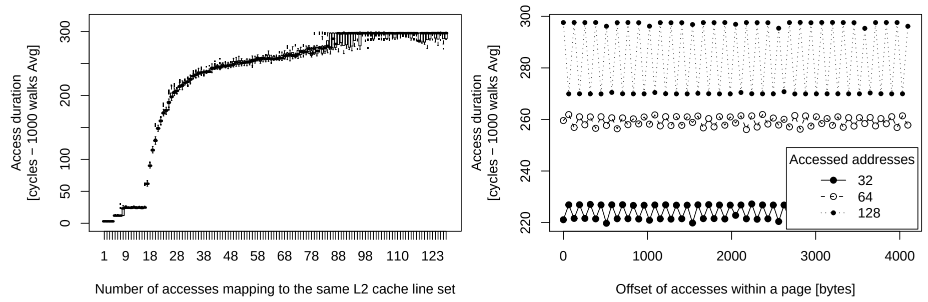

Fig. 9. L2 cache miss penalty when accessing single cache line set (left); dependency on cache line set selection in pages of color 0 (right) on Intel Server.

just exceeds the associativity, as illustrated in Fig. 10. Determined this way, the penalty would be 27 cycles, 40 cycles for the peak, which is significantly more than the stated L2 access latency of 9 cycles [9, page 223]. Without additional experiments, we speculate that the peak is caused by the workload attempting to access data that is still in transit from the L1 data cache to the L2 cache.

More light is shed on the unusually high penalty by another experiment, one which uses the random access pattern from Listing 1.4 rather than the set collision pattern from Listing 1.3. The workload allocates memory range twice the cache size and varies the portion that is actually accessed. Accessing the full range triggers cache misses on each access, the misses are randomly distributed to all cache sets. With this approach, we observe a penalty of approximately 12 cycles per miss, as illustrated on Fig. 10. We have extended this experiment to cover all caches on Platform AMD Server, the differences in penalties when accessing a single cache line set and when accessing multiple cache line sets is summarized in Table 2.

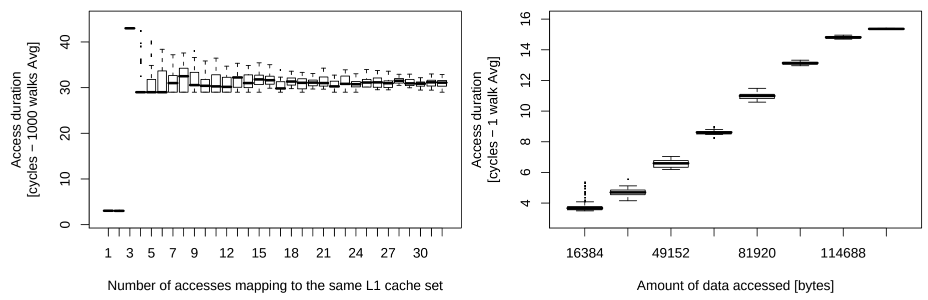

Fig. 10. L1 data cache miss penalty when accessing a single cache line set (left) and random sets (right) on AMD Server.

For the L2 cache, we have also observed a small dependency of the access duration on the access offset within the cache line when accessing random cache sets, as illustrated on Fig. 11. The access duration increases with each 16 B of the offset and can add almost 3 cycles to the L2 miss penalty. A similar dependency was also observed when accessing multiple addresses mapped to the the same cache line set, as illustrated on Fig. 11.

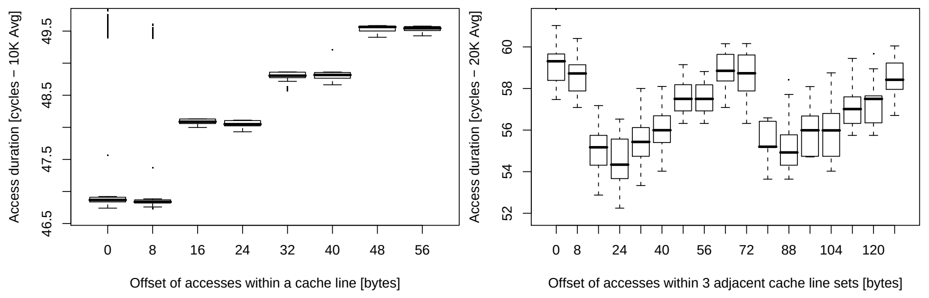

Fig. 11. Dependency of L2 cache miss penalty on access offset in a cache line when accessing random cache line sets (left) and 20 cache lines in the same set (right) on AMD Server.

Again, we believe that illustrating the many variables that determine the cache miss penalties is preferable to the incomplete information available in vendor documentation, especially when results of more complex experiments which include such effects are to be analyzed.

4 Experimental Framework

The experiments described here were performed within a generic benchmarking framework, designed to investigate performance related effects due to sharing of resources such as the processor core or the memory architecture among multiple software components. The framework source is available for download at http://dsrg.mff.cuni.cz/benchmark together with multiple benchmarks, including all the benchmarks described in this paper, implemented in the form of extensible workload modules. The support provided by the framework includes:

- Creating and executing parametrized benchmarks. The user can specify ranges of individual parameters, the framework executes the benchmark with all the specified combinations of the parameter values.
- Collecting precise timing information through RDTSC instruction and performance counter values through PAPI [4].
- Executing either isolated benchmarks or combinations of benchmarks to investigate the sharing effects.
- Plotting of results through R [12]. Supports boxplots for examining dependency on one benchmark parameter and plots with multiple lines for different values of other benchmark parameters.

Besides providing the execution environment for the benchmarks, the framework bundles utility functions, such as the colored allocation used in experiments with physically indexed caches in Section 3.

The colored allocation is based on page coloring [13], where the bits determining the associativity set are the same in virtual and physical address. The number of the associativity set is called a color. As an example, the L2 cache on Platform Intel Server has a size of 4 MB and 16-way associativity, which means that addresses with a stride of 256 KB will be mapped to the same cache line set [11, page 3-61]. With 4 KB page size, this yields 64 different colors, determined by the 6 least significant bits of the page address.

Although the operating system on our experimental platforms does not support page allocation with coloring, it does provide a way for the executed program to determine its current mapping. Our colored allocation uses this information together with the *mremap* function to allocate a continuous virtual memory area, determine its mapping and remap the allocated pages one by one to a different virtual memory area with the target virtual addresses matching the color of the physical addresses. This way, the allocator can construct a continuous virtual memory area with virtual pages having the same color as the physical frames that the pages are mapped to.

5 Conclusion

We have described a series of experiments designed to investigate some of the detailed parameters of the memory architecture of the x86 processor family. Although the knowledge of the detailed parameters is of limited practical use in general software development, where it is simply too involved and too specialized, we believe it is of significant importance in designing and evaluating research experiments that exercise the memory architecture. Without this knowledge, it is difficult to design experiments that target the intended part of the memory architecture and to distinguish results that are characteristic of the experiment workload from results that are due to incidental interference. We should point out that the detailed parameters are often not available in vendor documentation, or – since claiming to know all vendor documentation would be somewhat preposterous – at least are often only available as fragmented information buried among hundreds of pages of text.

Among the detailed parameters investigated in this paper are the address translation miss penalties (which are partially documented for Platform Intel Server and not documented for Platform AMD Server), the parameters of the additional translation caches (which are not documented for Platform Intel Server and not even mentioned for Platform AMD Server), the cache line size (which is well documented but measured incorrectly in [10]) together with the reasons for the cited incorrect measurement, the cache indexing (which seems to be generally known but is not documented for Platform AMD Server), and the cache miss penalties (which seem to be more complex than documented even when abstracting from the memory itself). Additionally, we show some interesting anomalies such as suspect values of performance counters.

We also provide a framework that makes it possible to easily reproduce our experiments, or to execute our experiments on different experiment platforms. The framework is used within the Q-ImPrESS project and many more collected results are available in [7].

To our knowledge, the experiments that we have performed are not available elsewhere. Closest to our work are the results in [10] and [14], which describe algorithms for automatic assessment of basic memory architecture parameters, especially the size and associativity of the memory caches. The workloads used in [10] and [14] share common features with some of our workloads, especially where the random pointer walk is concerned. Our workloads are more varied and therefore provide more results, although the comparison is not quite fair since we did not aim for automated analysis. We also show some effects that the cited workloads would not reveal.

Although this paper is primarily targeted at performance evaluation professionals involved in detailed measurements related to the memory architecture of the x86 processor family, our results in [7] demonstrate that the observed effects can impact performance modeling precision at much higher levels.

As far as the general applicability of our results is concerned, it should be noted that they are very much tied to the particular experimental platforms, and can change even with minor platform parameters such as processor or chipset stepping. For different experimental platforms, our results can serve to illustrate what effects can be observed, but not to guarantee what effects will really be present. The availability of our experimental framework, however, makes it possible to repeat our experiments with very little effort, leaving only the evaluation of the different results to be carried out where applicable.

References

1. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer 's Manual, Volume 3: System Programming, Order Nr. 253668-027 and 253669-027. (Jul 2008)
2. Advanced Micro Devices, Inc.: AMD64 Architecture Programmer's Manual Volume 2: System Programming, Publication Number 24593, Revision 3.14. (Sep 2007)
3. Drepper, U.: What every programmer should know about memory. http://people.redhat.com/drepper/cpumemory.pdf (2007)
4. PAPI: Performance application programming interface. http://icl.cs.utk.edu/papi
5. Pettersson, M.: Perfctr. http://user.it.uu.se/ mikpe/linux/perfctr/
6. Advanced Micro Devices, Inc.: AMD BIOS and Kernel Developer's Guide For AMD Family 10h Processors, Publication Number 31116, Revision 3.06. (Mar 2008)
7. Babka, V., Bulej, L., Děcký, M., Kraft, J., Libič, P., Marek, L., Seceleanu, C., Tůma, P.: Resource usage modeling, Q-ImPrESS deliverable 3.3. http://www.q-impress.eu (Sep 2008)
8. Intel Corporation: Intel 64 and IA-32 Architectures Application Note: TLBs, Paging-Structure Caches, and Their Invalidation, Order Nr. 317080-002. (Apr 2008)

9. Advanced Micro Devices, Inc.: AMD Software Optimization Guide for AMD Family 10h Processors, Publication Number 40546, Revision 3.06. (Apr 2008)
10. Yotov, K., Pingali, K., Stodghill, P.: Automatic measurement of memory hierarchy parameters. In: Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, ACM (2005) 181–192
11. Intel Corporation: Intel 64 and IA-32 Architectures Optimization Reference Manual, Order Nr. 248966-016. (Nov 2007)
12. R: The R Project for Statistical Computing. http://www.r-project.org/
13. Kessler, R.E., Hill, M.D.: Page placement algorithms for large real-indexed caches. ACM Trans. Comput. Syst. **10**(4) (1992) 338–359
14. Yotov, K., Jackson, S., Steele, T., Pingali, K., Stodghill, P.: Automatic measurement of instruction cache capacity. In: Languages and Compilers for Parallel Computing (LCPC) 2005. Volume 4339 of LNCS., Springer (2006) 230–243