Performance Analysis of the Memory Management Unit under Scale-out Workloads

Vasileios Karakostas[*†], Osman S. Unsal[†], Mario Nemirovsky[‡], Adrian Cristal[*†§], Michael Swift[¶]
[*] Barcelona Supercomputing Center
[†] Universitat Politecnica de Catalunya
[‡] ICREA Senior Research Professor at Barcelona Supercomputing Center
[§] Spanish National Research Council (IIIA-CSIC)
[¶] University of Wisconsin-Madison
{vasilis.karakostas, osman.unsal, mario.nemirovsky, adrian.cristal}@bsc.es, swift@cs.wisc.edu

***Abstract*—Much attention has been given to the efficient execution of the scale-out applications that dominate in datacenter computing. However, the effects of the hardware support in the Memory Management Unit (MMU) in combination with the distinct characteristics of the scale-out applications have been largely ignored until recently. In this paper, we comprehensively quantify the MMU overhead on a real machine leveraging the use of performance counters on a collection of emerging scale-out applications. We show that the MMU overhead accounts for up to 16% of the total execution time due to the high TLB miss rates and the interference between page walks and application data in the cache hierarchy. We find that decreasing the MMU overhead - with large pages - may improve the application performance by up to 13.9%. However, the limited MMU support for large pages in combination with the workloads' low memory locality may even harm the performance when large pages are enabled. By comparing the expected and measured application speedup, we observe a performance gap of up to 3.8%, indicating that any improvements in the MMU may result in more efficient utilization of the available execution resources. Finally, we find that the MMU overhead remains high for most scale-out applications even in the presence of large pages, leaving ample space for optimizations. In response, we present upper bounds for perfect MMU optimizations that motivate rethinking its design in the context of the scale-out applications.**

I. Introduction

In recent years, companies like Amazon, Google and Facebook have invested resources to build big datacenters where their software infrastructure run on a large number of inexpensive computers. The datacenters aim to provide the most scalable and economical way to leverage the vast amount of available processing power. Given the high cost of building and maintaining datacenters, a single-digit performance improvement in the utilization of datacenters translates directly to savings in money. To this end, datacenter infrastructures have received attention during the last years in improving the performance of all the involved components such as processors, storage, and interconnection networks.

To stimulate the research in the topic of datacenters, the CloudSuite benchmark suite was recently introduced [25]. CloudSuite is a collection of popular scale-out applications that target various domains of datacenter computing including data analytics (MapReduce), key-value caching (MemCached) and storing (NoSQL), large-scale graph analytics (GraphLab), and web-searching (Nutch) among others. The scale-out applications operate on large datasets with low memory locality exhibiting inefficient execution in the traditional server architectures [25]. In response, computer architects proposed novel designs to increase the efficiency of microprocessors for scale-out applications through improvements in the processor pipeline [36], the memory hierarchy [29], and the on-chip interconnection network [35].

However, the overhead of the Memory Management Unit (MMU) in the context of the scale-out applications has been largely ignored. There have been very few studies - all of them recent - on the performance cost of the MMU that proposed solutions to mitigate them through either reducing the number of TLB misses [13] or the cost of page walks [15]. Still, these studies used only a subset of scale-out applications and did not provide an extensive characterization of the MMU behavior in the context of datacenter computing.

Our goal in this paper is to understand how the MMU (i) performs under the execution of the scale-out applications, (ii) affects the application performance, (iii) interacts with other components of the processor, and (iv) can be potentially improved to boost the performance of datacenters. To this end we analyze the performance of the Memory Management Unit under the execution of various scale-out applications. We conduct our analysis leveraging the use of performance counters on an x86_64 real system. To the best of our knowledge, we are the first to undertake such an effort.

The main contributions of this paper are:

- We perform a comprehensive performance analysis of the MMU for several scale-out applications showing that the MMU overhead accounts up to 16% of the total execution time.
- We find that by reducing the MMU overheads, the performance improves by up to 13.9% enabling better exploitation of the available execution resources.
- We observe that large pages are beneficial for most applications without being an "always-win" option due to limited hardware support.
- We quantify the interference between the application data and the page-table structures in the cache hi-

erarchy, and show how page walks are affected by hardware prefetchers.

- We present upper-bound analyses and discuss potential future directions with the hope to help designing new MMUs in light of the characteristics of these widely-used modern scale-out applications.

In Section II we provide background information regarding the MMU and the scale-out applications we use in our study, while in Section III we discuss our methodology. We present the performance analysis of the MMU under the execution of the scale-out workloads in Section IV, and we discuss potentials for improving the MMU performance in Section V. Finally, in Section VI we discuss the related work and in Section VII we conclude our study.

II. Background

In this paper we focus on the performance of the Memory Management Unit (MMU) under the execution of scale-out applications. Here we briefly describe the scale-out applications from CloudSuite [1], [25] that we use in our study. We also provide background information about the hardware support of the MMU - the Translation Lookaside Buffer (TLB) and the MMU cache - of the x86_64 architecture which constitutes the dominant processor architecture deployed in today's datacenters [12].

A. Scale-out Applications

Data-analytics (MapReduce). This benchmark uses Mahout, a scalable machine learning and data mining library designed for the Hadoop MapReduce framework. The benchmark performs the Bayesian classification algorithm for a large input set of Wikipedia articles.

Data-caching (MemCached). MemCached is a distributed memory caching system that speeds up dynamic database-driven websites by caching data in main memory to reduce the number of accesses in the database. The benchmark simulates the behavior of a caching server for Twitter.

Data-serving (NoSQL). This benchmark targets the domain of NoSQL databases which have gained growing industry use in big data and real-time web applications. The benchmark uses Cassandra, a column-oriented database server, and simulates an update-heavy workload.

Graph-analytics (GraphLab). This benchmark relies on GraphLab, an abstraction framework that expresses asynchronous, dynamic, graph-parallel computation. The benchmark is a GraphLab-based implementation of *tunkrank* that measures a person's influence on Twitter.

Media-streaming (QuickTime). This benchmark targets the domain of media-streaming services and uses the Darwin-Streaming Server (open-source equivalent of Apple QuickTime Server) that streams media to clients across the Internet.

Software-testing (Cloud9). This benchmark uses Cloud9, an automated software-testing platform that parallelizes symbolic execution.

Web-search (Nutch). This benchmark targets web search engines that dominate among the internet services [30]. The benchmark uses the distributed version of Nutch, an open source web search engine, with content crawled from *http://en.wikipedia.org/*.

Per-core TLB Hierarchy		
I-TLB	4K	128 entries, 4-way assoc.
	2M	8 entries, fully assoc.
D-TLB	4K	64 entries, 4-way assoc.
	2M	32 entries, 4-way assoc.
L2 TLB	4K	512-entries, 4-way assoc.
	2M	-

TABLE I: TLB hierarchy of the test machine.

B. Memory Management Unit (MMU)

Translation Lookaside Buffer (TLB). Virtual memory effectively virtualizes the physical memory of a computing system by dividing it into blocks and allocating them to different processes [26]. In the x86_64 architecture, the translations from virtual to physical addresses are kept in the Page Table that is stored as a 4-level hierarchical radix tree [28]. To accelerate virtual memory, most processors employ a Translation Lookaside Buffer (TLB) that holds recently used page table entries. The TLB is on the critical path of every memory operation. This requirement has turned the TLB into a crucial component for the performance of the processor [28]. In case of a TLB miss, the MMU walks the page table through a hardware state machine. Thus, a TLB miss, i.e. page walk, requires up to four memory operations to resolve.

MMU cache. Due to the impact of the page walk latency in the performance, commercial processors have employed MMU caches [11], [15]. The MMU cache reduces the cost of page walks by caching intermediate levels of the page table, while the TLB only caches the leaves. A hit in the MMU cache enables the processor to skip one or more levels of the page table. Thus, a page walk requires between one and four memory operations to perform an address translation based on the contents of the MMU cache.

III. Methodology

Here we describe the experimental environment and the methodology we followed to analyze the MMU performance.

A. System Setup

We conduct our study on a 4-core Intel Xeon E3-1230 (Sandy Bridge) running at 3.2GHz equipped with 16GB memory. Each core has a private TLB hierarchy (Table I): a first-level Data-TLB, a first-level Instruction-TLB, and a second-level TLB, i.e. shared between I-TLB and D-TLB. Note that in this paper we focus on the impact of second-level TLB misses and misses to 2M pages, both of which trigger page walks.

The system runs OpenSuse 12.3 with the 3.7.10-1.4 Linux kernel. We used seven out of the eight scale-out applications from CloudSuite [1]; we faced tuning problems with *web-serving*. For all the server-oriented applications, we set up both clients and servers on the same machine pinning each to unique cores through the *taskset* utility and we measured

Equations & hardware performance counters
(%) Cycles spent in page walks (data) = (DTLB_LOAD_MISSES.WALK_DURATION + DTLB_STORE_MISSES.WALK_DURATION) / CPU_CLK_UNHALTED.THREAD_P
Page walks per 1000 instr. (data) = (DTLB_LOAD_MISSES.WALK_COMPLETED + DTLB_STORE_MISSES.WALK_COMPLETED) / (INST_RETIRED.ANY_P / 1000)
Average cycles per page walk (data) = (DTLB_LOAD_MISSES.WALK_DURATION + DTLB_STORE_MISSES.WALK_DURATION) / (DTLB_LOAD_MISSES.WALK_COMPLETED + DTLB_STORE_MISSES.WALK_COMPLETED)
(%) Cycles spent in page walks (instructions) = ITLB_MISSES.WALK_DURATION / CPU_CLK_UNHALTED.THREAD_P
Page walks per 1000 instr. (instructions) = ITLB_MISSES.WALK_COMPLETED * 1000 / INST_RETIRED.ANY_P
Average cycles per page walk (instructions) = ITLB_MISSES.WALK_DURATION / ITLB_MISSES.WALK_COMPLETED
L1 misses = L1D.REPLACEMENT
L2 misses = MEM_LOAD_UOPS_RETIRED.LLC_HIT + MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT + MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM + MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS
LLC misses = MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS

TABLE II: Equations and corresponding hardware performance counters.

only the activity of the server programs. Finally, to access the performance counters (Table II) we use the *perf* utility [4] and we report the average results of three runs.

B. Large Pages

Linux provides two mechanisms for enabling large 2M pages: (i) Transparent Huge Pages (THP) [7] and (ii) libhugetlbfs [2]. THP attempts to allocate large pages to service application's memory requests that are naturally 2MB-aligned in the anticipation of subsequent memory allocations. If no large pages are available, the kernel falls back to 4K pages transparently to the application, and periodically scans through the memory to substitute several 4K pages with a large page. On the other hand, with libhugetlbfs [2], large pages must be set aside at boot time, they are not swappable and the application must map them explicitly. The main difference between the two mechanisms is that THP supports 2M pages only for anonymous pages, while libhugetlbfs supports large pages for memory-mapped files as well.

We use the following methodology to decide which mechanism we should enable in this study. We run the scale-out applications only with THP enabled and we periodically collect memory statistics from the *proc* filesystem [6] regarding (i) the active working set, (ii) the percentage of the allocated pages that are anonymous, and (iii) the percentage of the allocated large pages over the total working set. Table III summarizes the results.

We observe that most scale-out applications use anonymous pages for more than 96% of their working set. The exception is

Benchmark	**Data Set**	**Anonymous Total Pages %**	**Anonymous Large Pages %**
Data-analytics	5 GB	99.84	72.53
Data-caching	8 GB	99.91	99.89
Data-serving	7 GB	46.42	37.61
Graph-analytics	12 GB	99.89	62.69
Media-streaming	700 MB	99.82	-
Software-testing	700 MB	97.11	25.67
Web-search	6 GB	99.37	75.07

TABLE III: Memory usage statistics. The first column shows the size of the working set, the second column indicates the percentage of allocated anonymous pages over the working set, and the third column shows the percentage of anonymous pages that were allocated as large pages with THP.

data-serving whose working set is mainly divided among the java heap that uses anonymous pages (46%) and the NoSQL database that is memory-mapped. Regarding the ability of THP to successfully allocate large pages, we find that large pages cover: (i) more than 70% for *data-analytics* (72%), *data-caching* (99%) and *web-search* (75%), (ii) more than 62% for *graph-analytics*, (iii) only the java heap (37%) for *data-serving*, (iv) 25% for *software-testing* and surprisingly 0% for *media-streaming*. These results indicate that THP are able to provide large pages for most of the scale-out applications. Finally, to get more confidence about our execution environment we run the applications with libhugetlbfs. We find that the performance is similar among the two configurations for all applications, including *data-serving* and *media-streaming*, after spending significant effort in tuning libhugetlbfs for the needs of each application. Thus, we decide to use large 2M pages through Transparent Huge Pages.

Discussion. Our machine does not have TLB support for 1G pages. Consequently we limit our evaluation of varying the page-size to 4K and 2M.

IV. MMU Performance Analysis

In this paper we analyze the performance of the Memory Management Unit (MMU) under the execution of scale-out applications. We mainly focus on the data accesses that typically stress the MMU more than the instruction accesses [17], [32]. We measure the overhead due to page walks and its impact on the application's performance, we quantify how often a page walk occurs in terms of TLB misses per 1000 instructions (MPKI), and we report the average cost of a page walk. Moreover, we evaluate the interference between the application data and the page walks in the cache hierarchy, we show how the cache hardware prefetchers affect the MMU performance and we discuss the performance of the MMU for instruction accesses. Finally, we summarize the key findings and their implications in the MMU performance.

A. How much time is spent in TLB misses?

The MMU overhead is dictated by the time spent in page walks, i.e. TLB misses. Figure 1 shows the percentage of the execution time spent in page walks due to data accesses with 4K pages (left bar). We make the following observations.

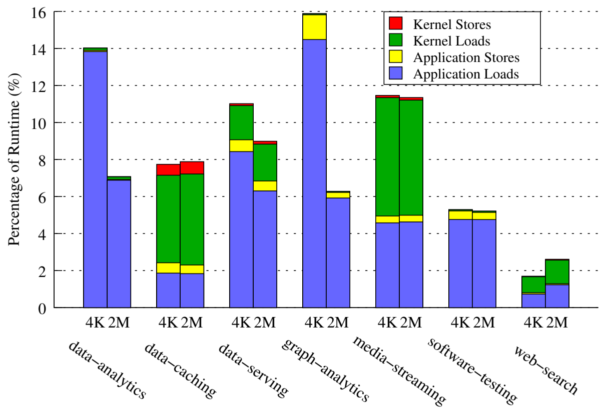


Fig. 1: Percentage of execution time spent in page walks due to data accesses with 4K and 2M pages. The MMU overhead accounts up to 16% of the total execution time.

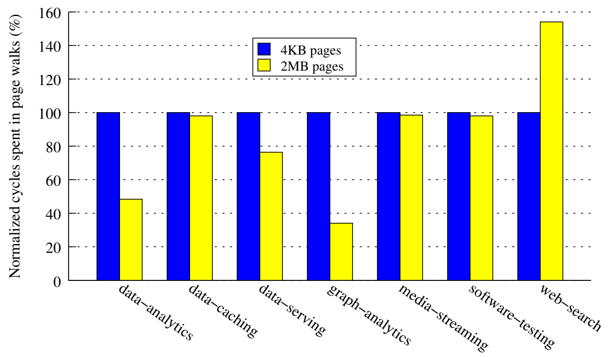


Fig. 2: Normalized cycles spent in page walks due to data accesses with 4K pages and 2M pages.

First, we find that all applications suffer from high MMU overheads with 4K pages. More specifically, *data-serving* and *media-serving* spend more than 10% of the execution time in page walks, while *data-analytics* and *graph-analytics* reach almost 14% and 16%, respectively. These applications operate on big-data (Table III) with low locality [25] stressing the performance of the MMU. To confirm this behavior, we also calculate the number of cold TLB misses based on the working set and the page-size. We find that the cold TLB misses contribute less than 0.02% to the total TLB misses for all the scale-out applications. These results indicate that the MMU overhead is practically dictated by capacity and conflict TLB misses due to the limited MMU resources and the low memory locality of the workloads, rather than by cold TLB misses.

Second, we find that the page walks due to kernel code contribute significantly to the total MMU overhead for *data-caching*, *data-serving*, *media-streaming* and *web-search*. The reason is that these applications stress the network and the file-system stack [25], [34]. Indeed we find that *data-caching*, *data-serving*, *media-streaming* and *web-search* spend 68.6%, 25.5%, 67.2% and 10.7% of the total execution time in kernel code respectively.

We analyze the kernel page walks and categorize them according to the execution path. More than 85% of the kernel page walks take place in functions due to both file-system and network activity for *data-caching*, *media-streaming* and *web-search*, while 44.3% accounts to both file-system and scheduler/synchronization activities for *data-serving*. Moreover, we identify two hotspot functions responsible for page walks: 7.1% for *data-caching* and 16.9% for *media-streaming* of *total* page walks occur only in the kernel function `tcp_poll()` due to network activity, and 5.7% for *data-caching*, 14.3% for *media-streaming* and 9.1% for *web-search* only in `fget_light()` due to both network and file-system activity.

Findings

- The MMU overhead for the scale-out applications is significantly high up to 16%.
- The kernel page walks may contribute more than 50% of the total MMU overhead mainly due to network and file-system activity.

B. *Do Large Pages help?*

To observe the impact of the page-size in the performance of the MMU, we leverage the Transparent Huge Pages (THP) that enable 2M pages. Depending on the application behavior, large pages may decrease the time spent in page walks due to: (i) reducing the number of TLB misses by increasing the TLB reach (Section IV-D), and/or (ii) reducing the average cost of page walk by requiring one memory reference less (Section IV-E). Figure 2 shows the cycles spent in page walks due to the employment of 2M pages normalized to the page walk cycles with 4K pages.

First, we notice that increasing the page-size reduces mostly the MMU overhead by up to 65.9% for *data-analytics*, *data-serving* and *graph-analytics*. *Data-serving* gets the least benefits from 2M pages among these three applications (23.6%). The reason is that *data-serving* accesses the NoSQL database through memory-mapped files. However, THP lack support for memory-mapped files as discussed in Section III-A. Consequently, the THP mechanism covers with 2M pages less part of the working set (only the java heap) for *data-serving* than for *data-analytics* and *graph-analytics* (Table III).

Second, we notice that large pages reduce slightly the time spent in page walks by less than 2% for *data-caching*. We find that the number of page walks cycles due to user code with large pages decreases by 31% on one hand. On the other hand, the number of the dominant page walks cycles due to kernel code increases by 19%. The reason for this behavior is the combination of the application's poor locality with the increased pressure on the TLB for 2M pages: (i) the TLB supports only 32 entries for 2M pages, (ii) the kernel typically uses 2M pages for its internal structures [7], [24], (iii) with THP enabled there is contention between user-level and kernel-level TLB entries in the limited-sized TLB for 2M pages, since x86_64 architecture does not lock TLB entries for kernel usage. Consequently, the time spent in page walks for the application data decreases at the cost of increasing the kernel overheads.

Third, we notice that for *web-search*, increasing the page-size actually increases the time spent in page walks by 54%.

Similarly to *data-caching*, the reason for this behavior is the limited TLB support for 2M pages in combination with the low data locality of the application.

Fourth, 2M pages bring negligible benefits for *media-streaming* and *software-testing*. Surprisingly we found THP fail to allocate any 2M pages for *media-streaming*. The reason is that this scale-out application performs a small number of memory allocations (`mmap()`) with such arguments (size, flags and access rights) that do not allow the THP mechanism to allocate large pages. Finally, the reduction of page walk cycles for *software-testing* is limited by the ability of THP to back only 25.7% of the application's working set with 2M pages.

Figure 1 shows the percentage of execution time spent in TLB misses with 2M pages (right bar). This percentage is now computed based on the total execution time with 2M pages (we discuss the actual performance differences in Section IV-C). We observe that an important fraction of time - more than 6% - is still spent in page walks even for those applications that take benefit from 2M pages (e.g. *data-analytics* and *graph-analytics*). For the rest of the applications the percentage remains practically the same. These results indicate that in case the application performance depends directly on any improvements in the MMU performance, there is still ample space for optimizing the MMU.

Findings

- Large pages reduce the MMU overhead for some scale-out applications by up to 65.9%. However, the limited hardware support for large pages may actually increase the MMU overhead by up to 54%.
- Large pages put more pressure on the TLB for those applications that suffer from a high number of kernel page walks due to the limited support for 2M.
- The software implementation of some scale-out applications cannot use large pages.
- Even if large pages benefit the application performance, still a significant percentage of time spent in page walks leaving space for optimizations.

C. Do TLB misses affect performance?

In this section we quantify the application speedup due to improving the MMU performance by changing the page-size from 4K to 2M. To assess the importance of the MMU performance in the processor pipeline, we compute also the *expected performance* with 2M pages based on Equation 1 [13]. The expected performance with 2M pages is computed as the measured number of execution cycles with 4K pages reduced by the measured improvement in cycles spent in page walks when increasing the page-size from 4K to 2M. In other words, the expected performance assumes that the page walks do not affect at all (neither positively nor negatively) the processor pipeline (out-of-order execution, memory hierarchy, etc.).

$$ExpectedTotalCycles_{2M} = TotalCycles_{4K} - TlbCycles_{4K} + TlbCycles_{2M} \quad (1)$$

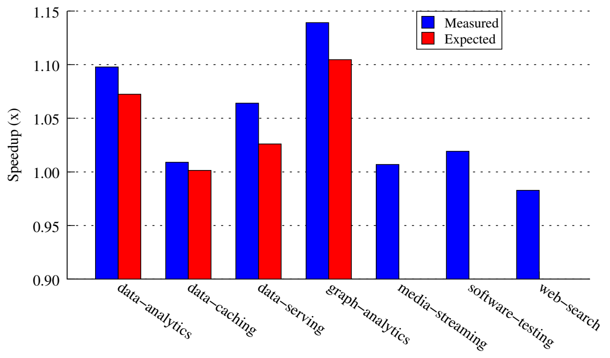


Fig. 3: Speedup due to increasing the page-size from 4K to 2M. The left bar corresponds to the actual measurements, while the right bar corresponds to the expected speedup based on Equation 1. The difference indicates the positive impact of reducing the MMU overhead in the processor pipeline. Note that we do not report expected speedup for those applications that use throughput as performance metric, i.e. *media-streaming*, *software-testing* and *web-search*.

Figure 3 shows the measured speedup that is achieved due to employing large pages on the left bar, and the expected speedup based on Equation 1 on the right bar.

Regarding the measured speedup we see that the performance increases for those applications that reduce the time spent in page walks (Figure 2). More specifically, 2M pages boost the performance of *data-analytics*, *data-serving* and *graph-analytics* by 9.7%, 6.4% and 13.9% respectively. The rest of the applications achieve negligible speedup, while the performance of *web-search* slightly drops by 2% as expected due to spending more cycles in page walks.

Regarding the expected speedup we notice a positive gap between the expected speedup and the measured one, i.e. 9.7% vs. 7.2% for *data-analytics*, 6.4% vs. 2.6% for *data-serving* and 13.9% vs. 10.5% for *graph-analytics*. We believe that the difference in expected and measured performance is due to the impact of page walks on the out-of-order execution and the memory hierarchy. The page walks occur less frequently and need less cycles to complete with large pages, allowing better utilization of the available pipeline resources. To verify this behavior, we measure also the number of stalled cycles in the back-end of the processor pipeline. We find that by using 2M pages, the number of back-end stalled cycles reduces by 16.7% for *data-analytics*, by 10.7% for *data-serving* and by 10.1% for *graph-analytics*, and we also find that the IPC increases by 12.3% for *data-analytics*, by 8.1% for *data-serving* and by 7.4% for *graph-analytics*. We also observe fewer cache misses with 2M pages as we will explain next in Section IV-G respectively. Our results suggest that future improvements in MMU performance will bring more-than-expected application speedup by exploiting the available execution resources better.

Findings

- Improved MMU performance speeds up the scale-out applications by up to 13.9%. However, the limited hardware support for 2M pages may reduce the application performance by 2%.

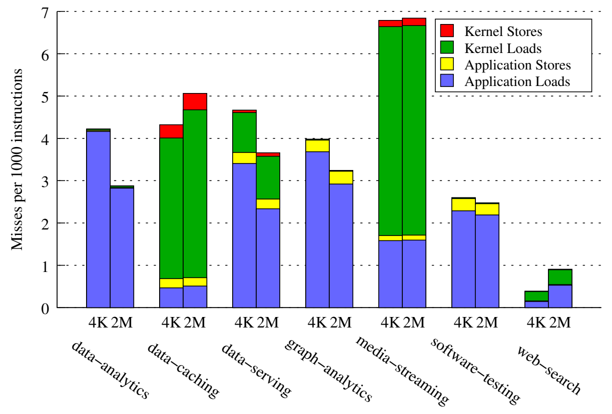


Fig. 4: Page walks (i.e. TLB misses) per 1000 instructions (MPKI) due to data accesses with 4K and 2M pages.

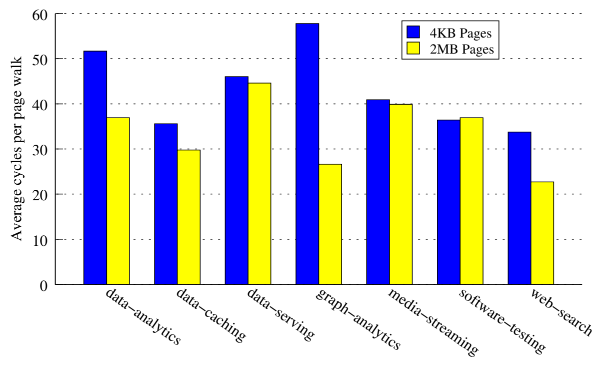


Fig. 5: Average number of cycles per page walk due to data accesses with 4K and 2M pages.

- The difference between the expected and the measured application speedup indicates that optimizations in the MMU will result in more efficient utilization of the execution pipeline.

D. How often do TLB misses occur?

Figure 4 shows the number of page walks (i.e. TLB misses) per thousand instructions (MPKI) due to data accesses when using 4K and 2M pages. By changing the page-size from 4K to 2M, the MPKI reduces for those applications that the page walk overhead decreases (e.g. *data-analytics*, *data-serving* and *graph-analytics*), but not in the same ratio as in Figure 1 since 2M pages reduce also the cost per TLB miss as we will show in the following subsection. However, we notice that the MPKI actually increases for *data-caching* and *web-search* with 2M pages. This behavior confirms the limited hardware support for 2M in current MMUs.

Findings

- The frequency of page walks decreases for some applications due to large pages. However the limited TLB support for 2M pages may increase the MPKI.

	Results for 2M pages normalized to 4K pages		
Benchmark	**Page walks per 1000 instr.**	**Average cycles per page walk**	**Time spent in page walks**
Data-analytics	0.68	0.71	0.48
Data-caching	1.17	0.83	0.98
Data-serving	0.78	0.97	0.76
Graph-analytics	0.81	0.46	0.34
Media-streaming	1.01	0.98	0.99
Software-testing	0.95	1.01	0.98
Web-search	2.32	0.67	1.54

TABLE IV: Summarized results for page walks per 1000 instructions, average cycles per page walk and cycles spent in page walks with 2M pages normalized to 4K pages (lower is better).

E. What is the cost of a TLB miss?

Figure 5 shows the average cost of a page walk with 4K and 2M pages respectively. We observe that the average cost of page walk with 4K pages is far lower than 100 cycles for all applications. This indicates that resolving a page walk does not require any off-chip memory access on average. Our results corroborate previous studies that focused on different applications and concluded that the page walks references typically hit in the cache hierarchy [11], [37].

The latency of the average cost per page walk depends on (i) the performance of the MMU cache, which dictates how many memory accesses (up to four) are necessary to resolve the page walk, and (ii) the locality of the page table references in the data cache hierarchy (i.e. L1, L2 or LLC). Unfortunately, our experimental machine does not provide any counters for measuring the performance of the MMU cache. However, in Section V-B we perform an upper-bound analysis of perfect MMU caches varying the level of data cache hierarchy where the page table references hit and we draw some conclusions about the locality of the page table references in the cache hierarchy.

By comparing the results for the two page-size configurations, we observe that the average cost per page walk is lower for most scale-out applications with 2M pages. In conjunction with the results of the previous sections - which are summarized in Table IV - we observe that the average cost of a page walk with 2M pages reduces significantly for *data-analytics* and *graph-analytics* as expected. For *data-caching* and *web-search*, the average cost also decreases and compensates the increase in MPKI, while for *data-serving*, *media-streaming* and *software-testing*, which have low use of 2M pages, the average cost remains practically the same.

Findings

- The average TLB miss cost indicates that page walk references typically hit in the data cache hierarchy.

F. Comparison with other benchmark suites

In this section we compare the performance of the MMU across different benchmark suites. Figure 6 shows the percentage of execution time spent in page walks for SPEC 2006 [5], BioBench [8], Parsec [18] and CloudSuite.

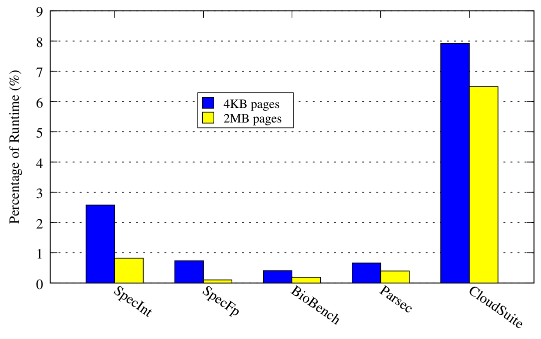


Fig. 6: Comparison of the MMU performance with various benchmark suites (geometric mean per suite).

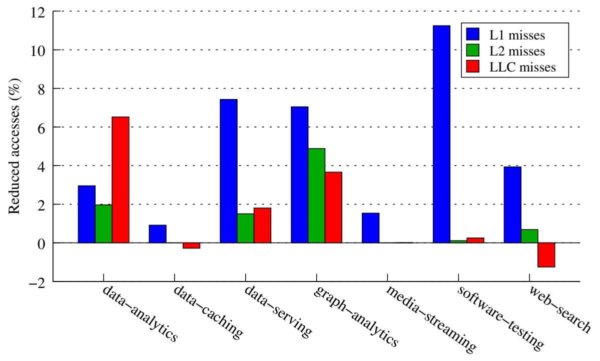


Fig. 7: Percentage of reduced L1, L2 and Last Level Cache (LLC) misses due to increasing the page-size from 4KB to 2MB. The results show that by improving the MMU performance, less interference occurs in the cache hierarchy between the application data and the page table.

We observe that the scale-out applications consistently stress the MMU more compared to other benchmark suites in terms of runtime overhead. Moreover, we find that the scale-out applications suffer almost an order of magnitude more frequently from page walks than other benchmarks; the page walks per 1000 instructions is well below 1 for the majority of Spec, BioBench and Parsec applications even with 4KB pages (35 out of 47 applications). The reason is that these suites consist of several benchmarks with small working sets that fit in the TLB hierarchy. Although there are some benchmarks that cause high MMU overheads (e.g. *mcf*, *omnetpp*, *cactusADM*, *mummer*, *tiger*, *canneal*) they still have smaller working sets, limited kernel activity and enjoy better performance improvement with large pages compared to the scale-out applications. Based on these findings, we corroborate a previous study [25] that pointed out the distinct characteristics of scale-out applications compared to other benchmarks, and we further show that the same observation holds with respect to the MMU behavior.

Findings

- Scale-out applications stress more the MMU performance compared to other compute-intensive and multi-threaded applications.

G. Interference in the cache hierarchy

In this section we quantify the interference in the data-cache hierarchy between the application data and the page table references. To accomplish this, we count the number of L1, L2 and LLC misses for the two page-size configurations. Figure 7 shows the percentage of reduced cache misses due to changing the page-size from 4K to 2M.

We observe that the number of cache misses for most scale-out applications reduces by up to 11.2% for L1 cache (*software-testing*), 4.8% for L2 cache (*graph-analytics*) and 6.5% for LLC (*data-analytics*). This happens due to the improved MMU performance that cause fewer memory accesses due to the page walks and less interference with the application data in the cache hierarchy since: (i) the page table occupies less memory space due to the elimination of one level in the page tree, and (ii) page walks occur less often and are cheaper (Figures 4 and 5). The only exceptions are *data-caching* and *web-search* which suffer more frequently from page walks with 2M than with 4K pages as we showed earlier in Section IV-D, increasing slightly the number of LLC misses.

Moreover, we notice that just reducing the number of L1 misses, as it happens for (*software-testing*), does not affect significantly the performance because they can be hidden by the out-of-order execution. However, we observe a correspondence between performance improvement and reduced data-cache interference - in L2 cache and LLC - for those applications that benefit most from 2M pages (*data-analytics*, *data-serving* and *graph-analytics*).

Findings

- Poor MMU performance can result in increased interference with application data in the cache hierarchy.

H. Interaction with Hardware Prefetchers

Previously we showed the interference between the application data and the page table in the cache hierarchy. However, the page table can be cached up to the L1 cache. Here we quantify this interference due to the activity of the hardware prefetchers from the MMU performance point of view.

Our experimental machine has four prefetching mechanisms; the two of them (DCU and IP stride) are responsible for prefetching data to the L1 cache, while the other two (ACL and Spatial) are responsible for prefetching data to the L2 and the LLC [27]. Table V summarizes three basic metrics of the MMU performance due to data accesses for the scale-out applications with 2M pages for three different prefetcher configurations: (i) all prefetchers are disabled, (ii) only L1 prefetchers (DCU and IP prefetchers) are enabled, and (iii) all prefetchers are enabled. The table shows the total number of page walks, the average number of cycles per page walk, and the total number of cycles spent in page walks, normalized to the case when all prefetchers are disabled. We make the following observations.

We see that the total number of page walks changes for different prefetcher configurations and actually increases for most scale-out applications compared to when all prefetchers are disabled. These results were not expected since the prefetcher

	ON only L1 prefetchers			ON all prefetchers		
Benchmark	**#page walks**	**#average cycles per page walk**	**#cycles spent in page walks**	**#page walks**	**#average cycles per page walk**	**#cycles spent in page walks**
Data-analytics	1.11	1.49	1.65	1.00	1.28	1.29
Data-caching	0.79	1.08	0.86	0.77	1.17	0.90
Data-serving	1.22	1.02	1.25	1.22	0.98	1.20
Graph-analytics	0.98	0.85	0.83	0.96	0.55	0.53
Media-streaming	1.00	1.05	1.05	1.06	1.07	1.14
Software-testing	0.81	1.02	0.83	1.04	1.14	1.19
Web-search	1.62	0.94	1.53	1.66	0.95	1.57

TABLE V: Summarized results for page walks, average cycles per page walk and cycles spent in page walks with 2M pages for various prefetcher configurations normalized to the case when all prefetchers are disabled (lower is better).

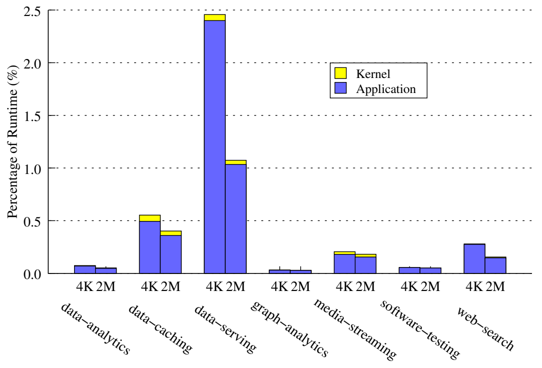


Fig. 8: Page walk overhead due to instruction TLB misses.

requests are supposeed not to trigger page walks [27]. We speculate that the number of page walks differs across configurations because the prefetcher requests either affect the TLB replacement policy (positively or negatively depending on the application), or indeed trigger page walks.

Similarly, we observe that the total number of cycles spent in page walks (and the average cost of page walk respectively) changes across the various configurations. More specifically, the number of page walk cycles is lower when all prefetchers are disabled for most of the scale-out applications. These results indicate that the prefetchers fetch aggressively application data that interfere with the page table in the cache hierarchy. However, we see that the hardware prefetchers reduce by 47% the cycles spent in page walks for *graph-analytics*, even though the number of page walks is reduced by only 4%. Thus, we observe an interaction between the hardware prefetchers and the MMU performance that require further research and documentation.

Findings

- The interference between the application data and the page table in the cache hierarchy due to the activity of the hardware prefetchers suggest that there is potential for reducing the MMU overhead if there is isolation between them in the memory hierarchy.

I. *Instruction TLB misses*

Figure 8 shows the time spent in page walks due to instruction accesses. We see that all scale-out applications, except for *data-serving*, spend a negligible amount of time in page walks due to instruction accesses (less than 0.6%). However, *data-serving* spends 2.45% of the total execution time in page walks due to instruction accesses with 4K pages. This MMU overhead accounts for 25% of the total MMU overhead, and far exceeds typical I-TLB results [17]. The reason lies on the software implementation of *data-serving* which is based on a high-level language (Java) with a managed runtime and extensive use of libraries. When 2M pages are employed, we find that the MPKI reduces by almost 50% due to the lower interference between instruction and data entries in the TLB. However, it is still comparable to the MPKI of TLB misses due to data accesses (0.8 for instruction TLB misses vs. 3.7 for data TLB misses).

Findings

- Instruction references may add non-negligible MMU overheads due to high-level languages and libraries.

J. *Summary & Implications*

We show that the MMU overhead for the scale-out applications is significantly high, up to 16% of the total execution time. As the data-sets for these applications constantly grow, these overheads are expected to increase. Thus, improving the MMU performance should be of paramount importance for the efficient execution of such applications in the big-data era [13], [15]. Moreover, to quantify the correlation between the MMU performance and the application performance, we conducted experiments with large pages. The results show that lower MMU overhead yields up to 13.9% application speed-up. However, even though large pages reduce the MMU overhead, the hardware support for large pages is limited and may actually harm the performance. These findings dictate investing effort in improving the limited hardware support for large pages. Furthermore, the kernel code contributes significantly to the total MMU overhead for most of the scale-out applications mainly due to intense network and file-system activities. This behavior requires further investigation from both operating system and architecture researchers. Finally, the interference between the application data and the poor MMU performance indicates the need for an holistic approach in boosting the performance of the memory hierarchy.

V. Potential Improvements in MMU

Until now we have shown that the MMU overhead accounts for a significant percentage of the total execution time. In this

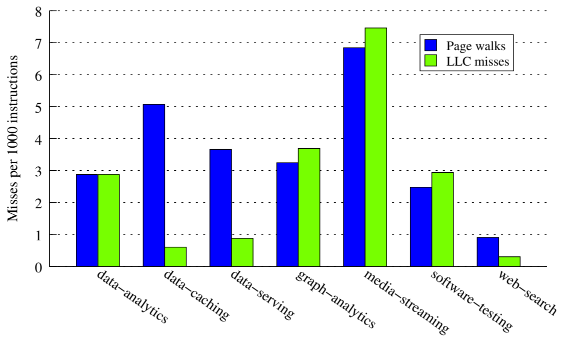


Fig. 9: Comparison of page walks and LLC misses per 1000 instructions due to data accesses with 2M pages.

section we discuss potential solutions and we present upper-bound analyses of performance improvements in the MMU.

A. Virtual Caches

Virtual caches [14], [19], [20], [33], [43], [47] have been proposed as an alternative to reduce the performance and power dissipation overheads of the MMU. Virtual caches use virtual addresses to access the cache hierarchy down to a certain level and only consult the TLB on a cache miss beyond the supported level in the cache hierarchy. Although the virtual caches provide attractive properties, ensuring correct execution requires extra hardware support and complexity (due to synonyms, access rights, etc.).

We want to identify the potential of virtual caches for the scale-out applications regarding performance, assuming a virtual cache design that accesses the MMU on LLC misses. To this end, we compare the MPKI of page walks (left bar) with that of LLC (right bar) with 2M pages in Figure 9. We make the following observations.

First, the results show that four out of seven applications experience similar or higher LLC miss rate compared to the page walks rate. The reason is that these scale-out applications operate on large data-sets suffering from LLC misses that are spread over a big memory space that will likely miss in the TLB as well. Consequently, although virtual caches would help in reducing the power dissipated in the TLB hierarchy, they would provide similar behavior in terms of performance due to the exposed cost of address translation in cache misses. Second, we observe that *data-caching*, *data-serving* and *web-search* suffer more often from page walks than from LLC misses. Such behavior indicates that there is useful data in the LLC that is not covered by the TLBs, thus exposing the inadequate design of current MMUs.

Based on these findings, we conclude that employing virtual caches and removing the MMU from the critical path would bring negligible performance improvement while it would add significant complexity in the implementation of the system.

B. Perfect MMU Caches

The MMU cache helps in reducing the cost of page walks by caching memory references of the upper levels of the page table. In this part of our analysis we evaluate the potential for reducing the time spent in page walks by implementing a perfect MMU cache. This means that the page walk requires only one memory reference that always hits in some level of the cache hierarchy. Bhattacharjee [15] performed a similar analysis to show potential improvements due to perfect MMU caches. In this paper, we go one step further and we quantify also the impact of the hit-level in the cache hierarchy during the page walk for perfect MMU caches. Using microbenchmarks [10], we found that the minimum cost for resolving a page walk that completely hits in the MMU cache and requires only one page table reference that hits in the L1, L2 and LLC cache is 12, 20 and 43 cycles on our platform respectively. Based on these values per page walk and the actually measured number of page walks, we estimate potential performance improvements of perfect MMU caches. Note that we assume pessimistically that the TLB misses have no effect on the rest of the execution pipeline, so that the baseline remains the same for all analyses. However, as we showed earlier in Section IV-C, better MMU performance will bring more-than-expected application speedup by exploiting better the available execution resources.

$$PerfectMMU_{LLC}(\%) = \frac{TLB\_Misses * 43cycles}{Total\_Execution\_Time} \tag{2}$$

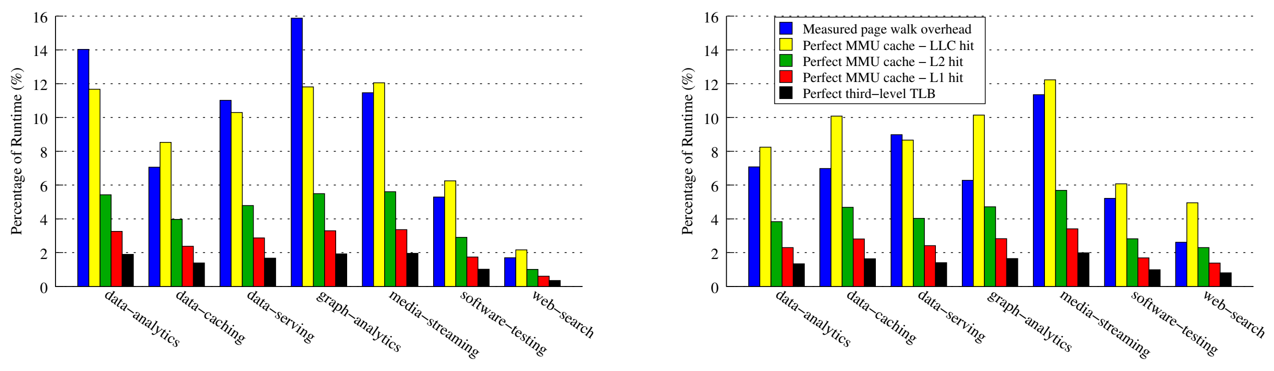
$$PerfectMMU_{L2}(\%) = \frac{TLB\_Misses * 20cycles}{Total\_Execution\_Time} \tag{3}$$

$$PerfectMMU_{L1}(\%) = \frac{TLB\_Misses * 12cycles}{Total\_Execution\_Time} \tag{4}$$

In Figure 10 we plot the percentage of time spent in page walks due to data accesses (i) for the real evaluated hardware (blue bar), (ii) enhanced with perfect MMU caches that require a single memory reference that hits in the LLC (PerfectMMU$_{LLC}$ - Equation 2 - yellow bar), (iii) in the L2 cache (PerfectMMU$_{L2}$ - Equation 3 - green bar), and (iv) in the L1 cache (PerfectMMU$_{L1}$ - Equation 4 - red bar).

We observe that for 4K pages, the actual measured performance overhead is close to that of PerfectMMU$_{LLC}$ or even lower. Since the MMU cache is not perfect for the real measurements, we conclude that the page table references with 4K pages typically hit earlier in the cache hierarchy, well before accessing the LLC. Regarding the configuration with 2M pages, we observe that the measured performance overhead is lower than that of PerfectMMU$_{LLC}$ and close to that of the PerfectMMU$_{L2}$. This implies that the page walks with 2M pages typically hit in L2.

Regarding the potential improvements of the perfect MMU cache itself, that motivated also a recent proposal for improving their performance [15], we notice that the perfect MMU cache brings better performance improvement for the scale-out applications with 4K pages rather than with 2M pages. However, the performance benefits still depend heavily on the level of the cache hierarchy where the page walk hits, indicating the need to keep the page table references as close as possible to the processor.



(a) 4KB Pages

(b) 2MB Pages

Fig. 10: Potential improvements for the MMU overheads.

C. *Perfect Cache Interference*

Overall, the perfect MMU cache provides limited performance benefits for the scale-out applications unless it is incorporated with a mechanism that preserves or promotes the page table references in the cache hierarchy closer to the processor (Figure 10). On the other hand, we showed that interference exists between the application data and the page table in the cache hierarchy, increasing the average cost of page walks and degrading the application performance.

The interference between application data and page table references in the cache hierarchy has been pointed by Wu et al. [48]. However, their study targeted compute-intensive applications that exhibit high cache hit ratio and low TLB miss ratio respectively. Thus, their proposal treated the page table references as polluting cache entries by de-prioritizing them through the cache replacement policy in favor of application's data. We believe that an opposite approach should be considered in the context of scale-out applications that suffer from poor data-cache locality, so that the page walks hit early in the cache hierarchy. Such a research direction is similar in vein with [25] that advocated for preserving instruction references in the cache hierarchy due to the high number of expensive instruction cache misses that take place in the scale-out applications.

D. *Perfect TLBs*

Here we discuss the possibility of employing a *perfect third-level* TLB that always hits (Equation 5 - black bar in Figure 10), i.e. the TLB has unlimited entries or reach, having the same latency (7 cycles) as the actual second-level TLB [27]. The results show that, in this case, the page walk overhead is reduced to less than 2% for all the scale-out applications making the use of virtual memory almost invisible.

$$Perfect_{TLB}(\%) = \frac{TLB\_Misses * 7cycles}{Total\_Execution\_Time} \quad (5)$$

One direction for achieving such performance is through architecting a third-level TLB with a high number of entries. However, such an implementation is likely unfeasible according to the CMOS technology predictions [3] due to leakage power and area overheads. On the other hand, novel memory technologies (e.g. STT-RAMs, Memristors) have been proposed to overcome these CMOS' limitations for other on-chip components such as last-level caches [22], [49]. Leveraging their unique characteristics and designing novel third-level TLBs with such memory technologies should be considered for future research in our opinion.

Another future direction for improving the MMU performance is to design a third-level *range-TLB* that can capture efficiently ranges of pages without constraints on the coverage by a single TLB entry (in contrast to [41], [42], [46]). Direct Segments [13] actually follow this approach, but they provide only a single range. Taking advantage of the fact that a third-level range-TLB is not on the critical path of every memory operation but it is accessed only when misses occur in the higher TLB levels, a more complex design with higher latency and improved range capacity could be beneficial.

Finally, TLB misses could be effectively hidden through smart prefetching. Surprisingly we notice limited proposals in the literature for TLB prefetching [16], [31], [45]. We believe that the high overheads of the MMU for the scale-out applications in combination with their distinct characteristics - low locality and limited sharing among threads [25] - require an effort in optimizing prefetching TLB entries, similarly as inter-core cooperative prefetching [37] leveraged the frequent sharing patterns of the multi-threaded applications to boost the TLB performance.

VI. Related Work

The MMU performance has attracted the interest of both academia and industry for several decades. Early evaluations of the MMU showed its importance in the overall processor performance [9], [21], [23], [40], [44]. However, these studies were conducted under systems with limited physical memory and less sophisticated MMU organization targeting different workloads compared to today's trends in the MMU architectural support and the big-memory scale-out applications respectively.

Jacob and Mudge [28] compared various MMU organizations and showed that the total MMU overhead is roughly twice to what was previously thought due to the interference between the application data and the page table in the cache hierarchy.

Kandiraju et al. [32] presented a detailed characterization of the data TLB behavior for the Spec2000 benchmark suite. The authors suggested that multi-level TLBs are useful in cutting down access times and evaluated different kinds of prefetching. McCurdy et al. [38] evaluated the MMU performance under scientific applications, addressing the limited TLB support for 2M pages and concluded that the false choice of page size can result in performance degradations of up to nearly 50%, while Morari et al. [39] evaluated the TLB miss impact in future HPC systems.

The most recent work in characterizing and analyzing the TLB performance was conducted in the context of Parsec multi-threaded applications [17]. The authors showed that the TLB misses are predictable due to sharing patterns among threads, and that inter-core TLB cooperation and prefetching mechanisms can be applied to improve TLB performance.

Finally, Basu et al. [13] showed that big-data applications stress the MMU performance even with 1G pages. However, their study includes a subset of the applications we use in this paper. Similarly, Bhattacharjee [15] showed that a significant amount of execution time is due to the MMU overhead. However, their study did not focus on scale-out applications.

Our approach. In contrast to previous works, we comprehensively characterize the MMU performance using representative scale-out applications from CloudSuite. We also provide deep insights in the interactions between the MMU and other processor components, and we point out to future directions for improving the performance of the MMU in the context of the emerging memory-intensive scale-out applications.

VII. Conclusions

Understanding the characteristics of the scale-out applications and identifying performance inefficiencies have turned out as fundamental requirements to boost the efficiency of datacenters and to further spread the deployment of the cloud-computing paradigm. With this goal in mind, we comprehensively analyzed the performance execution of the Memory Management Unit under the execution of the scale-out applications. We showed that the MMU overhead accounts up to 16% of the total execution time and we quantified the interference between the application data and the page walks. By reducing the MMU overheads through large pages, we found that the application performance accelerates by up to 13.9% due to better exploitation of the available execution resources. However, the limited hardware support for large pages may expose inefficiencies of the current support for the Virtual Memory. Consequently, based on upper-bound analyses for perfect improvements in the MMU, we suggested future directions for improving the MMU performance with the hope to motivate and help other researchers in designing novel MMU implementations.

VIII. Acknowledgments

We would like to thank Srdjan Stipic, Nehir Sonmez, Ibrahim Hur, Jayneel Gandhi, Gulay Yalcin and all anonymous reviewers for their comments and feedback. This work was partially supported by the cooperation agreement between the Barcelona Supercomputing Center and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIN2007-60625 and TIN2008-02055-E, by the European Network of Excellence on High Performance Embedded Architecture and Compilation (HiPEAC), and by the National Science Foundation under grant CNS-1302260. Vasilis Karakostas is also supported by an FPU-Research grant from the Spanish MEC.

References

[1] “Cloudsuite Overview,” http://parsa.epfl.ch/cloudsuite/overview.html.

[2] “Huge Pages Part 1 (Introduction),” http://lwn.net/Articles/374424/.

[3] “International Technology Roadmap for Semiconductors: 2012,” http://www.itrs.net/Links/2012ITRS/Home2012.htm.

[4] “Perf wiki,” https://perf.wiki.kernel.org/index.php/Main_Page.

[5] “SPEC CPU 2006,” https://www.spec.org/cpu2006/.

[6] “The /proc filesystem,” www.kernel.org/doc/Documentation/filesystems/proc.txt.

[7] “Transparent Huge Pages in 2.6.38,” http://lwn.net/Articles/423584/.

[8] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, “BioBench: A Benchmark Suite of Bioinformatics Applications,” ISPASS, 2005, pp. 2–9.

[9] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska, “The interaction of architecture and operating system design,” ASPLOS, 1991, pp. 108–120.

[10] V. Babka and P. Tuma, “Investigating Cache Parameters of x86 Family Processors,” SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking, 2009, pp. 77–96.

[11] T. W. Barr, A. L. Cox, and S. Rixner, “Translation Caching: Skip, Don’t Walk (the Page Table),” ISCA, 2010, pp. 48–59.

[12] L. A. Barroso, J. Clidaras, and U. Hlzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*, ser. Synthesis Lectures on Computer Architecture, 2013.

[13] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, “Efficient Virtual Memory for Big Memory Servers,” ISCA, 2013, pp. 237–248.

[14] A. Basu, M. D. Hill, and M. M. Swift, “Reducing memory reference energy with opportunistic virtual caching,” ISCA, 2012, pp. 297–308.

[15] A. Bhattacharjee, “Large-reach Memory Management Unit Caches,” MICRO, 2013, pp. 383–394.

[16] A. Bhattacharjee and M. Martonosi, “Inter-core Cooperative TLB for Chip Multiprocessors,” ASPLOS, 2010, pp. 359–370.

[17] A. Bhattacharjee and M. Martonosi, “Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors,” PACT, 2009, pp. 29–40.

[18] C. Bienia, “Benchmarking Modern Multiprocessors,” Ph.D. dissertation, Princeton University, January 2011.

[19] M. Cekleov and M. Dubois, “Virtual-address caches part 1: Problems and solutions in uniprocessors,” *IEEE Micro*, vol. 17, no. 5, pp. 64–71, Sep. 1997.

[20] M. Cekleov and M. Dubois, “Virtual-address caches, part 2: Multiprocessor issues,” *IEEE Micro*, vol. 17, no. 6, pp. 69–74, Nov. 1997.

[21] J. B. Chen, A. Borg, and N. P. Jouppi, “A Simulation Based Study of TLB Performance,” ISCA, 1992, pp. 114–123.

[22] Y.-T. Chen, J. Cong, H. Huang, B. Liu, C. Liu, M. Potkonjak, and G. Reinman, “Dynamically Reconfigurable Hybrid Cache: An Energy-efficient Last-level Cache Design,” DATE, 2012, pp. 45–50.

[23] D. W. Clark and J. S. Emer, “Performance of the VAX-11/780 translation buffer: simulation and measurement,” *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 31–62, Feb. 1985.

[24] C. Dougan, P. Mackerras, and V. Yodaiken, “Optimizing the Idle Task and Other MMU Tricks,” OSDI, 1999, pp. 229–237.

[25] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, “Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware,” ASPLOS, 2012, pp. 37–48.

[26] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., 2002.

[27] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, April 2012, no. 248966-026.

[28] B. L. Jacob and T. N. Mudge, “A Look at Several Memory Management Units, TLB-refill Mechanisms, and Page Table Organizations,” ASPLOS, 1998, pp. 295–306.

[29] D. Jevdjic, S. Volos, and B. Falsafi, “Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache,” ISCA, 2013, pp. 404–415.

[30] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo, “Characterizing Data Analysis Workloads in Data Centers,” IISWC, 2013, pp. 66–76.

[31] G. B. Kandiraju and A. Sivasubramaniam, “Going the distance for TLB prefetching: an application-driven study,” ISCA, 2002, pp. 195–206.

[32] G. B. Kandiraju and A. Sivasubramaniam, “Characterizing the d-TLB Behavior of SPEC CPU2000 Benchmarks,” SIGMETRICS, 2002, pp. 129–139.

[33] S. Kaxiras and A. Ros, “A new perspective for efficient virtual-cache coherence,” ISCA, 2013, pp. 535–546.

[34] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, “Thin servers with smart pipes: designing SoC accelerators for memcached,” ISCA, 2013, pp. 36–47.

[35] P. Lotfi-Kamran, B. Grot, and B. Falsafi, “NOC-Out: Microarchitecting a Scale-Out Processor,” MICRO, 2012, pp. 177–187.

[36] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, “Scale-out processors,” ISCA, 2012, pp. 500–511.

[37] D. Lustig, A. Bhattacharjee, and M. Martonosi, “TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs,” *ACM Trans. Archit. Code Optim.*, vol. 10, no. 1, pp. 2:1–2:38, Apr. 2013.

[38] C. McCurdy, A. L. Cox, and J. Vetter, “Investigating the TLB Behavior of High-end Scientific Applications on Commodity Microprocessors,” ISPASS, 2008, pp. 95–104.

[39] A. Morari, R. Gioiosa, R. W. Wisniewski, B. S. Rosenburg, T. Inglett, and M. Valero, “Evaluating the Impact of TLB Misses on Future HPC Systems,” IPDPS, 2012, pp. 1010–1021.

[40] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and R. Brown, “Design tradeoffs for software-managed TLBs,” ISCA, 1993, pp. 27–38.

[41] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, “Increasing TLB reach by exploiting clustering in page translations.” HPCA, 2014, pp. 558–567.

[42] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, “CoLT: Coalesced Large-Reach TLBs,” MICRO, 2012, pp. 258–269.

[43] X. Qiu and M. Dubois, “Towards Virtually-Addressed Memory Hierarchies,” HPCA, 2001, pp. 51–62.

[44] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta, “The Impact of Architectural Trends on Operating System Performance,” SOSP, 1995, pp. 285–298.

[45] A. Saulsbury, F. Dahlgren, and P. Stenström, “Recency-based TLB Preloading,” ISCA, 2000, pp. 117–127.

[46] M. Talluri and M. D. Hill, “Surpassing the TLB Performance of Superpages with Less Operating System Support,” ASPLOS, 1994, pp. 171–182.

[47] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, and J. M. Pendleton, “An In-cache Address Translation Mechanism,” ISCA, 1986, pp. 358–365.

[48] C.-J. Wu and M. Martonosi, “Characterization and dynamic mitigation of intra-application cache interference,” ISPASS, 2011, pp. 2–11.

[49] W. Xu, H. Sun, X. Wang, Y. Chen, and T. Zhang, “Design of Last-level On-chip Cache Using Spin-torque Transfer RAM (STT RAM),” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 19, no. 3, pp. 483–493, Mar. 2011.