Large Pages May Be Harmful on NUMA Systems

Fabien Gaud
Simon Fraser University

Baptiste Lepers
CNRS

Jeremie Decouchant
Grenoble University

Justin Funston
Simon Fraser University

Alexandra Fedorova
Simon Fraser University

Vivien Quéma
Grenoble INP

Abstract

Application virtual address space is divided into pages, each requiring a virtual-to-physical translation in the page table and the TLB. Large working sets, common among modern applications, necessitate a lot of translations, which increases memory consumption and leads to high TLB and page fault rates. To address this problem, recent hardware introduced support for *large pages*. Large pages require fewer translations to cover the same address space, so the associated problems diminish.

We discover, however, that on systems with non-uniform memory access times (NUMA) large pages may fail to deliver benefits or even cause performance degradation. On NUMA systems the memory is spread across several physical nodes; using large pages may contribute to the imbalance in the distribution of memory controller requests and reduced locality of accesses, both of which can drive up memory latencies.

Our analysis concluded that: (a) on NUMA systems with large pages it is more crucial than ever to use memory placement algorithms that balance the load across memory controllers and maintain locality; (b) there are cases when NUMA-aware memory placement is not sufficient for optimal performance, and the only resort is to split the offending large pages. To address these challenges, we extend an existing NUMA page placement algorithm with support for large pages. We demonstrate that it recovers the performance lost due to the use of large pages and makes their benefits accessible to applications.

1 Introduction

Applications with large memory working sets require many virtual-to-physical address translations in page tables and TLBs. This drives up physical RAM consumption, increases TLB miss rate, and hurts performance [1][2][10]. According to one report, a large Oracle DBMS installation with 500 concurrent connections consumed 7GB of RAM for page tables alone! [5]. To address this problem, most modern hardware and OS introduced support for large pages. On x86 systems large pages are typically 2MB (512 times larger than regularly-sized 4KB pages), and support for 1GB pages is on the way[1]. Using larger pages requires fewer translations to cover the address space and diminishes the pressure on the TLB and physical memory.

While large pages are so crucial for performance of large-memory systems, they, unfortunately, also have downsides. Previous work reported and addressed increased memory footprints and physical memory fragmentation [13]. In this work, we report on a new problem: ***large pages hurt performance on NUMA systems***.

Modern NUMA systems are comprised of several *processor nodes* each containing a multicore CPU and a local DRAM, all inside a single physical server. The nodes are connected by the high-speed interconnect into a cache-coherent system, forming an abstraction of a single globally addressable memory. While CPUs can transparently allocate and access the memory on any node, accesses to remote nodes traverse the interconnect and access a remote memory controller, incurring higher latency and contributing to congestion on the interconnect. To achieve good performance on NUMA systems, we need to (1) maximize the fraction of memory accesses going to local nodes and (2) balance the traffic across the nodes and interconnect links. Unbalanced distribution of memory requests can increase the memory access latency on the overloaded controller to as many as 1000 cycles, compared to about 200 cycles on a not overloaded controller [6].

In this paper we show that ***large pages can exacerbate harmful NUMA effects, such as poor locality and imbalance***. Using large pages makes the unit of mem-

[1] 1GB pages are already supported by the hardware; support by the OS is still nascent, so few applications are able to use them at the time of the writing.

ory management (a page) more coarse. As a result, it is more likely that many frequently accessed memory addresses happen to map to the same physical page and overload the memory node hosting it – the so-called *hot page effect*. The hot-page effect cannot be addressed by page migration and balancing; page splitting must be performed prior to any attempts to rebalance memory. Likewise, large pages lead to more frequent *page-level false sharing* among threads, where threads access *different* data on the *same* page. False sharing leads to poor locality, which cannot be addressed by page migration alone.

In this work we:

- Quantify the performance degradation due to large pages on NUMA systems. We find that they affect between 25% and 30% of applications in our benchmark set and cause degradations between 5% and 43%.
- Demonstrate that these performance losses are due to NUMA factors, such as poor locality and imbalance.
- Show that the problem can be addressed using a combination of old and new techniques.

Our solution consists of two components: an existing NUMA-aware page placement algorithm Carrefour [6], and large-page extensions to Carrefour: *Carrefour-LP*. For some of the affected applications Carrefour alone is able to recover the lost performance, but in other cases Carrefour is ineffective due to the hot-page effect and page-level false sharing.

Even though hot pages and false sharing touched only a couple of benchmarks in our set, these effects will become pervasive on systems with much larger pages (e.g., 1GB), which are becoming common. Therefore, we implemented Carrefour-LP which addresses these problems by dynamically splitting large pages as needed. For applications affected by hot pages and false sharing, Carrefour-LP improves performance by 10%-80% relative to Carrefour alone. Carrefour together with Carrefour-LP significantly diminish or completely eliminate the performance degradation introduced by large pages and improve performance of some applications by 2-3$\times$ relative to Linux with large pages.

The rest of the paper is structured as follows: Section 2 motivates the work by presenting performance effects of using large pages on NUMA systems, Section 3 presents the solution, Section 4 evaluates it, Section 5 discusses related work, and lastly Section 6 summarizes the paper.

2 Large Pages and Adverse NUMA Effects

2.1 Experimental platform

For our experiments, we used two different server-class machines. Machine A has two 1.7GHz AMD Opteron 6164 HE processors, with 12 cores per processor, and 64GB of RAM. The system is equally divided into four NUMA nodes (i.e., six cores and 12GB of RAM per node). Machine B has four AMD Opteron 6272 processors, each with 16 cores (64 cores in total), and 512GB of RAM. It has eight NUMA nodes – 8 cores and 64GB of RAM per node. Both machines have HyperTransport 3.0 interconnect links.

We are running on Linux 3.9 and are using Transparent Huge Pages (THP) for large page allocation[2]. THP works by backing allocations of anonymous memory with 2MB pages whenever possible. Other kinds of memory, such as memory mapped files, are unaffected by THP and use 4KB pages. THP also uses a kernel thread to periodically scan for free memory regions that are at least 2MB in size, which are then used to replace groups of existing 4KB pages.

We used several benchmark suites representing a variety of different workloads: the NAS Parallel Benchmarks suite which is comprised of numeric kernels, MapReduce benchmarks from Metis, SSCA v2.2 (a graph processing benchmark) with a problem size of 20, and SPECjbb. From the NAS benchmark suite we picked the benchmarks that ran for at least 15 seconds. The memory usage of the benchmarks ranges from 518MB for EP from the NAS suite to 34,291MB for IS from NAS.

2.2 Large Pages on Linux

Figure 1 compares the performance of 4KB pages and 2MB pages using THP. We can see that THP increases performance (by up to 109%) for several benchmarks on both machines (e.g. WC, WR, WRMEM, and SSCA), but also significantly decreases performance by as much as 43% in some cases. CG, UA, and SPECjbb are all negatively affected by THP. Therefore, 2MB pages are not universally beneficial and neither are 4KB pages, so there is no "one size fits all."

To understand this phenomenon, we recorded two metrics that represent the potential benefits of large pages: the number of L2 cache misses caused by page table walks (obtainable from hardware performance counters), and the maximum time spent in the page fault handler by any core. L2 misses due to page table walks is a good indicator for the effect of TLB misses on performance. We expect large pages to increase the TLB coverage and reduce page table sizes. As a result, we expect the number L2 cache misses due to page table walks to drop when we use large pages. Similarly, large pages will reduce the number of page faults for allocations and

[2]Linux also allows using large pages via *libhugetlbfs*, but the latter required recompiling applications and pre-allocating memory for large pages, which was inconvenient, and, moreover, did not perform better than THP in our experiments.

thus the time spent in the page fault handler.

We also monitored two metrics related to NUMA efficiency: the *local access ratio* (LAR), which is the percentage of accesses to local memory, and the *traffic imbalance* on the memory controllers. Traffic imbalance is defined as the standard deviation of the memory request rate across the controllers, expressed as the percent of the mean. For memory intensive applications, a low LAR and a high imbalance signify a NUMA issue.

Table 1 shows the profiling results for a subset of interesting applications. As expected, applications that benefited from 2MB pages in Figure 1 (WC and SSCA) have fewer L2 misses due to page table walks, and for WC significantly less time spent in the page fault handler. The effects can be dramatic. For example, with SSCA on machine A the percentage of L2 misses due to page table walks is decreased from 15% to 2% when using 2MB pages, which results in a 17% performance increase. WC, which experiences a similar decrease in L2 misses but also a large decrease in time spent on page faults, has its performance increased more than two-fold on machine B.

The two other profiled benchmarks – CG and UA – perform much worse with 2MB pages. The profiling reveals that the degradation is caused by NUMA effects. With CG and 4KB pages, the load on the memory controllers is almost perfectly balanced, but with 2MB pages the imbalance is 20% on machine A and 59% on machine B. For UA, the problem is that the LAR decreases when using large pages, from about 88% to around 66%.

SPECjbb presents an interesting case. While the data in Figure 1 suggests that it does not benefit from large pages, profiling reveals that using large pages actually decreases the percent of L2 misses due to page table walks. At the same time, SPECjbb suffers from NUMA issues: the imbalance rises from 16% to 39% with large pages. Therefore, SPECjbb *could benefit* from large pages if NUMA effects were reduced.

3 Solutions

The previous section demonstrated that using large pages may introduce NUMA issues, which may either degrade performance relative to small pages (as they did for CG and UA) or leave the performance unchanged but prevent an application from enjoying the benefits of large pages (as they did for SPECjbb). In this section we first demonstrate that using a NUMA-aware page placement algorithm eliminates the NUMA issues for some applications, motivating the use of NUMA-aware page placement with large pages.

We then identify two new problems that a placement algorithm unaware of large pages does not address: the hot-page effect and the page-level false sharing. These effects, while affecting only two applications in our experiments, will become especially important as much larger pages (e.g., 1GB) come into use. To address them, we introduce large-page extensions (LP) to an existing NUMA placement algorithm.

For clarity of presentation, from now on we will focus on those applications that experience NUMA issues when large pages are used. Specifically, if the LAR or the imbalance is made worse by more than 15% by using large pages as opposed to small ones on either machine, the application is selected for presentation, otherwise it is omitted. The selected applications are: CG.D, LU.B, UA.B, UA.C, matrixmultiply, wrmem, SSCA, SPECjbb. For completeness, and to demonstrate that our solutions do not hurt the applications they cannot help, we do include performance results for the excluded applications at the end of Section 4.

3.1 Page balancing is not enough

We used a NUMA-aware page balancing algorithm Carrefour, which was shown to perform better than other similar solutions [6]. Carrefour works by gathering access samples for memory pages and then choosing a host node for a page based on the samples. If all of the samples for a page originated from a single node, then the page is migrated to that node. If the samples came from multiple nodes, then the page is interleaved (i.e. migrated to a random node). Carrefour also includes thresholds based on hardware counters, so that it is only enabled if NUMA problems are detected such as when the local access ratio is low or the imbalance on memory controllers is high.

We ran Carrefour in the kernel configured with 2M pages (Carrefour-2M). Figure 2 shows the performance of Carrefour-2M compared to Linux with 2M pages (labeled as *THP*) relative to Linux with 4K pages (labeled as *Linux*). We observe that while Carrefour-2M does improve performance for some applications, it fails to solve the problem across the board. For SPECjbb, Carrefour-2M addresses the NUMA issue; as shown in Table 2 it restores the balance on memory controllers that was introduced by large pages and improves the LAR.

At the same time, Carrefour-2M fails to improve performance for UA and CG. To understand why, we show profiling data for these applications in Table 2. We report five metrics: the percentage of total accesses to the most used page (PAMUP), the number of hot pages (NHP) defined as pages comprising more than 6% of the total accesses[3], the percentage of memory accesses to pages

[3]In order to perfectly balance the load on a 8-node NUMA machine, each node must be the target of 12.5% of the total memory accesses. Thus, we consider that if a page represents more than half of this amount, it is likely to create imbalance.

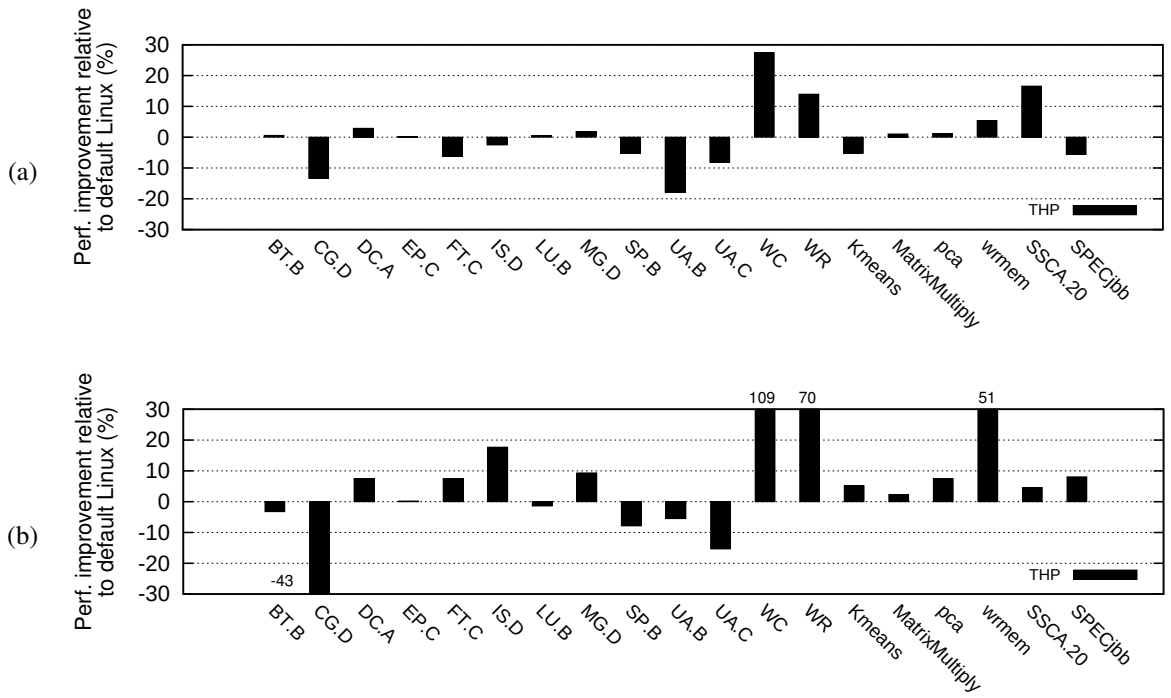

Figure 1: THP performance improvement over Linux on (a) machine A and (b) machine B. THP sometimes perform better than Linux, sometimes worse.

	Perf. incr. THP/4k (%)	**Time spent in page fault handler (% of total time)**		**% L2 misses due to page table walk**		**Local access ratio (%)**		**Imbalance (%)**	
		Linux	THP	Linux	THP	Linux	THP	Linux	THP
CG.D (B)	-43	2182ms (0.1%)	445ms (0%)	0	0	40	36	**1**	**59**
UA.C (B)	-15	102ms (0.2%)	53ms (0.1%)	0	0	**88**	**66**	14	12
WC (B)	109	**8731ms (37.6%)**	**3682ms (32.3%)**	**10**	**1**	50	55	147	136
SSCA.20 (A)	17	90ms (0%)	147ms (0.1%)	**15**	**2**	25	26	**8**	**52**
SPECjbb (A)	-6	8369ms (2.1%)	5905ms (1.5%)	**7**	**0**	12	15	**16**	**39**

Table 1: Detailed analysis of various application on machine A and B. The machine type is indicated in parentheses next to the name of the benchmark.

shared by at least two threads (PSP), the percentage of accesses to local memory (LAR), and the traffic imbalance on the memory controllers.

The results for CG reveal that there is a ***hot page problem***. Large pages cause the heavily accessed regions of the address space to be coalesced into a small number of hot pages (the PAMUP significantly increases), and because there are fewer hot pages than NUMA nodes it is impossible to balance them.

UA does not have a hot page issue, but it does have more pages that are shared among threads when large pages are used (the PSP significantly increases). This happens because each page holds more data and is thus more likely to contain data used by multiple threads. Since the threads do not share data, but share the *page*, we refer to this problem as ***page-level false sharing***. Carrefour-2M is then forced to interleave these pages whereas if there were less sharing the pages could be placed on the nodes where they are most heavily used for maximum locality. As a result, Carrefour-2M delivers a lower LAR than Linux with small pages.

In summary, Carrefour-2M is only able to address NUMA issues induced by large pages in cases where they are not caused by the hot-page effect and page-level false-sharing.

While these problems affected only two applications in our experiments, they will become pervasive as pages much larger than 2MB come into use. 1GB pages are already supported by the hardware; applications like large DBMS clearly motivate their use [5]. We did not evaluate 1GB pages, because they are poorly supported in Linux. 1GB pages are not compatible with THP, and while in

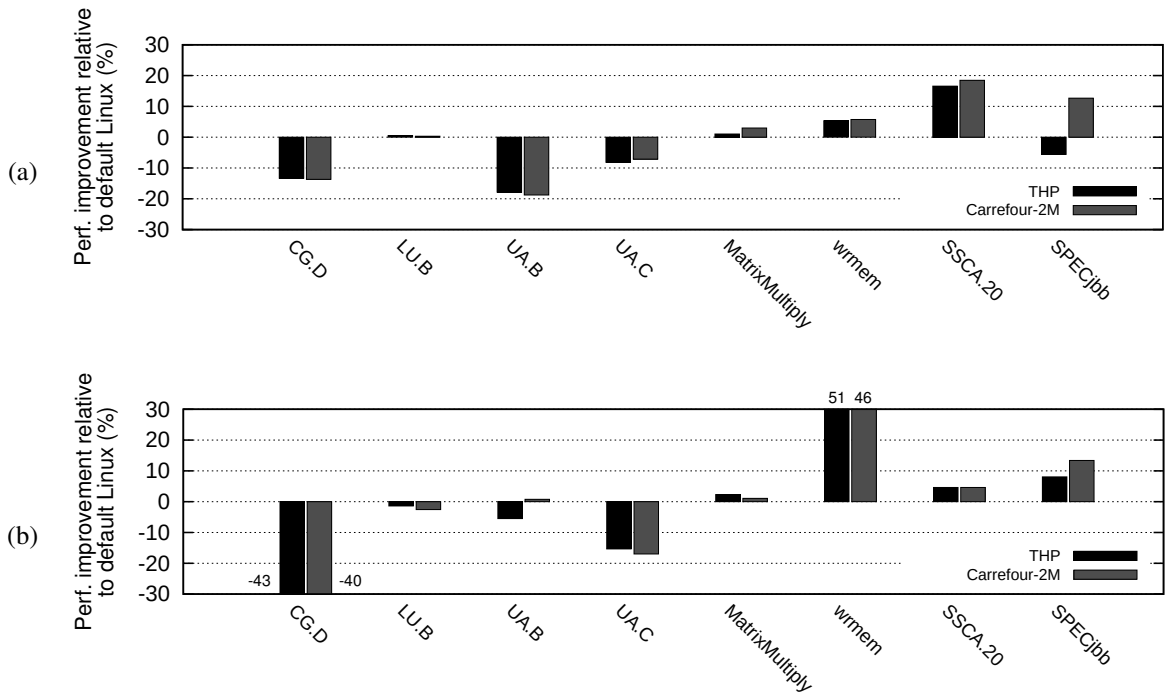

Figure 2: Performance improvement of Carrefour-2M and THP over Linux on applications whose NUMA metrics are affected by THP (2MB pages). Carrefour-2M is not always able to solve the problems for applications that suffer from THP.

theory it is possible to use them with lighugetlbfs, that has many challenges. First of all, the implementation is unreliable. We were not able to enforce the use of 1GB pages with NAS applications and observed many crashes with the Metis suite (because the latter uses a custom memory allocator). Second, the splitting of large pages, which is crucial to our solution, is not supported by libhugetlbfs and implementing it would require a significant effort.

However, since the use-case for very large pages is definitely there, they will become more common as the OS support improves. Then, the hot-page effect and page-level false sharing will become more common (Section 4.4 provides some preliminary data). To address these problems, we propose large-page extensions to Carrefour.

3.2 Carrefour-LP

Intuition suggests two basic solutions to the problem: *conservative* – prevent the problem by only creating large pages when necessary, or *reactive* – start with large pages and fix NUMA problems when they are observed. Each approach has potential benefits and drawbacks. The conservative approach can avoid NUMA related performance degradation but can also miss out on the benefits of large pages. On the other hand, the reactive approach

		Linux	THP	Carrefour 2M
SPECjbb	PAMUP	2%	6%	6%
	NHP	0	0	0
	PSP	**10%**	**36%**	**36%**
	Imbalance	**16%**	**39%**	**19%**
	LAR	26%	28%	27%
CG.D	PAMUP	**0%**	**8%**	**8%**
	NHP	**0**	**3**	**3**
	PSP	18%	34%	34%
	Imbalance	**0%**	**20%**	**20%**
	LAR	45%	45%	45%
UA.B	PAMUP	6%	6%	6%
	NHP	0	0	0
	PSP	**16%**	**70%**	**70%**
	Imbalance	9%	15%	17%
	LAR	**90%**	**61%**	**58%**

Table 2: Proportion of accesses to the most-used page (PAMUP) in %, number of hot pages (NHP), proportion of memory accesses to shared pages (PSP) in %, Imbalance in % and local access ratio (LAR) in % for Linux, THP and Carrefour-2M, on machine A (24 cores).

will benefit from large pages, but must be able to quickly and accurately detect NUMA issues and must pay the overhead of fixing them.

We found that a good algorithm must be a combination of these approaches. The ***reactive component*** of our algorithm continuously monitors the hardware counters looking for the presence of NUMA effects under large pages, applies the page balancing techniques of Carrefour and splits the large pages if the latter are ineffective. The ***conservative component*** of the algorithm continuously monitors the virtual memory metrics and re-enables large pages if they are expected to deliver benefit but were previously disabled.

We also found that it is more practical and involves less overhead to enable large pages in the beginning and disable them later if they are deemed harmful. In particular, many applications have intensive memory-allocation phases at the very beginning of the program that suffer from lock contention if small pages are used.

Our full algorithm is presented in Algorithm 1. Lines 4-9 corresponds to the conservative component, the rest to the reactive component. The algorithm also details the hardware counter metrics that are being monitored. Since the monitoring is done continuously, the algorithm caters to phase changes in applications. Below we describe the rationale behind the decisions made in the algorithm.

3.2.1 Reactive component

The job of the reactive component is to disable large pages when they are harmful to the extent that even Carrefour-2M's page-balancing techniques cannot address the performance degradation. To that end, it estimates the local access ratio (LAR), a vital metric for detecting NUMA issues, with and without Carrefour and large pages.

We use AMD's instruction-based sampling (IBS)[4] to sample memory accesses to pages, and to learn whether the access was made from a local or a remote node. We only consider pages that have at least one sample where the access was serviced from DRAM, so that our decisions are not affected by pages that are easily cached. From the IBS samples, we estimate the LAR that would be obtained if the shared pages were migrated to a random node and if non-shared pages were migrated to the local node (i.e. interleaving and migrating pages with the Carrefour-2M algorithm). We also calculate the LAR that would be obtained if the same technique were used but with all of the 2MB pages split into 4KB pages.

Estimating the LAR for various what-if scenarios (e.g., if a page were migrated or if large pages were split into regular-sized) is trivial with IBS samples. IBS gives us data addresses and the node from which they were accessed. So we can compute the current LAR as well as the LAR that would be obtained if the pages where placed on different nodes. Similarly, we can map the data addresses to 4KB pages and compute the same metrics for the scenario if the large pages were split.

If, based on our estimates, the LAR can be improved by 15% with Carrefour-2M only and without splitting the pages, we simply run Carrefour-2M. Otherwise, if splitting pages would improve the LAR by at least 5%, then all shared 2MB pages are demoted into 4KB pages. Note that we are being cautious here: we try to address NUMA issues by page migration first, and split pages only if absolutely necessary. Splitting pages has overhead and may hurt applications that need them – hence our decision. In addition, large pages with more than 6% of the total accesses (hot pages, as defined in Section 3.1) are split and the constituent 4KB pages are interleaved.

This part of the algorithm relies on two thresholds. The first one is the 15% threshold used to decide whether we can improve the LAR simply by rearranging memory pages, without having to split large pages. That threshold was relatively easy to set across applications: the key is to use a relatively large number, since we want to be rather confident that we can improve performance without having to split pages. The second threshold, the 5% performance gain that we expect from splitting pages, needs to be any non-negligible number that would justify the splitting. Again, that threshold was relatively easy to tune across applications.

In the algorithm, we use the LAR computed per-application. Another option would be to use the LAR computed per-page, however this was difficult to do, because existing hardware monitoring facilities prevent us from obtaining enough samples to accurately compute per-page LAR (and even per-application LAR as explained in the next section). This is why the algorithm splits all 2MB pages when it detects the LAR can be improved.

3.2.2 Conservative component

The job of the conservative component is to re-enable large pages when they have been disabled but monitoring shows that they would be beneficial again. The conservative component uses two criteria to determine the benefit of large pages: the performance impact of TLB misses (based on the fraction of L2 misses caused by page table walks) and the maximum percentage of time any core spends processing page faults. The reason why we consider the time spent processing page faults is that large pages improve performance by decreasing this time. Indeed, soft page faults not only take CPU time, but also

[4]Intel systems have a similar facility called PEBS (Precise Event-Based Sampling).

Algorithm 1 Large-page Extensions to Carrefour

```
1: Enable 2MB page allocation and promotion
2: while true do
3:    Gather hardware performance counters and IBS
      samples for 1 sec
4:    if L2 misses due to page table walks > 5% then
5:       Enable 2MB page allocation
6:       Enable 2MB page promotion
7:    else if Max time spent on page faults > 5% then
8:       Enable 2MB page allocation
9:    end if
10:   if Estimated LAR improvement with only Car-
      refour > 15% then
11:      SPLIT_PAGES = false
12:   else if Estimated LAR improvement with Car-
      refour and splitting pages > 5% then
13:      SPLIT_PAGES = true
14:   end if
15:   if SPLIT_PAGES = true or 2MB page allocation
      is disabled then
16:      Split all shared 2MB pages into 4KB pages
17:      Disable 2MB page allocation
18:   end if
19:   Split and interleave 2MB hot pages
20:   Interleave and migrate pages with Carrefour
21: end while
```

incur costly synchronization [3]. The latter is the reason why we use the *maximum* fraction as opposed to the average: lock contention will be determined by the slowest core that holds page table locks.

The conservative component works as follows. If the impact of TLB misses is estimated to be greater than a threshold of 5%, then 2MB page allocation and 2MB page *promotion*[5] are both enabled via THP. Similarly, if the time spent in the page fault handler was more than a threshold of 5%, then 2MB page allocation is enabled but not 2MB page promotion, since there is little benefit in promoting the pages on which we had already paid the cost of page faults.

In order to estimate the impact of TLB misses on performance, we use *the fraction of L2 cache misses due to page table walks*. This assumes that TLB misses primarily degrade performance when a page table traversal causes an L2-cache miss (in that case, the miss is satisfied either from the L3 cache or from the DRAM, both of which are costly), and that the application's performance is dominated by L2 cache misses. Although this is a coarse approximation, it works well because applications that experience a lot of cache misses due to page table walks are those with large page tables. This implies that they have large memory footprints, and so they are memory-intensive. Therefore, it is safe to assume, for these applications, that variations in performance can be primarily explained by the number of L2 cache misses. Conversely, applications with a very small fraction of L2 cache misses resulting from page table walks are not memory-intensive, so for them the impact of TLB misses is negligible.

[5]Page promotion refers to dynamic consolidation of regular-sized pages into large pages. It is supported by the default Linux kernel. We set the frequency for page promotion checks to every 10ms.

4 Evaluation

4.1 Performance evaluation

Figure 3 shows performance of Carrefour-LP and THP relative to Linux with 4K pages. We continue focusing only on the applications affected by NUMA issues; the remaining applications are presented for completeness in Figure 5. Figure 3 shows that Carrefour-LP:

- restores performance of applications that suffered under large pages and do not stand to benefit from them: CG.D, UA.B, UA.C,
- improves performance of applications that were expected to benefit from THP but did not (or did not benefit fully): SSCA and SPECjbb, both on machine A,
- does not significantly hurt performance of the applications where NUMA effects did not cause performance degradation under large pages and where no performance improvements from large pages were expected (the remaining applications).

We next provide the detailed analysis of Carrefour-LP. We analyze the contribution to performance improvements of its three components: Carrefour-2M, conservative and reactive. We demonstrate when and why it is sufficient to just use Carrefour-2M alone and explain how both conservative and reactive components contribute to the solution. The performance breakdown is shown in Figure 4.

Workloads other than CG.C, UA.B and UA.C are not affected by the hot-page effect and page-level false sharing, so in these cases Carrefour-LP performs similarly to Carrefour-2M alone. It is able to meet the performance of Carrefour-2M with minimal overhead (at most 3.7% on machine A and 2.1% on machine B).

Table 3 demonstrates that Carrefour-LP eliminates the hot-page effect and page-level false sharing and improves NUMA metrics where Carrefour-2M fails. For UA, the LAR drops from about 90% to roughly 60% under THP and remains at that low level under Carrefour-2M. Carrefour-LP is able to restore it almost to the previous level by dynamically splitting pages.

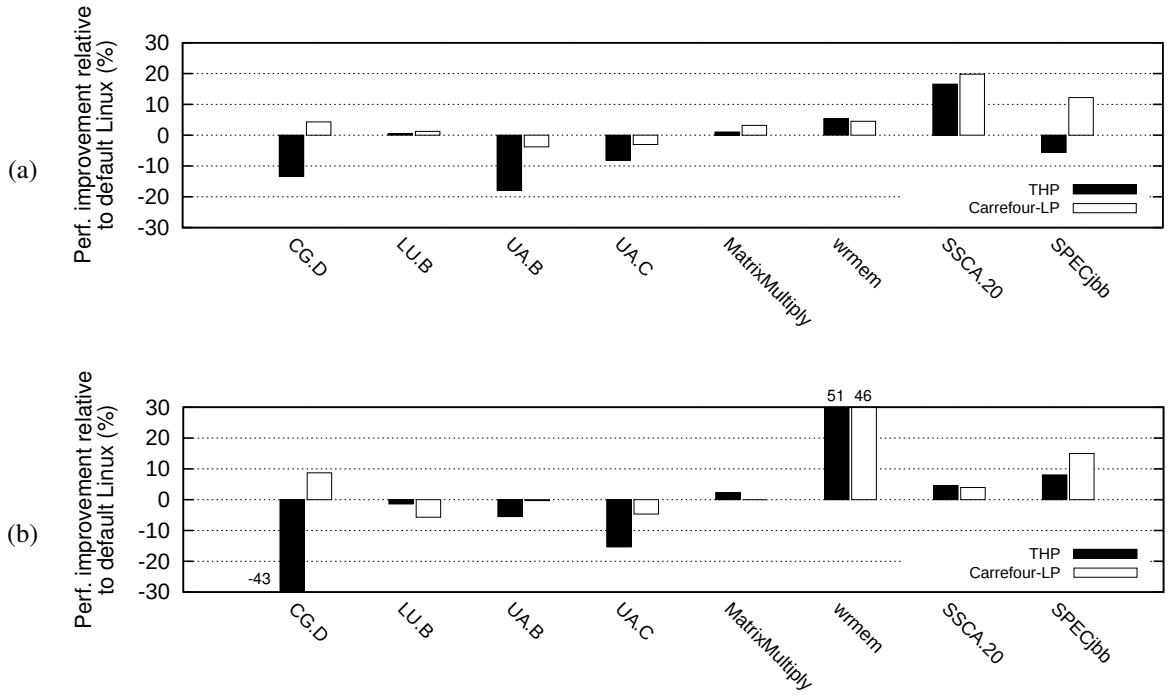

Figure 3: Performance improvement on a reduced set of applications of THP and Carrefour-LP over Linux, on (a) machine A and (b) machine B.

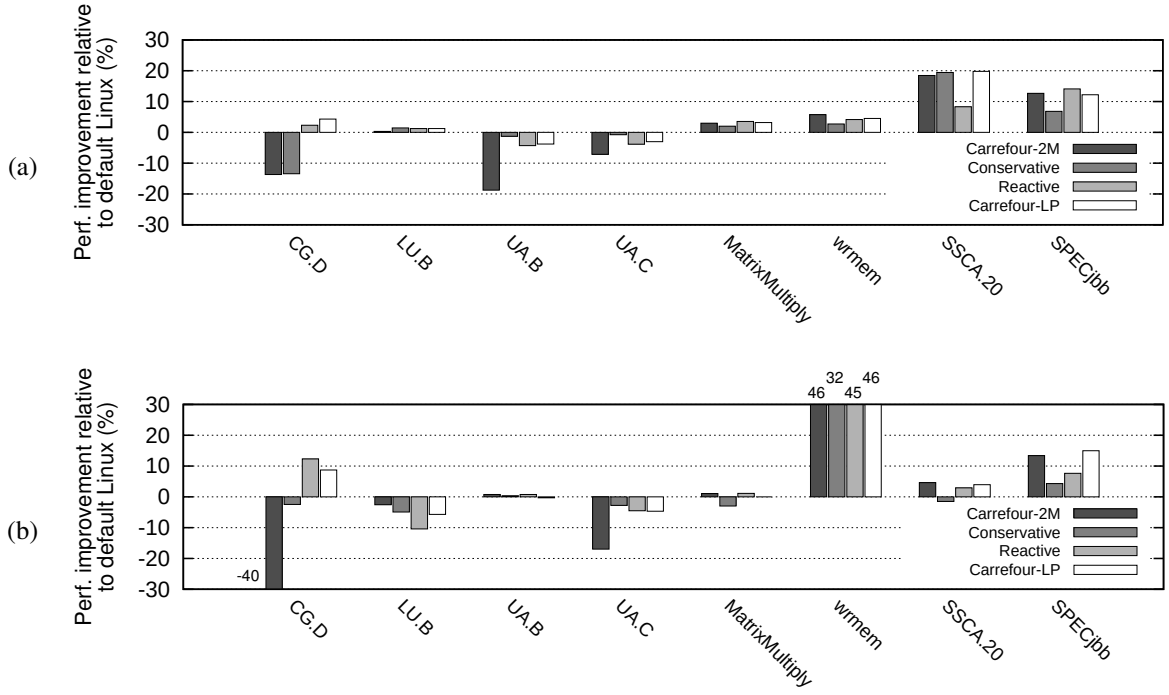

Figure 4: Performance improvement on a reduced set of applications of Carrefour-2M, the conservative component, the reactive component and Carrefour-LP over Linux with THP, on (a) machine A and (b) machine B.

For CG.D, enabling large pages disturbs the perfect memory-controller balance enjoyed under small pages. Carrefour-2M is unable to restore it, while Carrefour-LP restores it almost entirely.

We now analyze the importance of the two components in Carrefour-LP. Figure 4 presents the performance obtained when running Carrefour-2M alone (labeled as *Carrefour-2M*), Carrefour-2M with the reactive component designed for Carrefour-LP (labeled as *Reactive*), the original Carrefour runtime (working on 4kB pages) together with the conservative component (labeled as *Conservative*), and Carrefour-LP (labeled as *Carrefour-LP*). Figure 4 shows that in all cases, enabling the two components (as done in *Carrefour-LP*) is always the best choice (or close to the best). The conservative component alone does not solve the problem, because it begins with 4K pages. For SPECjbb, for example, it does not detect the need for large pages soon enough, so the performance is not as good as it could be. We similarly observed that using the conservative component alone hurts performance of many applications that were not included in this analysis (but shown in Figure 5) for the same reason: large pages were not enabled soon enough. These applications have an intense memory allocation phase at startup, which can benefit greatly from large pages due to fewer page faults, but the conservative component does not enable large pages soon enough.

Using the reactive component alone works well on some applications. For CG.D, it is able to detect the "hot page" and split it. Similarly, it is also able to split the falsely shared pages for UA.B and UA.C. However, on some applications, it fails to bring the maximum performance improvement that can be achieved with 2M pages (e.g. SSCA on machine A and SPECjbb on machine B). The reason is that the LAR is sometimes misestimated, and this results in 2M pages being split in applications that do not suffer from NUMA issues. For instance, on SSCA, the algorithm predicts a LAR of 59% if large pages were all split into 4k pages, whereas the actual LAR obtained after splitting is equal to 25%.

The problem is, in order to estimate the LAR under regular-sized pages given the data samples collected under large pages, we need to have enough samples on the constituent sub-pages. Unfortunately, we found it to be very difficult to gather enough samples; increasing the sampling rate results in unacceptably high overhead. A promising solution would be to use Lightweight Profiling (LWP). LWP is an extension of AMD processors that aims at providing the same level of details as IBS with less overhead. To reduce the overhead, LWP stores samples in a ring buffer and only interrupts the processor when the buffer is full. Unfortunately, on available AMD processors, LWP is only partially implemented: LWP samples only contain the instruction pointer of the sampled instruction and a timestamp. This information is not sufficient to predict LAR.

Because of these deficiencies in hardware profiling, the reactive component may make mistakes in deciding when to split large pages. This is where the conservative component comes to the rescue and re-creates the large pages when they are expected to help.

We conclude this section by explaining some performance results in Figure 5, which contains applications where THP did not create any NUMA issues. The key observation is that the overhead of Carrefour-LP does not significantly hurt these applications. Moreover, EP.C, SP.B and pca enjoy better (sometimes much better) performance with Carrefour-LP than with THP. That is because they had NUMA issues to begin with (which were not exacerbated by large pages), and so the Carrefour-2M component of the algorithm helped to address them.

4.2 Overhead assessment

Overhead in Carrefour-LP comes from collecting and storing IBS samples, computing the metrics based on these samples, migrating and splitting pages. Overall, the overhead of Carrefour-LP compared to the reactive approach is negligible: between 1% and 2% on all applications (on all machines) except CG (3.2%) and IS (2.1%) on machine B. Even on these two applications, the overhead is still within the standard deviation.

Compared to Carrefour-2M, the overhead is also small. The maximum overhead observed is 3.7% on machine A (SP.B) and 3.2% on machine B (LU.B), but on average it is below 2%.

Compared to Linux with 4k pages, Carrefour-LP has an overhead of less than 3%, except on FT, IS (machine A) and LU (machine B). This overhead is not specific to Carrefour-LP but is rather caused by Carrefour-2M, which spends too much time migrating large pages. Since our solution is built on top of Carrefour-2M, it also suffers from the same overhead.

4.3 Discussion

Our assessment of efficacy and downsides of Carrefour-LP is as follows.

The solution could be much improved if we had a more accurate way of estimating the LAR. Currently, with inaccurate estimates, the solution may split and migrate pages when there is no benefit to be gained, which is why Carrefour-LP degrades performance of LU by 3.5% compared to Carrefour-2M. We believe that the LAR could be predicted more accurately if we could collect more data samples without additional overhead. A complete implementation of LWP (i.e., if LWP provided

	Local Access ratio (%)				Imbalance (%)			
	Default Linux	*THP*	*Carr. 2M*	*Carr. LP*	*Default Linux*	*THP*	*Carr. 2M*	*Carr. LP*
CG.D (B)	40	36	38	39	**1**	**59**	**69**	**3**
UA.B (A)	**90**	**61**	**58**	**85**	9	15	17	10
UA.C (B)	**88**	**66**	**68**	**82**	14	12	9	14

Table 3: NUMA metrics for CG.D on machine B, UA.B on machine A, and UA.C on machine B.

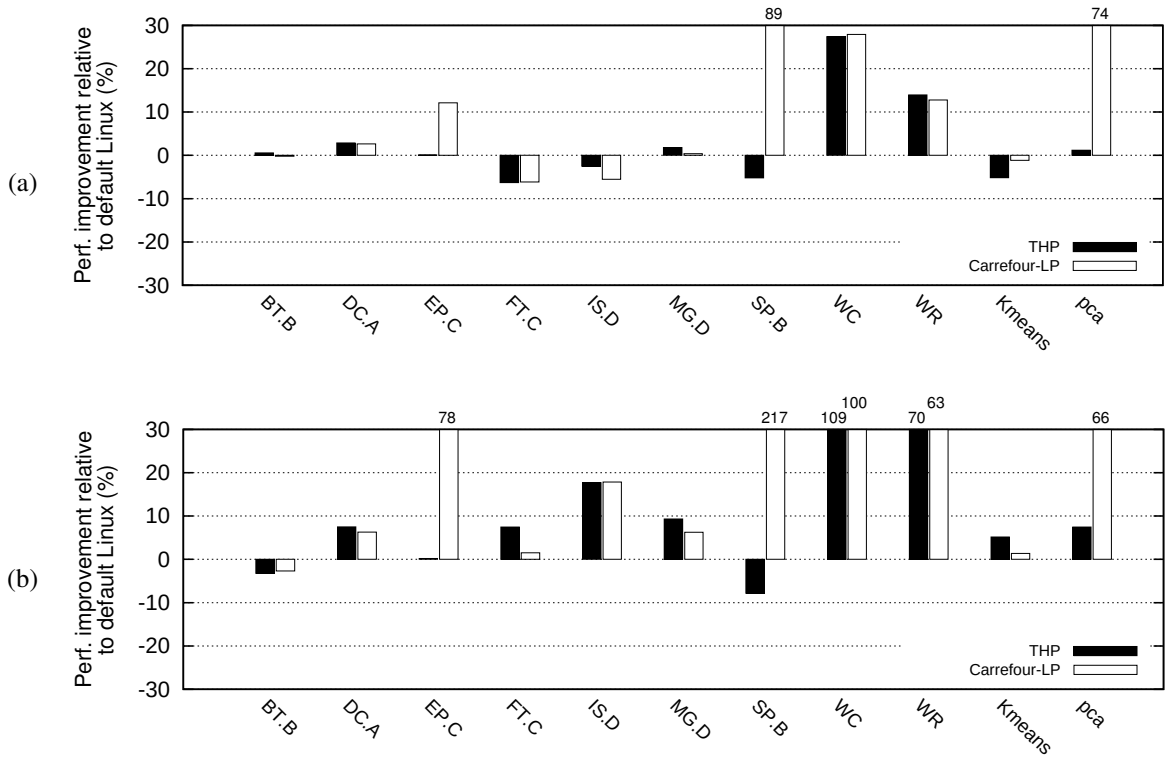

Figure 5: Performance improvement of THP and Carrefour-LP over Linux on applications whose NUMA metrics are **not** affected by THP, on (a) machine A and (b) machine B.

the same kind of samples as IBS) would solve this problem.

Our earlier implementation had scalability issues on the system with 64 cores. The reason was that the centralized data structure where we stored IBS samples had to be accessed and locked from multiple nodes. We addressed this problem by maintaining a data structure per node. The per-node structures are still accessed by multiple cores, so we may need to revisit this scaling issue on larger machines. Overall, the algorithm is likely to scale well because all work generated by an interrupt is performed independently on each node, so the number of nodes can grow without creating scalability bottlenecks.

Splitting pages did not create too much overhead, but the use of the page table lock for THP operations is clearly a scalability concern. Linux developers are working on finer grain locks at the time of the writing, so we hope that this problem will be avoided.

We did not observe many oscillations, where we go back and forth between splitting and enabling large pages. Overall, Carrefour-LP seems to be the more robust than the conservative and the reactive components used independently, because it naturally supports transient states and phase changes by continuously re-examining its decisions.

4.4 Very Large Pages

Although accessing the very large 1GB pages via libhugetlbfs proved challenging for most applications, we were able to enable them in SSCA and in streamclus-

ter (an application from PARSEC)[6]. We immediately observed the hot-page and page-level false-sharing problems. With 1GB pages, lots of hot small pages were coalesced on a single NUMA node, and the performance dropped dramatically. For SSCA it degraded by 34%; for streamcluster by a factor of 4. Neither of these applications suffered performance degradation when 2M pages were used. Although preliminary, these data suggest a much more pervasive presence of NUMA issues when very large pages are used, and so Carrefour-LP will become even more important in the future.

5 Related Work

5.1 Large pages and TLB performance

Several studies have characterized the effect of TLB misses and large pages [2][10][15][14][7]. Battacharjee and Martonosi [2] specifically looked at the effect of TLB misses on multicore systems with multithreaded workloads. They found that some applications, such as Canneal from the PARSEC benchmark suite, spend up to 0.7 cycles per instruction on servicing D-TLB misses. Another study [10] showed performance improvements of up to 25% in the NAS benchmark suite due to using large pages. For large-scale HPC applications, Zhang et al. [15] found that large pages improve communication performance significantly.

Weisberg and Wiseman [14] used the SPEC CPU2000 benchmarks to evaluate the relationship between page size and the number of TLB misses. They argue that a 4KB page size is much too small for most applications, and conclude that a page size of 256KB and a 64-entry TLB is sufficient to drastically reduce the number of TLB misses.

Sudan et al. [12] motivate the need for small pages. They show that using 1KB pages allows optimizing the usage of the DRAM row-buffer, yielding substantial energy savings and decreasing the average latency of memory accesses.

All these works motivate the use of different page sizes, but none of them highlight or quantify the impact of NUMA on the performance obtained when using different page sizes.

5.2 Large page support and optimization

Many software systems have been designed that make large pages easier to use or more effective.

Navarro et al. [9] described an algorithm for operating system support of large pages that reduces fragmentation and does not require memory copies to create large pages. Using their algorithm, a page fault reserves a physical memory region of the size of a large page, but it initially only allocates and maps a small page. Subsequent page faults use the reserved space until it has been completely allocated, at which point the region is promoted to a large page. The algorithm does not attempt to optimize the placement of large pages.

Cascaval et al. [4] developed a model to predict the benefit of using large pages on individual data structures of applications, based on the predicted number of TLB misses and page faults. The predictions are computed using hardware counters throughout multiple runs of the application. The data structures that are predicted to benefit the most from large pages are backed by large pages. A similar method is described in [11], with the major difference being that large page promotions are performed at runtime.

Magee and Qasem [8] also devised a system for restricting the usage of large pages to applications that benefit the most from them. At compile-time, the working-set size is estimated through static analysis. If the estimated working-set size is greater than the coverage of the target CPU's TLB, then large pages are used.

A different approach is explored by Basu et al. [1]. Instead of managing the use of large pages at the OS level, they propose a hardware extension that allows applications to directly map memory segments. Addresses within directly mapped segments bypass the TLB and so translation is nearly free. The segments are conceptually similar to very large pages and provide similar benefits, but the authors do not analyze the potential NUMA effects which would be exacerbated by the large size of the segments.

In summary, previous works mostly focused on the limited availability of large pages and on reducing memory fragmentation. Several systems have been designed to ensure that applications that benefit from large pages actually use them, but no existing work has revealed and addressed the NUMA issues raised by large pages.

6 Conclusion

We demonstrated that using large pages can create or exacerbate NUMA issues like reduced locality or imbalance. We showed that these problems can be in some cases addressed by using a NUMA-aware page placement algorithm, but the latter stumbles upon two problems: the hot-page effect and page-level false sharing, which cannot be addressed via page migration. To address these problems, we implemented Carrefour-LP: large-page extensions to the NUMA-aware page placement algorithm Carrefour. Our results show that Carrefour-LP restores the performance when it was lost

[6]The PARSEC suite was not included in our study, because its applications did not experience performance differences under THP with 2M pages.

due to large pages and makes their benefits accessible to applications.

Solutions like Carrefour-LP will be even more important in the future, when very large pages (1GB in size) will be in widespread use.

References

[1] BASU, A., GANDHI, J., CHANG, J., HILL, M. D., AND SWIFT, M. M. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013), ACM, pp. 237–248.

[2] BHATTACHARJEE, A., AND MARTONOSI, M. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques* (2009), PACT '09, pp. 29–40.

[3] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of linux scalability to many cores.

[4] CASCAVAL, C., DUESTERWALD, E., SWEENEY, P. F., AND WISNIEWSKI, R. W. Multiple page size modeling and optimization. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on* (2005), IEEE, pp. 339–349.

[5] CLOSSON, K. Quantifying Hugepages Memory Savings with Oracle Database 11g, July 2009. http://kevinclosson.wordpress.com/2009/07/28/quantifying-hugepages-memory-savings-with-oracle-database-11g/.

[6] DASHTI, M., FEDOROVA, A., FUNSTON, J., GAUD, F., LACHAIZE, R., LEPERS, B., QUÉMA, V., AND ROTH, M. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems* (2013), ACM, pp. 381–394.

[7] GORMAN, M., AND HEALY, P. Performance characteristics of explicit superpage support. In *Computer Architecture* (2012), Springer, pp. 293–310.

[8] MAGEE, J., AND QASEM, A. A case for compiler-driven superpage allocation. In *Proceedings of the 47th Annual Southeast Regional Conference* (2009), ACM, p. 82.

[9] NAVARRO, J., IYER, S., DRUSCHEL, P., AND COX, A. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (2002), OSDI '02, pp. 89–104.

[10] NORONHA, R., AND PANDA, D. K. Improving scalability of openmp applications on multi-core systems using large page support. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International* (2007), IEEE, pp. 1–8.

[11] ROMER, T. H., OHLRICH, W. H., KARLIN, A. R., AND BERSHAD, B. N. Reducing tlb and memory overhead using online superpage promotion. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on* (1995), IEEE, pp. 176–187.

[12] SUDAN, K., CHATTERJEE, N., NELLANS, D., AWASTHI, M., BALASUBRAMONIAN, R., AND DAVIS, A. Micro-pages: increasing dram efficiency with locality-aware data placement. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems* (2010).

[13] TALLURI, M., KONG, S., HILL, M. D., AND PATTERSON, D. A. Tradeoffs in supporting two page sizes. In *Computer Architecture, 1992. Proceedings., 19th Annual International Symposium on* (1992).

[14] WEISBERG, P., AND WISEMAN, Y. Using 4kb page size for virtual memory is obsolete. In *Information Reuse & Integration, 2009. IRI'09. IEEE International Conference on* (2009), IEEE, pp. 262–265.

[15] ZHANG, P., LI, B., HUO, Z., AND MENG, D. Evaluating the effect of huge page on large scale applications. In *Networking, Architecture, and Storage, 2009. NAS 2009. IEEE International Conference on* (2009), IEEE, pp. 74–81.