Scattered Superpage: A Case for Bridging the Gap between Superpage and Page Coloring

Licheng Chen[†‡], Yanan Wang[†‡], Zehan Cui[†‡], Yongbing Huang[†‡], Yungang Bao[†], Mingyu Chen[†]
[†]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[‡]University of Chinese Academy of Sciences
{chenlicheng, wangyanan, cuizehan, huangyongbing, baoyg, cmy}@ict.ac.cn

***Abstract*—Superpage and page coloring are two important practical techniques to improve the performance of Translation Lookaside Buffers (TLBs) and shared Last Level Cache (LLC) respectively. However, there exists a gap between these two techniques in current hardware-architecture design, resulting in the contradiction in adopting these two optimizations simultaneously: a superpage requires hundreds of contiguous (e.g. a power of two) base pages in both virtual and physical memory, which would compulsorily occupy all available page colors (or cache sets), thus making page coloring failed to work. This is because most contemporary architecture adopts the design with cache set indexes placed in the least significant part of block address.**

In this paper, we propose a lightweight approach named Scattered Superpage to bridge this gap. Scattered Superpage decouples a superpage from the limitation of occupying multiple contiguous physical base pages. A superpage is still contiguous in virtual memory, but it is scattered mapping into multiple physical superpages, and it just occupies specified partial page colors in each physical superpage, thus it allows us to configure page color for each superpage. The huge TLB is slightly modified to store page color configuration for each superpage and to calculate target physical address based on this configuration when doing address translation. The experimental results show that the Scattered Superpage can improve system performance by 20.51% and reduce unfairness by 27.77% in our 4-core simulation system (with multi-program memory-intensive workloads). It achieves this by reducing last level cache miss by 17.05% and reducing TLB miss by 86.02% simultaneously.

Keywords*—*Scattered Superpage, Page Coloring, TLB, Last Level Cache

I. Introduction

The DRAM memory system is considered as the main bottleneck in chip multiprocessor (CMP) system [18, 27]. There are two main sources that would result in high overhead DRAM accesses: 1) normal data access (e.g. load/store) miss in last level cache (LLC); 2) page table walks due to translation lookaside buffer (TLB) miss. The former contributes most of DRAM references, and it is considered critical to system performance, thus a large body of work has been contributed to LLC optimization. Although the latter contributes relatively smaller portion of DRAM references, it also becomes

This work is supported by the National Natural Science Foundation of China (NSFC) under grant numbers 60925009, 60921002, 60903046, the National Basic Research Program of China (973 Program) under a grant number 2011CB302502, the Strategic Priority Research Program under a grant number XDA06010401, and Huawei Research Program under a grant number YBCB2011030. Yungang Bao is partially supported by the CCF-Intel Young Faculty Research Program (YFRP) Grant.

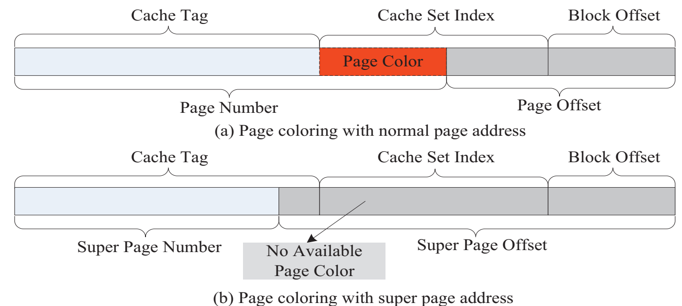

Fig. 1. Page coloring with normal page address [12] and with super page address.

increasingly important to system performance. It has been shown that TLB miss can have an impact on overall system performance by 5% to 14% for nominally sized applications [7] and even up to 50% for some larger applications [25]. Thus to achieve better system performance in multi-core/many-core system, it is necessary to take both shared LLC and TLB into account. However, most of previous work focused on either LLC optimization or TLB optimization, none of them took both factors into consideration, which thus motivates this work.

Superpage (or huge page) and page coloring are two important practical techniques to improve the performance of Translation Lookaside Buffers (TLBs) and shared Last Level Cache (LLC) respectively, and each of them has been widely used in real systems in separate. However, in this work, we will show that there exists a gap between **superpage** and **page coloring** in current hardware-architecture design, which would result in the contradiction in adopting both these two optimizations. And the goal of this work is to bridge the gap and to leverage both superpage and page coloring optimizations.

The **page coloring** technique is a software approach for shared cache partition, that is able to reduce cache contention through isolated mapping between virtual pages and physical pages among multiple concurrently executing applications (or threads). It can be implemented in operating systems (OS) [21] or in user level [12], and it has been proven to be practical and effective in real systems [4, 12, 20, 21, 23, 31, 38, 40]. Figure 1(a) shows the principle of page coloring with normal page (e.g. 4KB). A physical address is divided into *page offset* (least significant bits) and *page number* (most significant bits). The physical address is also used to index last level cache.

978-1-4799-2987-0/13/$31.00 ©2013 IEEE

And as a cache address, it contains three parts, from least to most significant are: *block offset*, *cache set index*, and *cache tag*. The common bits of *page number* and *cache set index* is named as **page color bits**. By manipulating each process, thread or object to use non-overlapping page colors, page coloring technique can isolate cache sets and thus can reduce cache contention among them. Actually it is especially useful to protect some good-locality data from being interfered by weak-locality data.

Superpage is used to reduce TLB miss overhead by increasing TLB coverage [3, 13, 15, 29, 33, 36, 37]. A superpage (or huge page)[1] is a page that is sized and aligned as a power of two multiple of system's base page, which are required to be contiguous both in virtual and physical memory. Hundreds of contiguous memory pages (within a superpage) that can be mapped with a single huge TLB entry, thus using superpage can boost TLB coverage and reduce TLB miss overhead. For example, x86 systems support 2MB and recently 1GB superpages while adopt 4KB as base pages. The x86 processors also provide separate huge TLB with each entry covering 512 base pages (for 2MB superpage). The performance of memory-intensive applications can improve by about 10.45% in our real experimental system with 2MB superpages (please refer to section II for detail).

However, there exists a gap between **page coloring** and **superpage**, as shown in Figure 1(b). It is noteworthy that page coloring technique works only if there exists overlapping bits between *cache set index* and *page number*. In contemporary hardware-architecture design, the *cache set index* is placed in the least significant part of block address (removing *block offset*). When using superpage, the *super page offset* (also in the least significant bits) would cover all the *cache set index* bits. It means that no available page color bits could be controlled by OS, and each superpage would compulsively occupy all the available page colors. Take a typical processor with 64B-block 8MB 16-way set-associative shared last level cache as an example, there are total 8K sets, thus the *cache set index* has 13 bits and resides in [6:18] bits of cache address ([0:5] are *block offset* bits), which is totally covered by the 2MB *super page offset* bits with [0:20]. Equipping with larger-size LLC is a straightforward way to make some page color bits available along with superpage (e.g. a 16-way 64 MB LLC has only 1 available page color bit), however increasing LLC size would introduce significant area, cost and energy overhead to processer.

In this work, we propose a new lightweight approach named **Scattered Superpage** to bridge the gap between superpage and page coloring. Scattered superpage decouples superpage from the limitation of using multiple contiguous physical base pages that would probably occupy all page colors. Instead, a virtual superpage (which is still contiguous in virtual address) is scattered mapping into multiple physical superpages with using specified (or controlled) partial page colors in each physical superpage. Scattered superpage allows software (OS or applications) to assign non-overlapping page colors to superpages from different processes or threads, which could effectively reduce last level cache contention among superpages. Thus it enables us to leverage both superpage and page coloring optimizations to improve both TLB and LLC performance. To implement scattered superpage, the huge TLB needs to be slightly modified to maintain page color configuration for each superpage, and this configuration is used to calculate target physical address when doing address translation (for scattered superpage) in huge TLB.

[1]In this paper, we use the term superpage and huge page interchangeably, and we use huge TLB for address translation of huge page.

Overall, we have made the following contributions:

- We identity the gap between superpage and page coloring in current hardware-architecture design, which results in the contradiction in adopting both these two optimizations.

- To bridge this gap, we propose a lightweight approach named **scattered superpage** that enables us to optimize both last level cache and TLB performance. A virtual superpage is scattered mapping into multiple physical superpages with each physical superpage using specified partial page colors. The huge TLB needs to maintain page color configuration for each superpage, and to calculate target physical address based on this configuration. To the best of our knowledge, this is the first work to leverage both **superpage** and **page coloring** optimizations to improve both the performance of TLB and shared LLC simultaneously.

- We implement scattered superpage in a cycle-accurate multi-core simulator and evaluate it using multi-program memory-intensive workloads on a 4-core system. Compared to the baseline setup, scattered superpage can improve system performance by 20.51% and reduce unfairness by 27.77% due to the reduction of LLC miss (17.05%) and TLB miss (86.02%).

The rest of the paper is organized as follows: Section II introduces our motivation and Section III describes the implementation of scattered superpage. We describe the experimental methodology in Section IV and demonstrate experimental results and discussion in Section V. Related work and conclusion are in Section VI and Section VII respectively.

II. Motivation

To motivate our work, we first show the performance improvement for memory-intensive applications with huge TLB on a real system. The experimental system has an Intel Xeon E5645 processor working at 2.40GHz, it has 2-level data TLB for 4KB base page. The first level TLB (DTLB) has a size of 64 and the second level TLB (STLB) has a size of 512. Furthermore, it has a separate 32-entry huge TLB for 2MB superpage, which could increase maximum TLB coverage to 64MB address space. We run each application five times with single instance that is bound to the same core. And we use libhugetlbfs [1] to access to huge pages of memory for applications.

Figure 2 shows the normalized performance speedup of memory-intensive applications when using huge TLB. We can see that, the average performance improves by about 10.45%. Except *470.lbm* (3.62%) and *403.gcc* (3.67%), all the other applications have improvements above 5%, and the maximum improvement is about 36.23% for *429.mcf*, this is because *429.mcf* has a large working set of about 1.6GB with random memory access pattern that results massive TLB misses for 4KB base page. And using superpage can effectively reduce TLB miss overhead. Thus, we can conclude that superpage is

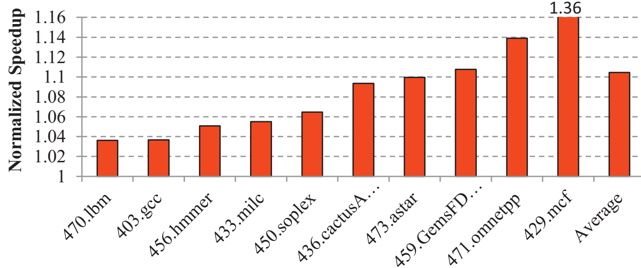

Fig. 2. Normalized speedup of memory-intensive benchmarks with using huge TLB (2MB superpage) on a real system.

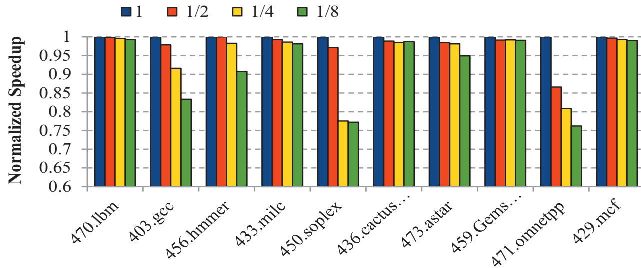

Fig. 3. Normalized speedup of memory-intensive benchmarks with different page colors (or cache sets) in simulation, the baseline is using all page colors (it is 64 in our simulation).

a practical and effective optimization, especially for memory-intensive applications with large working set.

We then show that many of these memory-intensive applications could have a small performance degradation when assigned with partial page colors, which could be used as a proof for the efficiency of page coloring. We adopt a Linux kernel that is modified to support page coloring in our simulation system, and it is configured with a total of 64 page colors (please refer to section IV for the detailed simulation configuration).

Figure 3 shows the normalized performance speedup (in IPC) of memory-intensive applications with different page colors varied from 1 (whole) to 1/8. We can see that 5 of the applications (*470.lbm, 433.milc, 436.cactusADM, 459.GemsFDTD, 429.mcf*) have performance degradation less than 2% even with only 1/8 page colors, these applications are either CCF (Core Cache Fitting: the working set size fits in the smaller levels of the cache hierarchy) or LLCT (LLC Thrashing: the working set size is greater than the available LLC) as defined in [17]. Two applications (*456.hmmer, 473.astar*) are degraded less than 2% with 1/4 page colors which could be classified to LLCFR (LLC Friendly, benefit from the available shared LLC) [17]. And the other 3 applications (*403.gcc, 450.soplex, 471.omnetpp*) need at least 1/2 page colors. These degradation results could be used to guide number of colors assigned to each application when adopting page coloring optimization. Take workload1 shown in section IV for example, we could assign 1/4 page colors for *456.hmmer*, 1/2 for *433.gcc*, 1/8 for *429.mcf* and 1/8 for *470.lbm*, and as shown in section V, the performance improvement is about 15.63% for workload1 with page coloring.

III. Scattered Superpage

A. Design and Implementation

Without loss of generality and to simplify our description, in the following subsection, we assume that each (2MB) superpage contains all available page colors of the system and each color appears exactly once in each superpage. The principle and implementation could be easily extended for other page color configurations.

Before describing scattered superpage, we firstly introduce and define **Color Region (CR)**. A Color Region consists of a set of contiguous physical base pages, which means that it contains multiple physical page colors. A color region is served as a base configurable logic color for a superpage[2]. Thus, a superpage is divided into multiple color regions. The number of color regions in each superpage should be configured based on the number of cores in the system, to insure each core having enough available number of color regions. For example, in our 4-core system, we divide each 2MB superpage into 8 configurable color regions.

In scattered superpage, a superpage is remain contiguous in virtual memory and it provides the same access interface for software, thus it doesn't need any changes for software to adopt scattered superpage. A virtual superpage is scattered mapping into multiple physical superpages, and it is not contiguous in physical memory any more. Actually it only uses partial specified color regions in each physical superpage, and multiple of these scattered (specified) color regions constitute an integral physical superpage. Other unused color regions can be used by other scattered superpages (with different color region configuration). Since different scattered superpages (from different processes or threads) can be configured to use non-overlapping color regions, it can effectively reduce cache contention among them. And each scattered superpage still needs only one huge TLB entry for its address translation.

Before accessing scattered superpage, each superpage needs to be configured to use which and how many color regions. There are two parameters: *CR_Entry* and *CR_Num*, which represents the start color region and the number of color regions used in a physical superpage respectively. We limit the *CR_Num* to be a power of two (e.g. 1,2,4,8) and the *CR_Entry* to be aligned with *CR_Num*. The *CR_Num* determines how many scattered physical superpages should be used, and it is named as the *Scattered_Num* of a scattered superpage:

$$Scattered_Num = Total_CR_Num / CR_Num \quad (1)$$

Where *Total_CR_Num* represents the total number of configurable color regions in each physical superpage, *CR_Num* represents the number of color regions used in each physical superpage, and *Scattered_Num* represents the number of scattered physical superpages used for a scattered superpage.

Figure 4 shows the virtual to physical address mapping for a regular superpage and for a scattered superpage. For a regular superpage as shown in Figure 4(a), a virtual superpage is mapped into a physical superpage, and the *superpage offset* is contiguous both in virtual and physical address space.

[2]Since each superpage contains hundreds of base pages, it is not necessary to make each page configurable in individual.

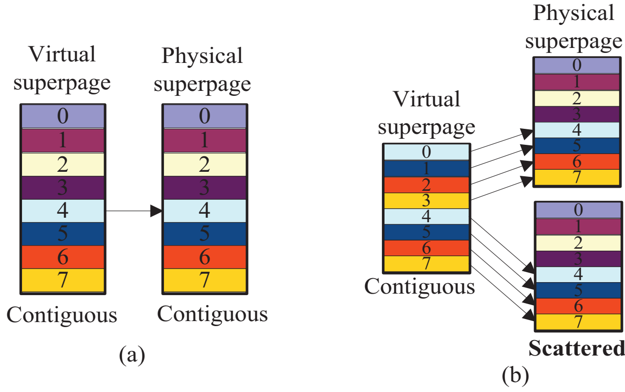

Fig. 4. The virtual to physical address mapping for regular superpage (a) and for scattered superpage (b).

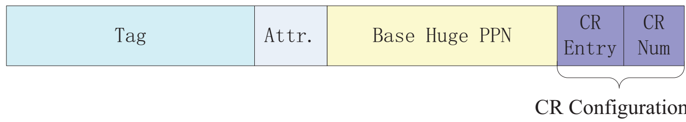

Fig. 5. The huge TLB entry for scattered superpage.

Thus a regular superpage needs to occupy all the available 8 color regions. For a scattered superpage as shown in Figure 4(b), although it is still contiguous in virtual superpage, it is scattered mapping into multiple physical superpages with using specified color regions in each physical superpage. In this example, it has total 8 color regions in each superpage, and the scattered superpage is configured with *CR_Entry*=4 and *CR_Num*=4, then the *Scattered_num* of it is 2 (8/4), which means that it is scattered mapping into 2 physical superpages, and in each physical superpage, it uses the entry color region of 4 and four color regions of {4,5,6,7}.

Although *Scattered Superpage* would affect the allocation, promotion of superpages and might introduce fragmentation in superpages (with unused color regions). This could be properly handled, since the *Scattered Superpage* is flexible enough for different workload scenarios. Actually the *Scattered Superpage* is compatible with the regular *Superpage*: when each physical superpage is configured to use all the available color regions (with *CR_Entry*=0 and *CR_Num*=*Total_CR_Num*), it would act the same as the regular superpage. Thus for LLC-interference intensive workloads, fine-grained color region configuration could be adopted to reduce LLC interference. And for other workloads, coarse grained or even regular superpages could be adopted to avoid fragmentation.

To support address translation for scattered superpage, each huge TLB entry is extended to store the color region (CR) configuration as shown in Figure 5. Besides the baseline *Tag*, *Attr.* (page attributes), and *Base Huge PPN* (Base Huge Physical Page Number), it needs a bit extra storage for *CR_Entry* and *CR_Num*. This CR configuration is used to calculate physical address when doing address translation.

B. Address Translation for Scattered Superpage

Figure 6 shows how the huge TLB does address translation for a regular superpage. A virtual superpage address is divided into two parts: *Virtual Huge Page Number* and *Huge Page Offset*. The address translation needs two steps:

1) **Searching**: The *Virtual Huge Page Number* is used to search in the huge TLB to check whether it is hit in huge TLB. For set-associative TLB, it firstly indexes the target set and then compares with all the Tags in the set. If it is a TLB miss, it needs to fetch the target entry from page table (in memory) through one or more page walks. After searching, the corresponding *Base Huge PPN (Physical Page Number)* is got.

2) **Combining**: To get the whole physical address, the *Base Huge PPN* is used to combine with the original *Huge Page Offset* (which puts *Base Huge PPN* in the most significant part and puts *Huge Page Offset* in the least significant part). It is noteworthy that the *Huge Page Offset* of a regular superpage remains the same in both virtual address and physical address.

Thus, the physical address could be expressed as:

$$Physical\ Address = Base\ Huge\ PPN \oplus^{3} Huge\ Page\ Offset \tag{2}$$

It needs a little more effort to do address translation for a scattered superpage. In virtual address, besides *Virtual Huge Page Number*, the *Huge Page Offset* is further divided into *Scattered Shift* and *CR Offset* as shown in Figure 7. The *Scattered Shift* is used to index target physical superpage among multiple scattered physical superpages, and it is determined by the *Scattered_Num*:

$$Scattered\ Shift = log(Scattered_Num) \tag{3}$$

For example, as shown in Figure 4(b), the *Scattered_Num* is configured as 2, thus the *Scattered Shift* is 1 (log(2)), and it represents the most 1 significant bit of *Huge Page Offset*. If the value of *Scattered Shift* bit is 0, it would be mapped into the first scattered physical superpage; and if its value is 1, it would be mapped into the second scattered physical superpage.

The address translation now needs three steps as shown in Figure 7:

1) **Searching**: This is the same with regular superpage: the *Virtual Huge Page Number* is used to search in huge TLB to get the *Base Huge PPN*. In scattered superpage, the *Base Huge PPN* represents the base Huge PPN of the first scattered physical superpage.

2) **Scattering**: the CR configuration is fetched from the target TLB entry, then it is used to calculate the *Base CR PPN*, which represents the target Color Region base physical address in target physical superpage. The *Scattered Shift* is firstly determined based on equation (3) with *CR_Num*, and then it is used to index target physical superpage. In parallel, the *CR_Entry* is used to index the target base color region in physical superpage.

3) **Combining**: The *CR Offset* is used to combine with the target *Base CR PPN* to get the whole physical address. Thus in scattered superpage, the *Huge Page Offset* is not contiguous in physical address any more, but the *CR Offset* remains the same both in virtual address and physical address.

[3] It represents combining operation.

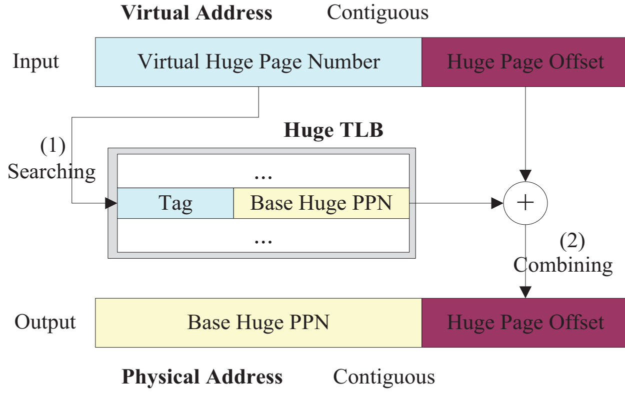

Fig. 6. Address translation for regular superpage.

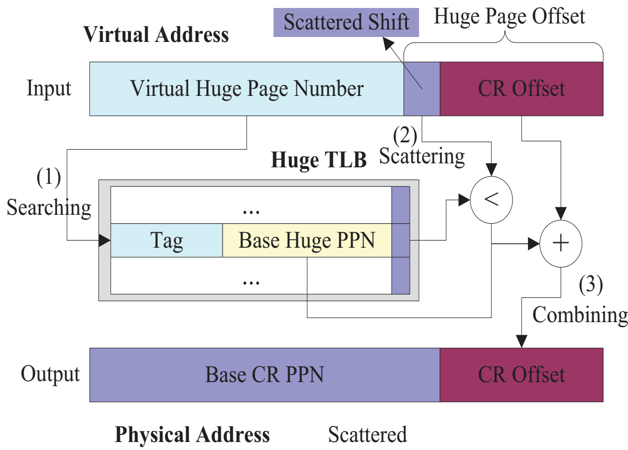

Fig. 7. Address translation for scattered superpage.

In conclusion, the physical address could be expressed as:

$$Physical\ Address\ = (Base\ Huge\ PPN\ +\ val(Scattered\ Shift)) \oplus CR_Entry\ \oplus\ CR_Offset \quad (4)$$

Where *val(Scattered Shift)* represents the value of *Scattered Shift* in the virtual address.

C. Overheads

Space Overhead: Each huge TLB entry needs to be extended with extra storage for *CR Configuration*. We model a 4-core system with 8 configurable color regions, thus the number of bits of *CR_Entry* is 3. Since we limit the *CR_Num* to be a power of two, it could be configured with 4 different values (1,2,4 or 8), thus it just needs 2 bits for *CR_Num* (where *00* represents 1, *01* represents 2, *10* represents 4 and *11* represents 8). Thus the total number of bits for *CR Configuration* is 5, and the total space overhead is only 160 bits for the 32-entry huge TLB in our system.

Latency Overhead: It needs an extra *Scattering* step when doing address translation for scattered superpage, which requires one addition operation and two combining operations as shown in equation (4). The two combining operations could be done in parallel with *Combining* step, thus the latency overhead is only 1-cycle latency (for addition operation). A possible optimization is to limit all the huge TLB entries in each TLB Set to be configured with the same *CR Configuration*. After indexing to the target TLB set, its *CR Configuration* could be accessed ahead. Thus the *Scattering* step could be done in parallel with tag comparison in the set, and no extra latency overhead would be introduced. Furthermore the space overhead could be reduced to 40 bits, because each set could share one *CR Configuration* (it is 4-way huge TLB in our system, 160/4).

IV. Experimental Methodology

We use the MARSSx86 [30] full system cycle-accurate simulator to model a four OoO (Out-of-Order) x86 cores system. It has two-level cache, the L1-I (Instruction) and L1-D (Data) cache is private and has a size of 128KB in each core. The L2 data cache is shared among four cores, the size of it is 4MB and it has 4K 16-way sets with a total of 64 available page colors. In our simulation, we adopt the Linux kernel 2.6.32.12 which is modified to support page coloring technique. We add a separate 4-way set-associative huge TLB for superpages as the baseline, and then we implement **scattered superpages** on the huge TLB. Since this work is mainly focus on last level cache and TLB performance, we adopt the simple main memory model in simulation. The system configuration is shown in Table I.

TABLE I. System Configuration.

CPU	Four 2.54GHz OoO cores, 4-wide issue, 128 entry reorder buffer
Caches	L1-I Cache: private, 128KB, 8-way, 64B cache line, 2-cycle latency L1-D Cache: private, 128KB, 8-way, 64B cache line, 2-cycle latency L2-D Cache: shared, 4MB, 16-way, 64B cache line, 14-cycle latency
TLB	ITLB for 4KB pages: private, 4-way, 256-entry DTLB for 4KB pages: private, 4-way, 256-entry huge DTLB for 2MB pages: private, 4-way, 32-entry
Memory	Simple main memory model, 8GB, 64-bank, 130-cycle latency

We use multi-program memory-intensive workloads from the SPEC CPU 2006 [2] benchmarks for the evaluation. We run 16 groups of four-core workloads with one benchmark dedicated to each core. The workloads are shown in table II. All benchmarks are run with their reference (maximum size) input. We first fast-forward 10 billion instructions of each benchmark and then simulate total 1 billion instructions for each workload.

TABLE II. Workloads.

	Workload	**LLC MPKI**	**DTLB MPKI**
WL1	hmmer, gcc, mcf, lbm	8.18	34.19
WL2	omnetpp, gcc, mcf, milc	9.11	40.53
WL3	astar, hmmer, lbm, milc	3.19	15.01
WL4	gcc, hmmer, milc, lbm	6.90	11.73
WL5	gcc, astar, lbm, mcf	8.65	44.65
WL6	soplex, gcc, lbm, mcf	5.51	12.04
WL7	soplex, hmmer, lbm, milc	1.98	16.45
WL8	hmmer, gcc, omnetpp, milc	12.80	7.88
WL9	soplex, astar, gcc, lbm	1.66	11.61
WL10	omnetpp, hmmer, mcf, lbm	4.48	18.40
WL11	gcc, astar, cactusADM, GemsFDTD	0.58	23.09
WL12	soplex, astar, mcf, cactusADM	6.70	46.20
WL13	soplex, hmmer, lbm, cactusADM	9.37	56.15
WL14	hmmer, gcc, cactusADM, mcf	2.56	17.90
WL15	soplex, gcc, milc, lbm	6.23	7.42
WL16	soplex, omnetpp, milc, lbm	5.27	12.61

MPKI represents Misses Per Kilo-Instruction

Metrics: For multi-program workloads, we use *System_Throughput* [27, 34] to evaluate system performance and evaluate QoS in term of *Unfairness* [14, 28], which is the ratio between the maximum slowdown and the minimum slowdown among all processes sharing the last level cache:

$$Slowdown_i = \frac{IPC_i^{shared}}{IPC_i^{alone}} \tag{5}$$

$$System_Throughput = \sum_i Slowdown_i \tag{6}$$

$$Unfairness = \frac{Max_i\{Slowdown_i\}}{Min_i\{Slowdown_i\}} \tag{7}$$

V. EXPERIMENTAL RESULTS

We compare four different approaches to show the benefits of scattered superpage:

- **Original**: each workload is run without any optimization, which would encounter severe last level cache contention and TLB miss, and this is served as the baseline.
- **Page Coloring**: each workload is run with page coloring optimization. We assign each application with intuitional number of color regions in each workload based on the result shown in Figure 3. It could only reduce cache contention.
- **Superpage**: all applications in each workload are run with (2MB) superpage and thus use huge TLB for address translation. Since the TLB is private in each core, there is no inter-TLB-interference among cores. Superpage could effectively reduce TLB miss, but it could not reduce cache contention.
- **Scattered Superpage**: We use (2MB) superpage for all applications in each workload, and assign intuitional number of page colors for each application (as the same with **Page Coloring**). It is able to reduce cache contention as well as to reduce TLB miss.

A. Performance

Figure 8 shows the normalized performance speedup of 4-core memory-intensive workloads in term of *System_Throughput*, where the baseline is *Original*. We can see that all the three optimizations could improve system performance, and the average speedup for *Page Coloring*, *Superpage* and *Scattered Superpage* is about 6.50%, 15.04% and 20.51% respectively. Besides WL1 (workload), WL4 and WL7, all the other workloads get higher performance speedup with *Super Page* than with *Page Coloring*, and the *Page Coloring* optimization shows small improvement for some workloads (e.g. 2.00% for WL5, 2.55% for WL6, and 3.35% for WL9). That is because in our experiments, we roughly chose page color configuration based on Figure 3, and we did not tune page color configuration to achieve the best speedup. The speedup of *Page Coloring* would be higher after tuning. *Super Page* optimization is effective for multi-program workloads, besides WL4 and WL7, all the other workloads has a speedup more than 10%. For all the workloads, the *Scattered Superpage* has the largest speedup, because it could reduce both cache contention and TLB miss. And the most speedup of *Scattered Superpage* is about 29.94% for WL2.

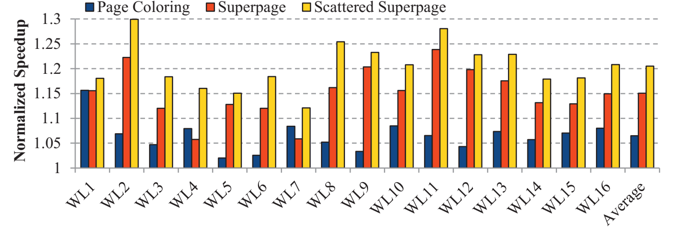

Fig. 8. The performance speedup of 4-core memory-intensive workloads in term of normalized system throughput.

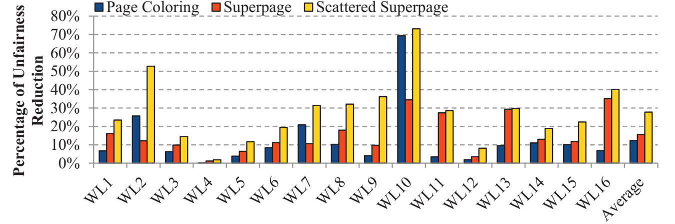

Fig. 9. The percentages of unfairness reduction of 4-core memory-intensive workloads.

B. Unfairness

Figure 9 shows the percentage of unfairness reduction of 4-core memory-intensive workloads. We can see that *Scattered Superpage* could reduce the maximum unfairness, and the average reduction is about 27.77%. While the average reduction for *Page Coloring* and *Superpage* is about 12.43% and 15.64% respectively. The WL10 had a maximum unfairness reduction, it is about 69.29%, 34.46% and 73.17% for *Page Coloring*, *Superpage* and *Scattered Superpage* respectively. This result shows that *Scattered Superpage* could also effectively reduce *Unfairness* among applications.

C. Cache Miss and TLB Miss Reduction

Figure 10 shows the shared L2 Cache (Last Level Cache) miss reduction of 4-core memory-intensive workloads. We can see that both the *Page Coloring* and *Scattered Superpage* could reduce L2 Cache miss by about 25.75% and 17.05% respectively. While *Superpage* has uncertain effect on L2 Cache, it might make worse for some workloads (e.g. -3.89% for WL1, -9.45% for WL6 and -5.00% for WL16) or make better for other workloads (e.g. 12.69% for WL3 and 16.91% for WL9). The average reduction is about 1.74%, thus it is unable to reduce cache contention. The probable reason for it is that by adopting *Superpage*, it would affect physical memory allocation (each superpage requires hundreds of contiguous physical base pages) and thus affect memory access pattern on L2 Cache.

Figure 11 shows the percentage of TLB miss reduction of 4-core memory-intensive workloads. We can see that both *Superpage* and *Scattered Superpage* could greatly reduce TLB miss, and the average reduction is about 91.06% and 86.02% respectively. The main reason for the difference of TLB miss reduction is that *Scattered Superpage* would affect physical superpage allocation (or layout) with page color controlling and thus would affect the memory access pattern to superpage

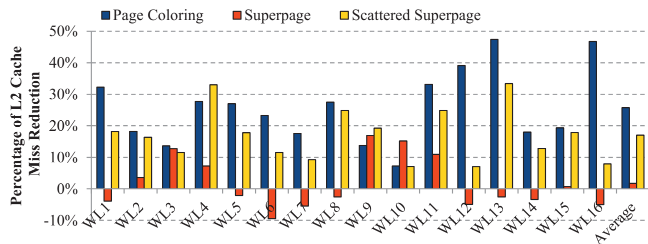

Fig. 10. The percentage of L2 cache miss reduction of 4-core memory-intensive workloads.

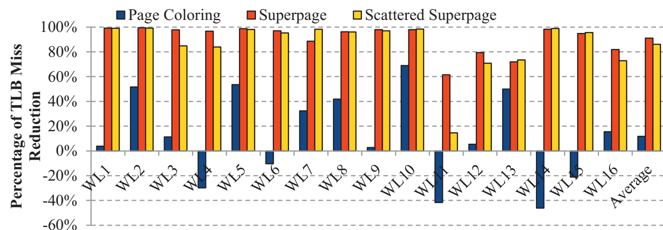

Fig. 11. The percentages of TLB miss reduction of 4-core memory-intensive workloads.

TLB. And we can also see that *Page Coloring* has uncertain effect on TLB miss, the average reduction of it is about 11.76%.

In conclusion, *Scattered Superpage* could reduce both cache contention and TLB miss, which results in the best performance improvement (20.51%).

VI. Related Work

Superpage: Superpages offer a way to increase TLB coverage without increasing the number of TLB entries, which need to be supported by OS [15, 29, 36, 37], recently transparent hugepage support has been implemented in Linux system [3]. Romer et al. [33] study several different policies for dynamically creating superpages. Fang et al. [13] proposed a hardware mechanism for dynamically creating superpages in shadow address. A shadow superpage could be mapped to multiple arbitrary physical pages in memory controller, thus their approach decoupled superpage from using multiple contiguous physical pages. However in shadow address, a superpage was still required to be contiguous. Since the shadow address was used to index last level cache, it would still conflicted with page coloring. Our work adds a simple address mapping in huge TLB, which is able to bridge the gap between superpage and page coloring.

Page Coloring: Page coloring was first proposed by Cho and Jin [11] to manage data placement in a tiled CMP system. Static page coloring was first implemented by Tam et al. [38] for cache partition, and dynamic page coloring was implemented by Lin et al. [21] which was based on page-copying in (DRAM) memory. Lin et al. [20] further proposed a low-overhead hardware-based cache management mechanism to support flexible and effective page re-coloring by eliminating page migration costs. Awasthi et al. [4] proposed a hardware-centric mechanism to implement page migration at low overheads while eliminating DRAM page copies within a static-NUCA cache. They used shadow address spaces to introduce another level of indirection before looking up the L2 cache. The TLB is modified to store New Page Color (NPC) which is used to generate a new shadow address for cache (indexing). Although our work also needs to slightly modify huge TLB, our goal is to bridge the gap between superpage and page coloring, while their work aimed at dynamic page-recoloring in NUCA for base page (TLB), and they did not take superpage into account. Hardavellas et al. proposed Reactive NUCA [16] which explored OS control of cache placement to optimize block placement in distributed caches, and the page table was slightly extended to support page classification. The Tilera TILEPro64 [6] adopted a software configurable cache management approach for the home core for a cache line. Soares et al. [35] dynamically re-mapped cache unfriendly pages to a pollute buffer in cache based on page coloring technique. Zhang et al. [40] proposed a hot-page coloring approach to enforce coloring on only a small set of frequently accessed pages. Ding et al. [12] proposed ULCC which enabled programmers to optimize last level cache usage at user level based on page coloring technique. Liu et al. [22] extended the scope of software page coloring technology for DRAM bank partition, which could efficiently alleviate bank-level interference in multi-core systems. Page coloring could also used to optimize object-level partition [23], Databases [19] and HPC applications [31].

TLB Optimization: TLB is critical to overall system performance. Bhattacharjee et al. [9] characterized TLB behavior on chip multiprocessors and then proposed Inter-core cooperative TLB [10] and shared last level TLB [8, 24]. Pham et al. [32] proposed CoLT to coalesce multiple virtual-to-physical page translation into single TLB entries, which could effectively eliminate TLB misses. Barr et al. [5] proposed SpecTLB that provided speculative translations for many TLB misses on small pages without referencing the page table, which would effectively hide the execution latency of these TLB misses. Zhang et al. [39] proposed Enigma that deferred address translation until main memory needs to be accessed, they introduced a new intermediate address (IA) space for cache addressing and coherence traffic. Previous work also evaluated TLB miss impact on high-end scientific applications [25], future HPC system [26] and virtualization [7].

VII. Conclusion

In this paper, we first identify the gap between superpage and page coloring in current hardware-architecture design, resulting in the contradiction in adopting these two optimizations simultaneously. To bridge this gap, we propose a lightweight approach named **Scattered Superpage**: a superpage is remain contiguous in virtual memory, but it is scattered mapping into multiple physical superpages with using specified partial page colors in each physical superpage. The huge TLB is slightly modified to maintain color region configuration for each superpage, and to calculate target physical address based on this configuration. The experimental results show that Scattered Superpage can improve system performance by 20.51% and reduce unfairness by 27.77% due to the reduction of LLC miss (17.05%) and TLB miss (86.02%). To the best of our knowledge, this is the first work to leverage both superpage and page coloring optimizations to improve both the TLB and shared LLC performance.

REFERENCES

[1] "libhugetlbfs." [Online]. Available: http://libhugetlbfs.sourceforge.net/

[2] "Spec cpu2006 results." Standard Performance Evaluation Corporation. [Online]. Available: http://www.spec.org/cpu2006/

[3] A. Arcangeli, "Transparent hugepage support," in *KVM Forum*, 2010.

[4] M. Awasthi *et al.*, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *HPCA 2009*, pp. 250–261.

[5] T. W. Barr, A. L. Cox, and S. Rixner, "Spectlb: a mechanism for speculative address translation," in *ISCA 2011*, pp. 307–318.

[6] S. Bell *et al.*, "Tile64 - processor: A 64-core soc with mesh interconnect," in *ISSCC 2008. Digest of Technical Papers*, pp. 88–598.

[7] R. Bhargava *et al.*, "Accelerating two-dimensional page walks for virtualized systems," in *ASPLOS 2008*, pp. 26–35.

[8] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level tlbs for chip multiprocessors," in *HPCA 2011*, pp. 62–73.

[9] A. Bhattacharjee and M. Martonosi, "Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors," in *PACT 2009*, pp. 29–40.

[10] A. Bhattacharjee and M. Martonosi, "Inter-core cooperative tlb for chip multiprocessors," in *ASPLOS 2010*, pp. 359–370.

[11] S. Cho and L. Jin, "Managing distributed, shared l2 caches through os-level page allocation," in *MICRO 2006*, pp. 455–468.

[12] X. Ding, K. Wang, and X. Zhang, "Ulcc: a user-level facility for optimizing shared cache performance on multicores," in *PPoPP 2011*, pp. 103–112.

[13] Z. Fang *et al.*, "Reevaluating online superpage promotion with hardware support," in *HPCA 2001*, pp. 63–.

[14] R. Gabor, S. Weiss, and A. Mendelson, "Fairness and throughput in switch on event multithreading," in *MICRO 2006*, pp. 149–160.

[15] N. Ganapathy and C. Schimmel, "General purpose operating system support for multiple page sizes," in *USENIX ATC 1998*, pp. 8–8.

[16] N. Hardavellas *et al.*, "Reactive nuca: near-optimal block placement and replication in distributed caches," in *ISCA 2009*, pp. 184–195.

[17] A. Jaleel *et al.*, "Cruise: cache replacement and utility-aware scheduling," in *ASPLOS 2012*, pp. 249–260.

[18] Y. Kim *et al.*, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *MICRO 2010*, pp. 65–76.

[19] R. Lee *et al.*, "Mcc-db: minimizing cache conflicts in multi-core processors for databases," in *VLDB 2009*, pp. 373–384.

[20] J. Lin *et al.*, "Enabling software management for multicore caches with a lightweight hardware support," in *SC 2009*, pp. 14:1–14:12.

[21] J. Lin *et al.*, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *HPCA 2008*, pp. 367–378.

[22] L. Liu *et al.*, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *PACT 2012*, pp. 367–376.

[23] Q. Lu *et al.*, "Soft-olp: Improving hardware cache performance through software-controlled object-level partitioning," in *PACT 2009*, pp. 246–257.

[24] D. Lustig, A. Bhattacharjee, and M. Martonosi, "Tlb improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level tlbs," *TACO 2013*, vol. 10, no. 1, pp. 2:1–2:38.

[25] C. McCurdy, A. L. Coxa, and J. Vetter, "Investigating the tlb behavior of high-end scientific applications on commodity microprocessors," in *ISPASS 2008*, pp. 95–104.

[26] A. Morari *et al.*, "Evaluating the impact of tlb misses on future hpc systems," in *IPDPS 2012*, pp. 1010–1021.

[27] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *ISCA 2008*, pp. 63–74.

[28] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO 2007*, pp. 146–160.

[29] J. Navarro *et al.*, "Practical, transparent operating system support for superpages," in *OSDI 2002*, pp. 89–104.

[30] A. Patel *et al.*, "MARSSx86: A Full System Simulator for x86 CPUs," in *DAC 2011*.

[31] S. Perarnau, M. Tchiboukdjian, and G. Huard, "Controlling cache utilization of hpc applications," in *ICS 2011*, pp. 295–304.

[32] B. Pham *et al.*, "Colt: Coalesced large-reach tlbs," in *MICRO 2012*, pp. 258–269.

[33] T. H. Romer *et al.*, "Reducing tlb and memory overhead using online superpage promotion," in *ISCA 1995*, pp. 176–187.

[34] A. Snavely and D. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreading processor," in *ASPLOS 2000*, pp. 234–244.

[35] L. Soares, D. Tam, and M. Stumm, "Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer," in *MICRO 2008*, pp. 258–269.

[36] M. Talluri and M. D. Hill, "Surpassing the tlb performance of superpages with less operating system support," in *ASPLOS 1994*, pp. 171–182.

[37] M. Talluri *et al.*, "Tradeoffs in supporting two page sizes," in *ISCA 1992*, pp. 415–424.

[38] D. Tam *et al.*, "Managing shared l2 caches on multicore systems in software," in *WIOSCA 2007*.

[39] L. Zhang *et al.*, "Enigma: architectural and operating system support for reducing the impact of address translation," in *ICS 2010*, pp. 159–168.

[40] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *EuroSys 2009*, pp. 89–102.