Dynamic Hardware-Assisted Software-Controlled Page Placement to Manage Capacity Allocation and Sharing within Large Caches *

Manu Awasthi, Kshitij Sudan, Rajeev Balasubramonian, John Carter
School of Computing, University of Utah

Abstract

In future multi-cores, large amounts of delay and power will be spent accessing data in large L2/L3 caches. It has been recently shown that OS-based page coloring allows a non-uniform cache architecture (NUCA) to provide low latencies and not be hindered by complex data search mechanisms. In this work, we extend that concept with mechanisms that dynamically move data within caches. The key innovation is the use of a shadow address space to allow hardware control of data placement in the L2 cache while being largely transparent to the user application and off-chip world. These mechanisms allow the hardware and OS to dynamically manage cache capacity per thread as well as optimize placement of data shared by multiple threads. We show an average IPC improvement of 10-20% for multi-programmed workloads with capacity allocation policies and an average IPC improvement of 8% for multi-threaded workloads with policies for shared page placement.

Keywords: *page coloring, shadow-memory addresses, cache capacity allocation, data/page migration, last level caches, non-uniform cache architectures (NUCA).*

1. Introduction

Future high-performance processors will implement hundreds of processing cores. The data requirements of these many cores will be fulfilled by many megabytes of shared L2 or L3 caches. These large caches will likely be heavily banked and distributed on chip: perhaps, each core will be associated with one bank of the L2 cache, thus forming a tiled architecture as in [37, 42]. An on-chip network connects the many cores and cache banks (or tiles). Such caches represent a non-uniform cache architecture (NUCA) as the latency for each cache access is a function of the distance traveled on the on-chip network. The design of large last-level caches continues to remain a challenging problem for the following reasons: (i) Long wires and routers in the on-chip network have to be traversed to access cached data. The on-chip network can contribute up to 36% of total chip power [23, 40] and incur delays of nearly a hundred cycles [28]. It is therefore critical for performance and power that a core's data be placed in a physically nearby cache bank. (ii) On-chip cache space is now shared by multiple threads and multiple applications, leading to possibly high (destructive) interference. Poor

allocation of cache space among threads can lead to suboptimal cache hit rates and poor throughput.

Both of the above two problems have been actively studied in recent years. To improve the proximity of data and computation, dynamic-NUCA policies have been proposed [1–3, 9, 10, 18, 20, 22]. In these policies, the ways of the cache are distributed among the various banks and a data block is allowed to migrate from a way in one bank to another way in a different bank that is hopefully closer to the core accessing this data block. The problem with these approaches is the need for a complex search mechanism. Since the block could reside in one of many possible ways, the banks (or a complex tag structure) must be systematically searched before a cache hit/miss can be declared. As the number of cores is scaled up, the number of ways will have to also likely be scaled up, further increasing the power and complexity of the search mechanism.

The problem of cache space allocation among threads has also been addressed by recent papers [19, 29, 35, 38]. Many of these papers attempt to distribute ways of a cache among threads by estimating the marginal utility of an additional way for each thread. Again, this way-centric approach is not scalable as a many-core architecture will have to support a highly set-associative cache and its corresponding power overheads. These way-partitioning approaches also assume uniform cache architectures (UCA) and are hence only applicable for medium-sized caches.

Recent work by Cho and Jin [11] puts forth an approach that is inherently scalable, applicable to NUCA architectures, and amenable to several optimizations. Their work adopts a static-NUCA architecture where all ways of a cache set are localized to a single bank. A given address thus maps to a unique bank and a complex search mechanism is avoided. The placement of a data block within the cache is determined by the physical memory address assigned to that block. That work therefore proposes OS-based page coloring as the key mechanism to dictate placement of data blocks within the cache. Cho and Jin focus on the problem of capacity allocation among cores and show that intelligent page coloring can allow a core to place its data in neighboring banks if its own bank is heavily pressured. This software control of cache block placement has also been explored in other recent papers [25, 30]. Note that the page-coloring approach attempts to split sets (not ways) among cores. It is therefore more scalable and applies seamlessly to static-NUCA designs.

But several issues still need to be addressed with the page coloring approach described in recent papers [11, 25, 36]: (i) Non-trivial changes to the OS are required. (ii) A

page is appropriately colored on first touch, but this may not be reflective of behavior over many phases of a long-running application, especially if threads/programs migrate between cores or if the page is subsequently shared by many cores. (iii) If we do decide to migrate pages in response to changing application behavior, how can efficient policies and mechanisms be designed, while eliminating the high cost of DRAM page copies?

This paper attempts to address the above issues. We use the concept of shadow address spaces to introduce another level of indirection before looking up the L2 cache. This allows the hardware to dynamically change page color and page placement without actually copying the page in physical memory or impacting the OS physical memory management policies. We then describe robust mechanisms to implement page migration with two primary applications: (i) controlling the cache capacity assigned to each thread, (ii) moving a shared page to a location that optimizes its average access time from all cores.

This work therefore bridges advances in several related areas in recent years. It attempts to provide the benefits of D-NUCA policies while retaining a static-NUCA architecture and avoiding complex searches. It implements cache partitioning on a scalable NUCA architecture with limited associativity. It employs on-chip hardware supported by OS policies for on-chip data movement to minimize the impact on physical memory management and eliminate expensive page copies in DRAM.

The rest of the paper is organized as follows. Section 2 sets this work in the context of related work. Section 3 describes the proposed mechanisms. These are evaluated in Section 4 and we conclude in Section 5.

2. Related Work

In recent years, a large body of work has been dedicated to intelligently managing shared caches in CMPs, both at finer (cache line, *e.g.*, [2, 3, 7, 18, 22]) and coarser (page based, *e.g.*, [11, 13, 25, 30]) granularities. Given the vast body of literature, we focus our discussion below to only the most related pieces of work.

Cache partitioning has been widely studied of late [19, 29, 35, 38]. Almost all of these mechanisms focus on way-partitioning, which we believe is inherently non-scalable. These mechanisms are primarily restricted to UCA caches. The use of way-partitioning in a NUCA cache would require ways to be distributed among banks (to allow low-latency access for each core's private data), thus requiring a complex search mechanism. This work focuses on set partitioning with page coloring and allows lower complexity and fine-grained capacity allocation.

A number of papers attempt to guide data placement within a collection of private caches [7, 14, 34]. This work focuses on a shared cache and relies on completely different mechanisms (page coloring, shadow addresses, etc.) to handle capacity and sharing. It is worth noting that a private cache organization is a special case of a shared cache

augmented with intelligent page coloring (that places all private data in the local cache slice).

A number of papers propose data block movement in a dynamic-NUCA cache [1–3, 9, 10, 18, 20, 22]. Most of these mechanisms require complex search to locate data [1, 3, 18, 20, 22] or per-core private tag arrays that must be kept coherent [9, 10]. We eliminate these complexities by employing a static-NUCA architecture and allowing blocks to move between sets, not ways. Further, we manage data at the granularity of pages, not blocks. Our policies attempt to migrate a page close to the center of gravity of its requests from various cores: this is an approach borrowed from the dynamic-NUCA policy for block movement in [3].

The most related body of work is that by Cho and Jin [11], where they propose the use of page coloring as a means to dictate block placement in a static-NUCA architecture. That work shows results for a multi-programmed workload and evaluates the effect of allowing a single program to borrow cache space from its neighboring cores if its own cache bank is pressured. Cho and Jin employ static thresholds to determine the fraction of the working set size that spills into neighboring cores. They also color a page once at first touch and do not attempt page migration (the copying of pages in DRAM physical memory), which is clearly an expensive operation. They also do not attempt intelligent placement of pages within the banks shared by a single multi-threaded application. Concurrent to our work, Chaudhuri [8] also evaluates page-grain movement of pages in a NUCA cache. That work advocates that page granularity is superior to block granularity because of high locality in most applications. Among other things, our work differs in the mechanism for page migration and in our focus on capacity allocation among threads.

Lin et al. [25] extend the proposals by Cho and Jin and apply it to a real system. The Linux kernel is modified to implement page coloring and partition cache space between two competing threads. A re-mapped page suffers from the cost of copying pages in DRAM physical memory. Lin et al. acknowledge that their policies incur high overheads and the focus of that work is not to reduce these overheads, but understand the impact of dynamic OS-based cache partitioning on a real system. In a simulation-based study such as ours, we are at liberty to introduce new hardware to support better cache partitioning mechanisms that do not suffer from the above overheads and that do not require significant OS alterations. We also consider movement of shared pages for multi-threaded applications.

Ding et al. [13] employ page re-coloring to decrease conflict misses in the last level cache by remapping pages from frequently used colors to least used colors. Their work deals exclusively with reducing conflicts for a single thread in a single-core UCA cache environment. Rafique et al. [30] leverage OS support to specify cache space quotas for each thread in an effort to maximize throughput and fairness. That work does not take advantage of page coloring and ultimately resorts to way-partitioning at the block

level while meeting the quota restrictions.

A key innovation in our work is the use of shadow address spaces and another level of indirection within the L2 cache to manage page re-mapping at low overheads. This is not a feature of any of the related work listed thus far. Shadow address spaces have been previously employed in the Impulse memory controller [5] to efficiently implement superpages and scatter-gather operations. The use of another level of indirection before accessing the L2 has been previously employed by Min and Hu [27] to reduce conflict misses for a single-thread UCA cache platform.

Page coloring for data placement within the cache was extensively studied by Kessler and Hill [21]. Several commercial systems have implemented page migration for distributed memory systems, most notably SGI's implementation of page-migration mechanisms in their IRIX operating system [12]. LaRowe et al. [24, 31, 32] devised OS support mechanisms to allow page placement policies in NUMA systems. Another body of work [6, 39] explored the problem from multiprocessor compute server perspective and dealt with similar mechanisms as LaRowe et al. to schedule and migrate pages to improve data locality in cc-NUMA machines. The basic ideas in this paper also bear some similarities to S-COMA [33] and its derivatives (R-NUMA [15] and Wildfire [16]). But note that there are no replicated pages within our L2 cache (and hence no intra-L2 cache coherence). Key differences between our work and the cc-NUMA work is our use of shadow addresses to rename pages elegantly, the need to be cognizant of bank capacities, and the focus on space allocation among competing threads. There are also several differences between the platforms of the 90s and multi-cores of the future (sizes of caches, latencies, power constraints, on-chip bandwidth, transistors for hardware mechanisms, etc.).

In summary, while we clearly stand on the shoulders of many, the key novel contributions of this work are:

- We introduce a hardware-centric mechanism that is based on shadow addresses and a level of indirection within the L2 cache to allow pages to migrate at low overheads within a static-NUCA cache.

- The presence of a low-overhead page migration mechanism allows us to devise dynamic OS policies for page movement. Pages are not merely colored at first touch and our schemes can adapt to varying program behavior or even process/thread migration.

- The proposed novel dynamic policies can allocate cache space at a fine granularity and move shared pages to the center of gravity of its accesses, while being cognizant of cache bank pressure, distances in a NUCA cache, and time-varying requirements of programs. The policies do not rely on a-priori knowledge of the program, but rely on hardware counters.

- The proposed design has low complexity, high performance, low power, and policy flexibility. It represents the state-of-the-art in large shared cache design, providing the desirable features of static-NUCA
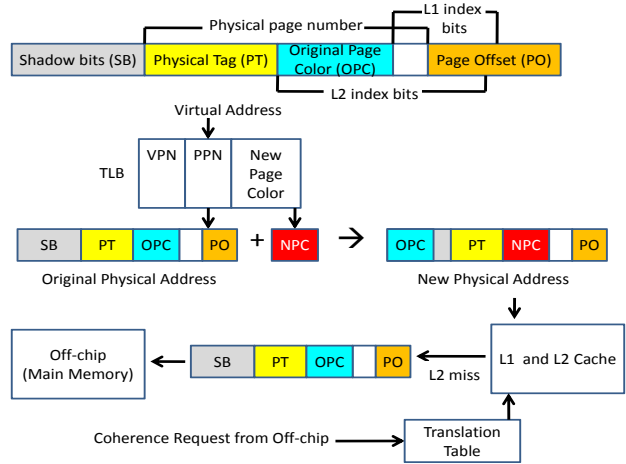

Figure 1. Address structure and address modifications required to access migrated pages.

(simple data look-up), dynamic-NUCA (proximity of data and computation), set-partitioning (high scalability and adaptability to NUCA), hardware-controlled page movement/placement (low-cost migration and fine-grained allocation of space among threads), and OS policies (flexibility).

3. Proposed Mechanisms

We first describe the mechanisms required to support efficient page migration. We avoid DRAM page copies and simply change the physical address that is used internally within the processor for that page. We then discuss the policies to implement capacity allocation and sharing. The discussion below pertains to a multi-core system where each core has private L1-D/I caches and a large shared L2. Each L2 block maintains a directory to keep track of L1 cached copies and implement a MESI coherence protocol.

3.1 Page Re-Coloring
Baseline Design

In a conventional cache hierarchy, the CPU provides a virtual address that is used to index into the L1 cache and TLB. The TLB converts the *virtual page number (VPN)* to a *physical page number (PPN)*. Most L1 caches are virtually indexed and physically tagged and the output of the TLB is required before performing the tag comparison.

The top of Figure 1 shows the structure of a typical physical address. The intersection of the physical page number bits and the *cache index* bits are often referred to as the *page color bits*. These are the bits that the OS has control over, thereby also exercising control over where the block gets placed in cache. Without loss of generality, we focus on a subset of these bits that will be modified by our mechanisms to alter where the page gets placed in L2 cache. This subset of bits is referred to as the *Original Page Color (OPC)* bits in Figure 1.

Modern hardware usually assumes 64-bit wide memory addresses, but in practice only employs a subset of these 64 bits. For example, SUN's UltraSPARC-III architecture [17] has 64-bit wide memory addresses but uses only

44 and 41 bits for virtual and physical addresses, respectively. The most significant 23 bits that are unused are referred to as *Shadow Bits (SB)*. Since these bits are unused throughout the system, they can be used for internal manipulations within the processor.

Page Re-Naming

The goal of our page migration mechanism is to preserve the original location of the page in physical memory, but refer to it by a new name within the processor. When the virtual address (VA) indexes into the TLB, instead of producing the original true physical address (PA), the TLB produces a new physical address (PA'). This new address is used to index into the L1 and L2 caches. If there is an L2 cache miss and the request must travel off-chip, PA' is converted back to PA before leaving the processor. In order for these translations to happen efficiently and correctly, we must make sure that (i) complex table look-ups are not required and (ii) the new name PA' does not overwrite another existing valid physical address. This is where the shadow bits can be leveraged.

Unique Addresses

When a page is migrated (renamed within the processor), we change the OPC bits of the original address to a set of *New Page Color (NPC)* bits to generate a new address. We then place the OPC bits into the most significant shadow bits of this new address. We are thus creating a new and unique address as every other existing physical address has its shadow bits set to zero. The address can also not match an existing migrated address: if two PA's are equal, the corresponding PAs must also be equal. If the original PA is swapped out of physical memory, the TLB entries for PA' are invalidated (more on TLB organization shortly); so it is not possible for the name PA' to represent two distinct pages that were both assigned to address PA in physical memory at different times.

TLB Modifications

To effect the name change, the TLBs of every core on chip must be updated (similar to the well-known process of TLB shootdown). Each TLB entry now has a new field that stores the NPC bits if that page has undergone migration. This is a relatively minor change to the TLB structure. Estimates with CACTI 6.0 [28] show that the addition of six bits to each entry of a 128-entry TLB does not affect access time and slightly increases its energy per access from 5.74 to 5.99 pJ (at 65 nm technology). It is therefore straightforward to produce the new address.

Off-Chip Access

If the request must travel off-chip, PA' must be converted back to PA. This process is trivial as it simply requires that the NPC bits in PA' be replaced by the OPC bits currently residing in shadow space and the shadow bits are all reset (see Figure 1). Thus, no table look-ups are required for this common case.

Translation Table (TT)

In addition to updating TLB entries, every page re-color must also be tracked in a separate structure (co-located with the L2 cache controller) referred to as the *Translation Table (TT)*. This structure is required in case a TLB entry is evicted, but the corresponding blocks still reside with their new name in L1 or L2. This structure keeps track of process-id, VPN, PPN, and NPC. It must be looked up on a TLB miss before looking up page tables. It must also be looked up when the processor receives a coherence request from off-chip as the off-chip name PA must be translated to the on-chip name PA'. This structure must be somewhat large as it has to keep track of every recent page migration that may still have blocks in cache. If an entry is evicted from this structure, it must invalidate any cached blocks for that entry and its instances in various TLBs.

Our simulations assume a fully-associative LRU structure with 10K entries and this leads to minimal evictions. We believe that set-associative implementations will also work well, although, we haven't yet focused on optimizing the design of the TT. Such a structure has a storage requirement of roughly 160KB, which may represent a minor overhead for today's billion-transistor architectures. The TT is admittedly the biggest overhead of the proposed mechanisms, but it is accessed relatively infrequently. In fact, it serves as a second-level large TLB and may be more efficient to access than walking through the page tables that may not be cache-resident; it may therefore be a structure worth considering even for a conventional processor design. The inefficiency of this structure will be a problem if the processor is inundated with external coherence requests (not a consideration in our simulations). One way to resolve this problem is to not move individual pages, but entire colored regions at a time, *i.e.*, all pages colored red are re-colored to yellow.

Cache Flushes

When a page is migrated within the processor, the TLB entries are updated and the existing dirty lines of that page in L2 cache must be flushed (written back). If the directory for that L2 cache line indicates that the most recent copy of that line is in an L1 cache, then that L1 entry must also be flushed. All non-dirty lines in L1 and L2 need not be explicitly flushed. They will never be accessed again as the old tags will never match a subsequent request and they will be naturally replaced by the LRU replacement policy. Thus, every page migration will result in a number of L1 and L2 cache misses that serve to re-load the page into its new locations in cache. Our results later show that these "compulsory" misses are not severe if the data is accessed frequently enough after its migration. These overheads can be further reduced if we maintain a small writeback buffer that can help re-load the data on subsequent reads before it is written back to memory. For our simulations, we pessimistically assume that every first read of a block after its page migration requires a re-load from memory. The L1 misses can be potentially avoided if the L1 caches continue to use the original address while the L2 cache uses the new address (note that page migration is being done to improve placement in the L2 and does not help L1 behavior in any

way). But this would lead to a situation where data blocks reside in L1, but do not necessarily have a back-up copy in L2, thus violating inclusivity. We do not consider this optimization here in order to retain strict inclusivity within the L1-L2 hierarchy.

Cache Tags and Indexing

Most cache tag structures do not store the most significant shadow bits that are always zero. In the proposed scheme, the tag structures are made larger as they must also accommodate the OPC bits for a migrated page. Our CACTI 6.0 estimates show that this results in a 5% increase in area/storage, a 2.64% increase in access time, and a 9.3% increase in energy per access for our 16 KB/4-way L1 cache at 65 nm technology (the impact is even lower on the L2 cache). We continue to index into the L1 cache with the virtual address, so the TLB look-up is not on the L1 critical path just as in the baseline. The color bits that we modify must therefore not be part of the L1 index bits (as shown at the top of Figure 1).

3.2 Managing Capacity Allocation and Sharing

In our study, we focus our evaluations on 4- and 8-core systems as shown in Figure 2. The L2 cache is shared by all the cores and located centrally on chip. The L2 cache capacity is assumed to be 2 MB for the 4-core case and 4 MB for the 8-core case. Our solutions also apply to a tiled architecture where a slice of the shared L2 cache is co-located with each core. The L2 cache is partitioned into 16 banks and connected to the cores with an on-chip network with a grid topology. The L2 cache is organized as a static-NUCA architecture. In our study, we use 64 colors for the 4-core case and 128 colors for the 8-core case.

When handling multi-programmed workloads, our proposed policy attempts to spread the working set of a single program across many colors if it has high capacity needs. Conversely, a program with low working-set needs is forced to share its colors with other programs. When handling a multi-threaded workload, our policies attempt to move a page closer to the center-of-gravity of its accesses, while being cognizant of cache capacity constraints. The policies need not be aware of whether the workload is multi-programmed or multi-threaded. Both sets of policies run simultaneously, trying to balance capacity allocations as well as proximity of computation to data. Each policy is discussed separately in the next two sub-sections.

3.2.1 Capacity Allocation Across Cores

Every time a core touches a page for the first time, the OS maps the page to some region in physical memory. We make no change to the OS' default memory management and alter the page number within the processor. Every core is assigned a set of colors that it can use for its pages and this is stored in a small hardware register. At start-up time, colors are equally distributed among all cores such that each core is assigned colors in close proximity. When a page is brought in for the first time, does not have an entry in the TT, and has an original page color (OPC) that is not in the assigned set of colors for that core, it is migrated to one of the assigned colors (in round-robin fashion). Every time a page re-coloring happens, it is tracked in the TT, every other TLB is informed, and the corresponding dirty blocks in L2 are flushed. The last step can be time-consuming as the tags of a number of sets in L2 must be examined, but this is not necessarily on the critical path. In our simulations, we assume that every page re-color is accompanied by a 200 cycle stall to perform the above operations. A core must also stall on every read to a cache line that was flushed. We confirmed that our results are not very sensitive to the 200 cycle stall penalty as it is incurred infrequently and mostly during the start of the application.

There are two key steps in allocating capacity across cores. The first is to determine the set of colors assigned to each core and the second is to move pages out of banks that happen to be heavily pressured. Both of these steps are performed periodically by the OS. Every 10 million cycle time interval is referred to as an *epoch* and at the end of every epoch, the OS executes a routine that examines various hardware counters. For each color, these hardware counters specify number of misses and *usage* (how many unique lines yield cache hits in that epoch). If a color has a high miss rate, it is deemed to be in need of more cache space and referred to as an *"Acceptor"*. If a color has low usage, it is deemed to be a *"Donor"*, *i.e.*, this color can be shared by more programs. Note that a color could suffer from high miss rate and low usage, which hints at a streaming workload, and the color is then deemed to be a Donor. For all cores that have an assigned color that is an Acceptor, we attempt to assign one more color to that core from the list of Donor colors. For each color i in the list of Donor colors, we compute the following cost function:

$$color\_suitability_i = \alpha_A \times distance_i + \alpha_B \times usage_i$$

α_A and α_B are weights that model the relative importance of usage and the distance between that color and the core in question. The weights were chosen such that the distance and usage quantities were roughly equal in magnitude in the common case. The color that minimizes the above cost function is taken out of the list of Donors and placed in the set of colors assigned to that core. At this point, that color is potentially shared by multiple cores. The OS routine then handles the next core. The order in which we examine cores is a function of the number of Acceptors in each core's set of colors and the miss rates within those Acceptors. This mechanism is referred to as PROPOSED-COLOR-STEAL in the results section.

If a given color is shared by multiple cores and its miss rate exceeds a high threshold for a series of epochs, it signals the fact that some potentially harmful re-coloring decisions have been made. At this point, one of the cores takes that color out of its assigned set and chooses to migrate a number of its pages elsewhere to another Donor color (computed with the same cost function above). The pages that are migrated are the ones in the TLB of that
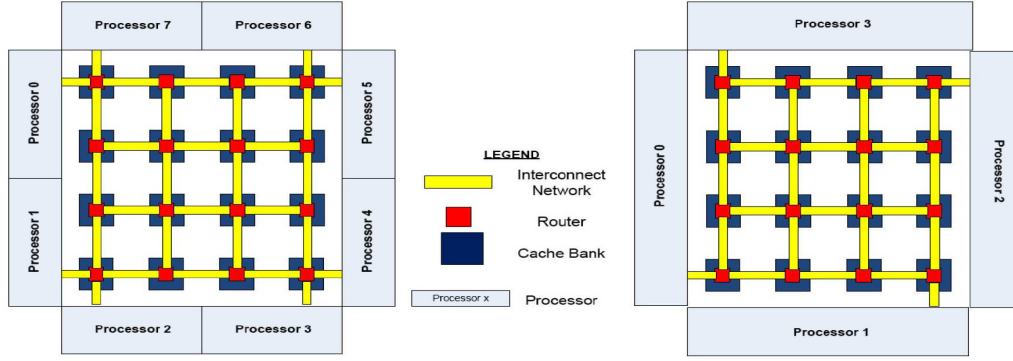
Figure 2. Arrangement of processors, NUCA cache banks, and the on-chip interconnect.

core with the offending color. This process is repeated for a series of epochs until that core has migrated most of its frequently used pages from the offending color to the new Donor color. With this policy included, the mechanism is referred to as PROPOSED-COLOR-STEAL-MIGRATE.

Minimal hardware overhead is introduced by the proposed policies. Each core requires a register to keep track of assigned colors. Cache banks require a few counters to track misses per color. Each L2 cache line requires a bit to indicate if the line is touched in this epoch and these bits must be counted at the end of the epoch (sampling could also be employed, although, we have not evaluated that approximation). The OS routine is executed once every epoch and will incur overheads of less than 1% even if it executes for as many as 100,000 cycles. An update of the color set for each core does not incur additional overheads, although, the migration of a core's pages to a new donor color will incur TLB shootdown and cache flush overheads. Fortunately, the latter is exercised infrequently in our simulations. Also note that while the OS routine is performing its operations, a core is stalled only if it makes a request to a page that is currently in the process of migrating[1].

3.2.2 Migration for Shared Pages

The previous sub-section describes a periodic OS routine that allocates cache capacity among cores. We adopt a similar approach to also move pages that are shared by the threads of a multi-threaded application. Based on the capacity heuristics just described, pages of a multi-threaded application are initially placed with a focus on minimizing miss rates. Over time, it may become clear that a page happens to be placed far from the cores that make the most frequent accesses to that page, thus yielding high average access times for L2 cache hits. As the access patterns for a page become clear, it is important to move the page to the *Center-of-Gravity (CoG)* of its requests in an attempt to minimize delays on the on-chip network.

Just as in the previous sub-section, an OS routine executes at the end of every epoch and examines various hardware counters. Hardware counters associated with every TLB entry keep track of the number of accesses made to

L1 I-cache	16KB 4-way 1-cycle
L1 D-cache	16KB 4-way 1-cycle
Page Size	4 KB
Memory latency	200 cycles for the first block
L2 unified cache	2MB (4-core) / 4MB (8-core) 8-way
DRAM Size	4 GB

NUCA Parameters			
Network	4×4 grid	Bank access time	3 cycles
Hop Access time	2 cycles	Router Overhead	3 cycles
(Vertical & Horizontal)			

Table 1. Simics simulator parameters.

that page by that core. The OS collects these statistics for the 10 most highly-accessed pages in each TLB. For each of these pages, we then compute the following cost function for each color i:

$$color\_suitability_i = \alpha_A \times total\_latency_i + \alpha_B \times usage_i$$

where $total\_latency_i$ is the total delay on the network experienced by all cores when accessing this page, assuming the frequency of accesses measured in the last epoch. The page is then moved to the color that minimizes the above cost function, thus attempting to reduce latency for this page and cache pressure in a balanced manner. Page migrations go through the same process as before and can be relatively time consuming as TLB entries are updated and dirty cache lines are flushed. A core's execution will be stalled if it attempts to access a page that is undergoing migration. For our workloads, page access frequencies are stable across epochs and the benefits of low-latency access over the entire application execution outweigh the high initial cost of moving a page to its optimal location.

This policy introduces hardware counters for each TLB entry in every core. Again, it may be possible to sample a fraction of all TLB entries and arrive at a better performance-cost design point. This paper focuses on evaluating the performance potential of the proposed policies and we leave such approximations for future work.

4. Results
4.1 Methodology

Our simulation infrastructure uses Virtutech's Simics platform [26]. We build our own cache and network models upon Simics' *g-cache* module. Table 1 summarizes the configuration of the simulated system. All delay calculations are for a 65 nm process and a clock frequency of

---

[1]This is indicated by a bit in the TLB. This bit is set at the start of the TLB shootdown process and reset at the very end of the migration.

Acceptor Applications	bzip2*, gobmk*, hmmer*, h264ref*, omnetpp*, xalancbmk*, soplex*, mummer•, tigr•, fasta-dna•
Donor Applications	namd*, libquantum*, sjeng*, milc*,povray*,swaptions*

Table 2. **Workload Characteristics.** * **- SpecCpu2006,** • **- BioBench,** * **- PARSEC**

5 GHz and a large 16 MB cache. The delay values are calculated using CACTI 6.0 [28] and remain the same irrespective of cache size being modeled. For all of our simulations, we shrink the cache size (while retaining the same bank and network delays), because our simulated workloads are being shrunk (in terms of number of cores and input size) to accommodate slow simulation speeds. Ordinarily, a 16 MB L2 cache would support numerous cores, but we restrict ourselves to 4 and 8 core simulations and shrink the cache size to offer 512 KB per core (more L2 capacity per core than many modern commercial designs). The cache and core layouts for the 4 and 8 core CMP systems are shown in Figure 2. Most of our results focus on the 4-core system and we show the most salient results for the 8-core system as a sensitivity analysis. The NUCA L2 is arranged as a 4x4 grid with a bank access time of 3 cycles and a network hop (link plus router) delay of five cycles. We accurately model network and bank access contention. An epoch length of 10 million instructions is employed.

Our simulations have a warm-up period of 25 million instructions. The capacity allocation policies described in Section 3.2.1 are tested on multi-programmed workloads from SPEC, BioBench, and PARSEC [4], described in Table 2. As described shortly, these specific programs were selected to have a good mix of small and large working sets. SPEC and BioBench programs are fast forwarded by 2 billion instructions to get past the initialization phase while the PARSEC programs are observed over their defined *regions of interest*. After warm-up, the workloads are run until each core executes for two billion instructions.

The shared-page migration policies described in Section 3.2.2 are tested on multi-threaded benchmarks from SPLASH-2 [41] and PARSEC described in Table 5. All these applications were fast forwarded to the beginning of parallel section or the region of interest (for SPLASH-2 and PARSEC, respectively) and then executed for 25 million instructions to warm up the caches. Results were collected over the next 1 billion instruction execution, or, end of parallel section/region-of-interest, whichever occurs first.

Just as we use the terms *Acceptors* and *Donors* for colors in Section 3.2.1, we also similarly dub programs depending on whether they benefit from caches larger than 512 KB. Figure 3(a) shows IPC results for a subset of programs from the benchmark suites, as we provide them with varying sizes of L2 cache while keeping the L2 (UCA) access time fixed at 15 cycles. This experiment gives us a good idea about capacity requirements of various applications and the 10 applications on the left of Figure 3(a) are termed *Acceptors* and the other 6 are termed *Donors*.

4.2 Baseline Configurations
We employ the following baseline configurations to understand the roles played by capacity, access times, and

Application	Pages Mapped to Stolen Colors	Total Pages Touched
bzip2	200	3140
gobmk	256	4010
hmmer	124	2315
h264ref	189	2272
omnetpp	376	8529
xalancbmk	300	6751
soplex	552	9632
mimmer	9073	29261
tigr	6930	17820
fasta-dna	740	1634

Table 3. **Behavior of PROPOSED-COLOR-STEAL.**

data mapping in S-NUCA caches:

1. **BASE-UCA**: Even though the benefits of NUCA are well understood, we provide results for a 2 MB UCA baseline as well for reference. Similar to our NUCA estimates, the UCA delay of 15 cycles is based on CACTI estimates for a 16 MB cache.

2. **BASE-SNUCA**: This baseline does not employ any intelligent assignment of colors to pages (they are effectively random). Each color maps to a unique bank (the least significant color bits identify the bank). The data accessed by a core in this baseline are somewhat uniformly distributed across all banks.

3. **BASE-PRIVATE**: All pages are colored once on first touch and placed in one of the four banks (in round-robin order) closest to the core touching this data. As a result, each of the four cores is statically assigned a quarter of the 2 MB cache space (resembling a baseline that offers a collection of private caches). This baseline does not allow spilling data into other colors even if some color is heavily pressured.

The behavior of these baselines, when handling a single program, is contrasted by the three left bars in Figure 3(b). This figure only shows results for the Acceptor applications. The UCA cache is clearly the most inferior across the board. Only two applications (gobmk, hmmer) show better performance with BASE-PRIVATE than BASE-SHARED – even though these programs have large working sets, they benefit more from having data placed nearby than from having larger capacity. This is also of course trivially true for all the Donor applications (not shown in figure).

4.3 Multi-Programmed Results
Before diving into the multi-programmed results, we first highlight the behavior of our proposed mechanisms when executing a single program, while the other three cores remain idle. This is demonstrated by the rightmost bar in Figure 3(b). The proposed mechanisms (refered to as PROPOSED-COLOR-STEAL) initially color pages to place them in the four banks around the requesting core. Over time, as bank pressure builds, the OS routine alters the set of colors assigned to each core, allowing the core to steal colors (capacity) from nearby banks. Since these are

	4 Cores
2 Acceptors	{gobmk, tigr, libquantum, namd}^{M1}, {mummer, bzip2, milc, povray}^{M2}, {mummer, mummer, milc, libquantum}^{M3}, {mummer, omnetpp, swaptions, swaptions}^{M4}, {soplex, hmmer, sjeng, milc}^{M5}, {soplex, h264ref, swaptions, swaptions}^{M6} {bzip2, soplex, swaptions, povray}^{M7}, {fasta-dna, hmmer, swaptions, libquantum}^{M8}, {hmmer, omnetpp, swaptions, milc}^{M9}, {xalancbmk, hmmer, namd, swaptions}^{M10}, {tigr, hmmer, povray, libquantum}^{M11}, {tigr, mummer, milc, namd}^{M12}, {tigr, tigr, povray, sjeng}^{M13}, {xalancbmk, h264ref, milc, sjeng}^{M14},
3 Acceptors	{h264ref, xalancbmk, hmmer, sjeng}^{M15}, {mummer, bzip2, gobmk, milc}^{M16}, {fasta-dna, tigr, mummer, namd}^{M17}, {omnetpp, xalancbmk, fasta-dna, povray}^{M18}, {gobmk, soplex, tigr, swaptions}^{M19}, {bzip2, omnetpp, soplex, libquantum}^{M20}
4 Acceptors	{bzip2, soplex, xalancbmk, omnetpp}^{M21}, {fasta-dna, mummer, mummer, soplex}^{M22}, {gobmk, soplex, xalancbmk, h264ref}^{M23}, {soplex, h264ref, mummer, omnetpp}^{M24}, {bzip2, tigr, xalancbmk, mummer}^{M25}

	8 -cores
4 Acceptors	{mummer,hmmer,bzip2,xalancbmk,swaptions,namd,sjeng,povray}^{M26}, {omnetpp,h264ref,tigr,soplex,libquantum,milc, swaptions,namd}^{M27}
6 Acceptors	{h264ref,bzip2,tigr,omnetpp,fasta-dna,soplex,swaptions,namd}^{M28} {mummer,tigr,fasta-dna,gobmk,hmmer,bzip2,milc,namd}^{M29}
8 Acceptors	{bzip2, gobmk,hmmer,h264ref,omnetpp,soplex,mummer,tigr}^{M30} {fasta-dna,mummer,h264ref,soplex,bzip2,omnetpp,bzip2,gobmk}^{M31}

Table 4. Workload Mixes - 4 and 8 cores. Each workload will be referred to by its superscript name.

single-program results, the program does not experience competition for space in any of the banks. The proposed mechanisms show a clear improvement over all baselines (an average improvement of 15% over BASE-SNUCA and 21% over BASE-PRIVATE). They not only provide high data locality by placing most initial (and possibly most critical) data in nearby banks, but also allow selective spillage into nearby banks as pressure builds. Our statistics show that compared to BASE-PRIVATE, the miss rate reduces by an average of 15.8%. The number of pages mapped to stolen colors is summarized in Table 3. Not surprisingly, the applications that benefit most are the ones that touch (and spill) a large number of pages.

4.3.1 Multicore Workloads

We next present our simulation models that execute four programs on the four cores. A number of workload mixes are constructed (described in Table 4). We vary the number of acceptors to evaluate the effect of greater competition for limited cache space. In all workloads, we attempted to maintain a good mix of applications not only from different suites, but also with different runtime behaviors. For all experiments, the epoch lengths are assumed to be 10 million instructions for PROPOSED-COLOR-STEAL. Decision to migrate already recolored pages (PROPOSED-COLOR-STEAL-MIGRATE) are made every 50 million cycles. Having smaller epoch lengths results in frequent movement of recolored pages.

The same cache organizations as described before are compared again; there is simply more competition for the space from multiple programs. To demonstrate the impact of migrating pages away from over-subscribed colors, we show results for two versions of our proposed mechanism. The first (PROPOSED-COLOR-STEAL) never migrates pages once they have been assigned an initial color; the second (PROPOSED-COLOR-STEAL-MIGRATE) reacts to poor initial decisions by migrating pages. The PROPOSED-COLOR-STEAL policy, to some extent, approximates the behavior of policies proposed by Cho and

Jin [11]. Note that there are several other differences between our approach and theirs, most notably, the mechanism by which a page is re-colored within the hardware.

To determine the effectiveness of our policies, we use weighted system throughputs as the metric. This is computed as follows:

$$weighted\_throughput = \sum_{i=0}^{NUM\_CORES-1} \{IPC_i/IPC_{i\_BASE\_PRIVATE}\}$$

Here, IPC_i refers to the application IPC for that experiment and $IPC_{i\_BASE\_PRIVATE}$ refers to the IPC of that application when it is assigned a quarter of the cache space as in the BASE-PRIVATE case. The weighted throughput of the BASE-PRIVATE model will therefore be very close to 4.0 for the 4-core system.

The results in Figures 4 and 5 are organized based on the number of acceptor programs in the workload mix. For 2, 3, and 4 acceptor cases, the maximum/average improvements in weighted throughput with the PROPOSED-COLOR-STEAL-MIGRATE policy, compared to the best baseline (BASE-SNUCA) are 25%/20%, 16%/13%, and 14%/10%, respectively. With only the PROPOSED-COLOR-STEAL policy, the corresponding improvements are 21%/14%, 14%/10%, and 10%/6%. This demonstrates the importance of being able to adapt to changes in working set behaviors and inaccuracies in initial page coloring decisions. This is especially important for real systems where programs terminate/sleep and are replaced by other programs with potentially different working set needs. The ability to seamlessly move pages with little overhead with our proposed mechanisms is important in these real-world settings, an artifact that is hard to measure for simulator studies. For the 1, 2, 3, and 4-acceptor cases, an average 18% reduction in cache miss rates and 21% reduction in average access times were observed.

Not surprisingly, improvements are lower as the number of acceptor applications increases because of higher competition for available colors. Even for the 4-acceptor case,
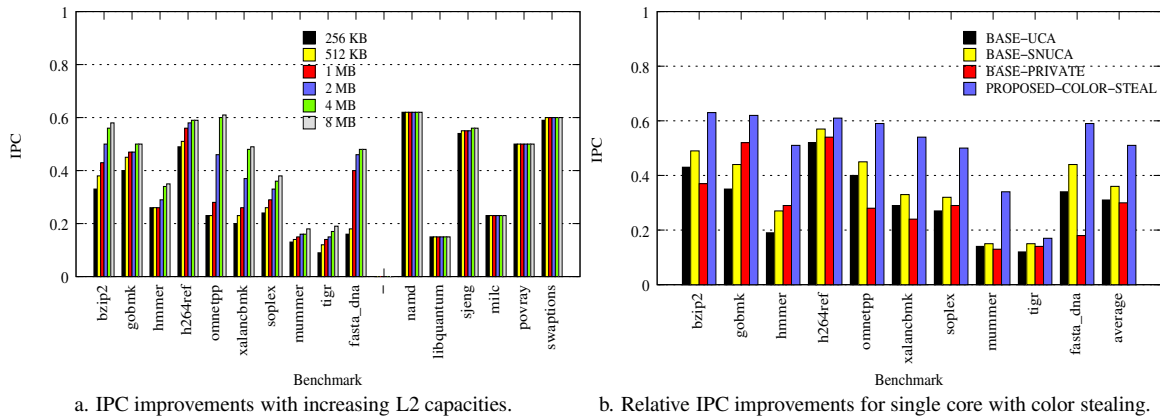
a. IPC improvements with increasing L2 capacities.

b. Relative IPC improvements for single core with color stealing.

Figure 3. **(a) Experiments to determine workloads (b) Relative IPC improvements of proposed color stealing approach.**
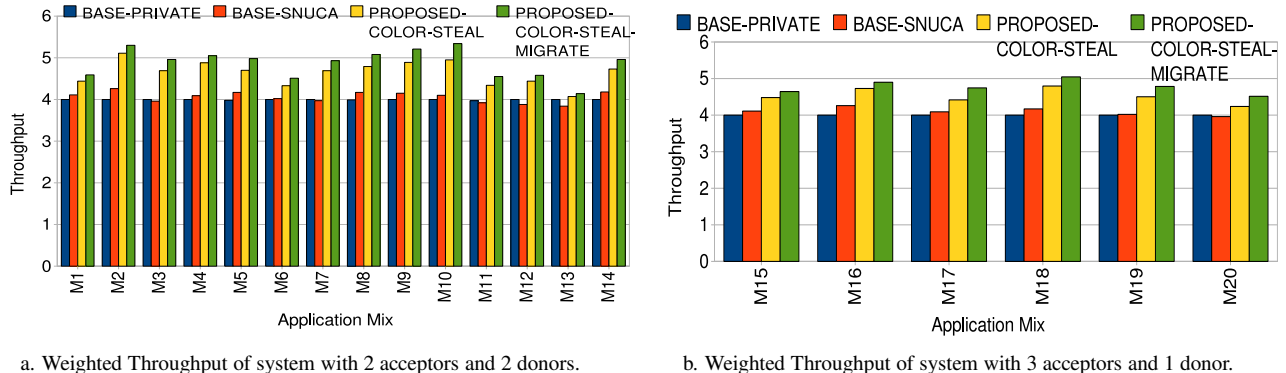

a. Weighted Throughput of system with 2 acceptors and 2 donors.

b. Weighted Throughput of system with 3 acceptors and 1 donor.

Figure 4. **System throughputs. Results for workloads with 2 acceptors are shown in (a) and with 3 acceptors in (b).**

non-trivial improvements are seen over the static baselines because colors are adaptively assigned to applications to balance out miss rates for each color. A maximum slow-down of 4% was observed for any of the donor applications, while much higher improvements are observed for many of the co-scheduled acceptor applications.

As a sensitivity analysis, we show a limited set of experiments for the 8-core system. Figure 5(b) shows the behavior of the two baselines and the two proposed mechanisms for a few 4-acceptor, 6-acceptor, and 8-acceptor workloads. The average improvements with PROPOSED-COLOR-STEAL and PROPOSED-COLOR-STEAL-MIGRATE are 8.8% and 12%, respectively.

4.4 Results for Multi-threaded workloads

In this section, we evaluate the page migration policies described in Section 3.2.2. We implement a MESI directory-based coherence protocol at the L1-L2 boundary with a writeback L2. The benchmarks and their properties are summarized in Table 5. We restrict most of our analysis to the 4 SPLASH-2 and 2 PARSEC programs in Table 5 that have a large percentage of pages that are frequently accessed by multiple cores. Not surprisingly, the other applications do not benefit much from intelligent migration of shared pages and are not discussed in the rest of the paper.

Since all these benchmarks must be executed with smaller working sets to allow for acceptable simulation times (for example, PARSEC programs can only be sim-

ulated with *large* input sets, not the *native* input sets), we must correspondingly also model a smaller cache size [41]. If this is not done, there is almost no cache capacity pressure and it is difficult to test if page migration is not negatively impacting pressure in some cache banks. Preserving the NUCA access times in Table 1, we shrink the total L2 cache size to 64 KB. Correspondingly, we use a scaled down page size of 512B. The L1 caches are 2-way 4KB.

We present results below, first for a 4-core CMP, and finally for an 8-core CMP as a sensitivity analysis. We experimented with two schemes for migrating shared pages. Proposed-CoG migrates pages to their CoG, without regard for the destination bank pressure. Proposed-CoG-Pressure, on the other hand, also incorporates bank pressure into the cost metric while deciding the destination bank. We also evaluate two other schemes to compare our results. First, we implemented an Oracle placement scheme which directly places the pages at their CoG (with and without consideration for bank pressure - called Oracle-CoG and Oracle-CoG-Pressure, respectively). These optimal locations are determined based on a previous identical simulation of the baseline case. Second, we shrink the page size to merely 64 bytes. Such a migration policy attempts to mimic the state-of-the-art in D-NUCA fine-grain migration policies that move a single block at a time to its CoG. Comparison against this baseline gives us confidence that we are not severely degrading performance by performing
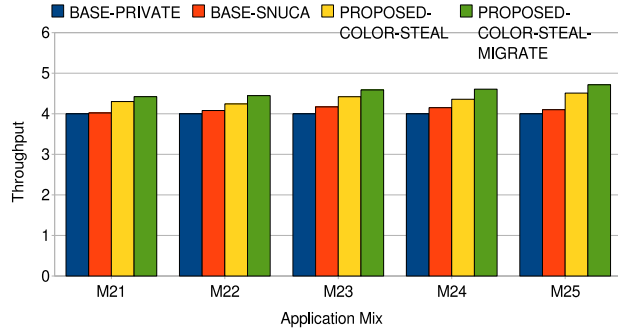
a. Weighted Throughput of system with 4 Cores and 4 Acceptors.
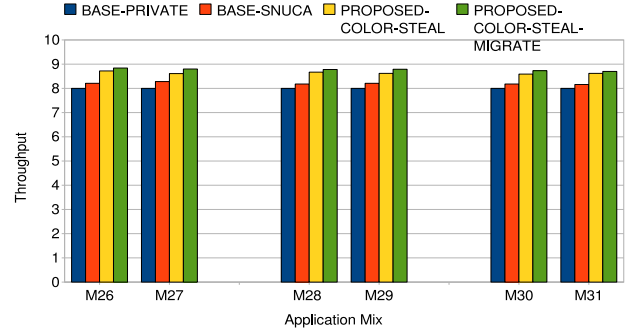
b. Weighted throughputs for 8 core workloads.

Figure 5. **Normalized system throughputs as compared to BASE-PRIVATE. Results for workloads with 4 Cores/4 acceptors are shown in (a) and all 8 Core mixes in (b).**

Application	Percentage of RW-shared pages	Application	Percentage of RW-shared pages
fft(ref)	62.4%	water-nsq(ref)	22%
cholesky(ref)	30.6%	water-spa(ref)	22.2%
fmm(ref)	31%	blackscholes(simlarge)	24.5%
barnes(ref)	67.7%	freqmine(simlarge)	16%
lu-nonc(ref)	61%	bodytrack(simlarge)	19.7%
lu-cont(ref)	62%	swaptions(simlarge)	20%
ocean-cont(ref)	50.3%	streamcluster(simlarge)	10.5%
ocean-nonc(ref)	67.2%	x264(simlarge)	30%
radix(ref)	40.5%		

Table 5. **SPLASH-2 and PARSEC programs with their inputs and percentage of RW-shared pages.**
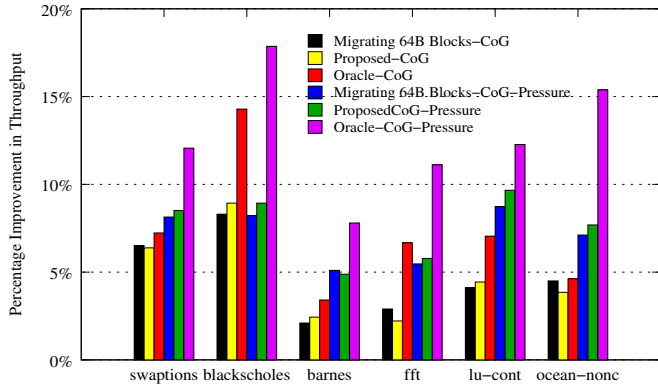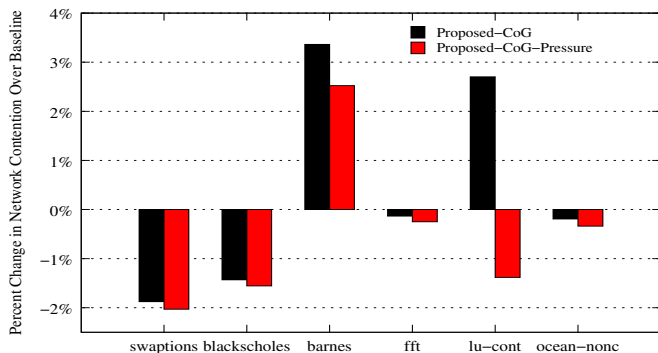

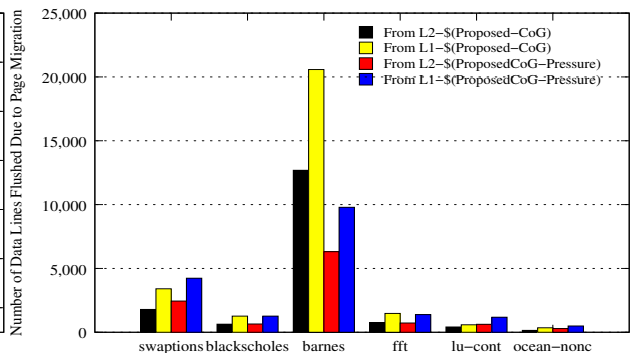a. Percentage improvement in throughput.
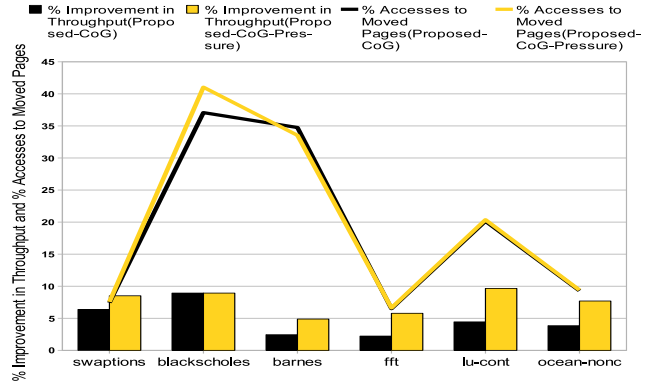
b. Improvement in throughput and percentage accesses to moved pages

Figure 6. **(a). Percentage improvement in throughput (b). Percentage improvement in throughput overlaid with percentage accesses to moved pages**

a. Percentage change in network contention due to proposed schemes.

b. Number of cache lines flushed due to migration of RW-shared pages.

Figure 7. **(a). Network contention behavior. (b). Cache flushes.**

10

migrations at the coarse granularity of a page. The baseline in all these experiments is BASE-SNUCA.

Figure 6(a) presents the percentage improvement in throughput for the six models, relative to the baseline. The Proposed-CoG-Pressure model outperforms the Proposed-CoG model by 3.1% on average and demonstrates the importance of taking bank pressure into account during migration. This feature was notably absent from prior D-NUCA policies (and is admittedly less important if capacity pressures are non-existent). By taking bank pressure into account, the number of L2 misses is reduced by 5.31% on average, relative to Proposed-CoG. The proposed models also perform within 2.5% and 5.2%, on average, of the corresponding oracle scheme. It is difficult to bridge this gap because the simulations take fairly long to determine the optimal location and react. This gap will naturally shrink if the simulations are allowed to execute much longer and amortize the initial inefficiencies. Our policies are within 1% on average to the model that migrates 64B pages. While a larger page size may make sub-optimal CoG decisions for each block, it does help prefetch a number of data blocks into a close-to-optimal location.

To interpret the performance improvements, we plot the percentage of requests arriving at L2 for data in migrated pages. Figure 6(b) overlays this percentage with percentage improvement in throughput. The Y-axis represents percentage improvement in throughput and percentage of accesses to moved pages. The curves plot accesses to moved pages and the bars show improvement in throughput. As can be seen, the curves closely track the improvements as expected, except for *barnes*. This is clarified by Figure 7(a) that shows that moving pages towards central banks can lead to higher network contention in some cases (*barnes*), and slightly negate the performance improvements. By reducing capacity pressures on a bank, the Proposed-CoG-Pressure also has the side effect of lowering network contention. At the outset, it might appear that migrating pages to central locations may increase network contention. In most cases however, network contention is lowered as network messages have to travel shorter distances on average, thus reducing network utilization.

Figure 7(b) plots the number of lines flushed due to migration decisions. The amount of data flushed is relatively small for nearly 1 billion or longer instruction executions. *barnes* again is an outlier with highest amount of data flushed. This also contributes to its lower performance improvement. The reason for this high amount of data flush is that the sharing pattern exhibited by *barnes* is not uniform. The accesses by cores to RW-shared data keeps varying (due to possibly variable producer-consumer relationship) among executing threads. This leads to continuous corrections in CoG which further leads to large amount of data flushes.

As a sensitivity analysis of our scheme, for an 8-core CMP we only present percentage improvement in throughput in Figure 8. The proposed policies show an average
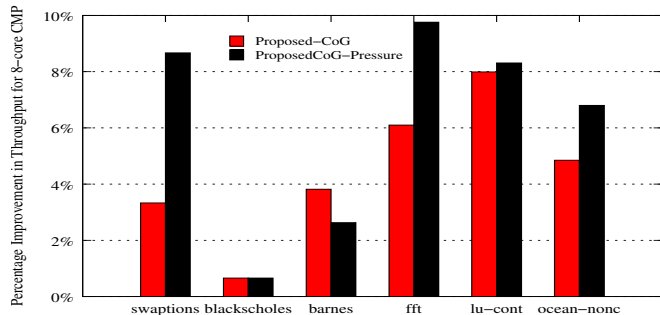

Figure 8. **Throughput improvement for 8-core CMP.**

improvement of 6.4%.

5. Conclusions

In this paper, we attempt to combine the desirable features of a number of state-of-the-art proposals in large cache design. We show that hardware mechanisms based on shadow address bits are effective in migrating pages within the processor at low cost. This allows us to design policies to allocate cache space among competing threads and migrate shared pages to optimal locations. The resulting architecture allows for high cache hit rates, low cache access latencies on average, and yields overall improvements of 10-20% with capacity allocation, and 8% with shared page migration. The design also entails low complexity for the most part, for example, by eliminating complex search mechanisms that are commonly seen in way-partitioned NUCA designs. The primary complexity introduced by the proposed scheme is the Translation Table (TT) and its management. Addressing this problem is important future work. We also plan to leverage the page coloring techniques proposed here to design mechanisms for page replication while being cognizant of bank pressures.

References

[1] S. Akioka, F. Li, M. Kandemir, and M. J. Irwin. Ring Prediction for Non-Uniform Cache Architectures . In *Proceedings of PACT-2007*, September 2007.

[2] B. Beckmann, M. Marty, and D. Wood. ASR: Adaptive Selective Replication for CMP Caches. In *Proceedings of MICRO-39*, December 2006.

[3] B. Beckmann and D. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *Proceedings of MICRO-37*, December 2004.

[4] C. Benia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical report, 2008.

[5] J. Carter, W. Hsieh, L. Stroller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a Smarter Memory Controller. In *Proceedings of HPCA-5*, January 1999.

[6] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of ASPLOS-VI*, 1994.

[7] J. Chang and G. Sohi. Co-Operative Caching for Chip Multiprocessors. In *Proceedings of ISCA-33*, June 2006.

[8] M. Chaudhuri. PageNUCA: Selected Policies for Page-grain Locality Management in Large Shared Chip-Multiprocessor Caches. In *Proceedings of HPCA-15*, 2009.

[9] Z. Chishti, M. Powell, and T. Vijaykumar. Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures. In *Proceedings of MICRO-36*, December 2003.

[10] Z. Chishti, M. Powell, and T. Vijaykumar. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *Proceedings of ISCA-32*, June 2005.

[11] S. Cho and L. Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *Proceedings of MICRO-39*, December 2006.

[12] J. Corbalan, X. Martorell, and J. Labarta. Page Migration with Dynamic Space-Sharing Scheduling Policies: The case of SGIO200. *International Journal of Parallel Programming*, 32(4), 2004.

[13] X. Ding, D. S. Nikopoulosi, S. Jiang, and X. Zhang. MESA: Reducing Cache Conflicts by Integrating Static and Run-Time Methods . In *Proceedings of ISPASS-2006*, 2006.

[14] H. Dybdahl and P. Stenstrom. An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In *Proceedings of HPCA-13*, February 2007.

[15] B. Falsafi and D. Wood. Reactive NUMA: A Design for Unifying S-COMA and cc-NUMA. In *Proceedings of ISCA-24*, 1997.

[16] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of HPCA*, 1999.

[17] T. Horel and G. Lauterbach. UltraSPARC III: Designing Third Generation 64-Bit Performance. *IEEE Micro*, 19(3), May/June 1999.

[18] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. Keckler. A NUCA Substrate for Flexible CMP Cache Sharing. In *Proceedings of ICS-19*, June 2005.

[19] R. Iyer, L. Zhao, F. Guo, R. Illikkal, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *Proceedings of SIGMETRICS*, June 2007.

[20] Y. Jin, E. J. Kim, and K. H. Yum. A Domain-Specific On-Chip Network Design for Large Scale Cache Systems. In *Proceedings of HPCA-13*, February 2007.

[21] R. E. Kessler and M. D. Hill. Page Placement Algorithms for Large Real-Indexed Caches. *ACM Trans. Comput. Syst.*, 10(4):338–359, 1992.

[22] C. Kim, D. Burger, and S. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Dominated On-Chip Caches. In *Proceedings of ASPLOS-X*, October 2002.

[23] P. Kundu. On-Die Interconnects for Next Generation CMPs. In *Workshop on On- and Off-Chip Interconnection Networks for Multicore Systems (OCIN)*, December 2006.

[24] P. R. LaRowe and S. C. Ellis. Experimental comparison of memory management policies for numa multiprocessors. Technical report, Durham, NC, USA, 1990.

[25] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *Proceedings of HPCA-14*, February 2008.

[26] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[27] R. Min and Y. Hu. Improving Performance of Large Physically Indexed Caches by Decoupling Memory Addresses from Cache Addresses. *IEEE Trans. Comput.*, 50(11):1191–1201, 2001.

[28] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of the 40th International Symposium on Microarchitecture (MICRO-40)*, December 2007.

[29] M. Qureshi and Y. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of MICRO-39*, December 2006.

[30] N. Rafique, W. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *Proceedings of PACT-2006*, September 2006.

[31] J. Richard P. LaRowe and C. S. Ellis. Page placement policies for numa multiprocessors. *J. Parallel Distrib. Comput.*, 11(2):112–129, 1991.

[32] J. Richard P. LaRowe, J. T. Wilkes, and C. S. Ellis. Exploiting operating system support for dynamic page placement on a numa shared memory multiprocessor. In *Proceedings of PPOPP*, 1991.

[33] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An Argument for Simple COMA. In *Proceedings of HPCA*, 1995.

[34] E. Speight, H. Shafi, L. Zhang, and R. Rajamony. Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors. In *Proceedings of ISCA-32*, 2005.

[35] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *J. Supercomput.*, 28(1):7–26, 2004.

[36] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing L2 Caches in Multicore Systems. In *Proceedings of CMPMSI*, June 2007.

[37] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In *Proceedings of ISSCC*, February 2007.

[38] K. Varadarajan, S. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, S. Makineni, and D. Newell. Molecular Caches: A Caching Structure for Dynamic Creation of Application-Specific Heterogeneous Cache Regions. In *Proceedings of MICRO-39*, December 2006.

[39] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. *SIGPLAN Not.*, 31(9):279–289, 1996.

[40] H. S. Wang, L. S. Peh, and S. Malik. A Power Model for Routers: Modeling Alpha 21364 and InfiniBand Routers. In *IEEE Micro, Vol 24, No 1*, January 2003.

[41] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of ISCA-22*, pages 24–36, June 1995.

[42] M. Zhang and K. Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of ISCA-32*, June 2005.