PageNUCA: Selected Policies for Page-grain Locality Management in Large Shared Chip-multiprocessor Caches

Mainak Chaudhuri
Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur 208016
INDIA
mainakc@cse.iitk.ac.in

Abstract

As the last-level on-chip caches in chip-multiprocessors increase in size, the physical locality of on-chip data becomes important for delivering high performance. The non-uniform access latency seen by a core to different independent banks of a large cache spread over the chip necessitates active mechanisms for improving data locality. The central proposal of this paper is a fully hardwired coarse-grain data migration mechanism that dynamically monitors the access patterns of the cores at the granularity of a page to reduce the book-keeping overhead and decides when and where to migrate an entire page of data to amortize the performance overhead. The page-grain migration mechanism is compared against two variants of previously proposed cache block-grain dynamic migration mechanisms and two OS-assisted static locality management mechanisms. Our detailed execution-driven simulation of an eight-core chip-multiprocessor with a shared 16 MB L2 cache employing a bidirectional ring to connect the cores and the L2 cache banks shows that hardwired dynamic page migration, while using only 4.8% of extra storage out of the total L2 cache and book-keeping budget, delivers the best performance and energy-efficiency across a set of shared memory parallel applications selected from the SPLASH-2, SPEC OMP, DARPA DIS, and FFTW suites and multiprogrammed workloads prepared out of the SPEC 2000 and BioBench suites. It reduces execution time by 18.7% and 12.6% on average (geometric mean) respectively for the shared memory applications and the multiprogrammed workloads compared to a baseline architecture that distributes the pages round-robin across the L2 cache banks.

1. Introduction

In the recent years, chip-multiprocessors (CMPs) have become the central focus of the chip industry. As the number of processors or cores on a single chip increases, the amount of on-die cache is also expected to increase. In one of the popular CMP cache hierarchy designs, the entire last level is dynamically shared among the cores while each core has its private lowest level of cache hierarchy.[1] Figure 1 shows an architecture with eight cores and two levels of cache hierarchy, the last level being divided into multiple small banks and shared across all the cores. Each L2 cache bank has its own portion of tag and data arrays. The bank holding a block B is designated as the home bank of B. While dynamic sharing of the last-level cache is increasingly becoming popular due to better utilization of the cache space, it introduces an important performance problem related to the non-uniform nature of the L2 cache access latency. An L2 cache access from a core to a nearby bank takes less time compared to an access to a bank far away. While offering fully private cache hierarchy to each core solves this problem, private hierarchies become less attractive due to wastage of cache space resulting from replication of shared blocks unless active mechanisms are in place to avoid this [9, 29].

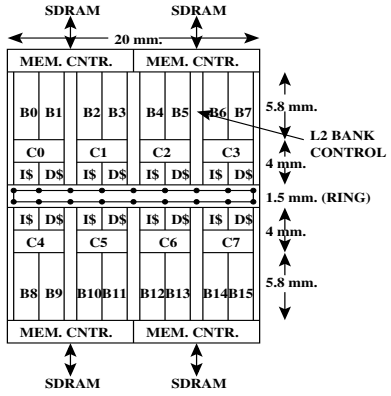

Figure 1. An example floorplan of an 8-core CMP with a shared L2 cache and private per-core L1 caches. Cn is the n^{th} core and Bn is the n^{th} L2 cache bank. The switches in the bidirectional ring are shown with solid dots. Each switch connects to either the L1 cache controllers of a core or the controllers of a pair of adjacent L2 cache banks, except at the corners where it connects to one L2 cache bank only. The dimensions are explained in Section 5.

In this paper, we squarely focus on the problem of on-chip data locality in large shared non-uniform cache architecture (NUCA) [19] designs resembling the one shown in Figure 1 and present the first proposal on coarse-grain dynamic migration as well as operating system (OS)-assisted static mechanisms to manage locality at a granularity of pages. The primary motivation to explore page-grain policies stems from the fact that learning dynamic reference patterns at page granularity requires less state and storage compared to the already proposed block-grain policies [5, 6, 12]. Also, page-grain mechanisms can amortize

[1] There may be intermediate levels of the hierarchy that are shared among a subset (possibly singleton) of cores.

the start-up overhead by migrating several cache blocks at a time and can pipeline the physical block transfers. While any coarse-grain data migration technique transferring more than one block at a time would possess these two nice properties, page-grain migration is particularly attractive due to the fact that physical addresses are assigned at page granularity. As a result, the maximally sized portion of a contiguous virtual address region that is guaranteed to appear contiguously in the physical address space is a page. Therefore, a page of data has a fairly high chance of exhibiting an access pattern that is homogeneous over the entire page. However, multiple contiguous physical pages may not be homogeneous in nature because they may correspond to completely unrelated regions in the virtual address space.

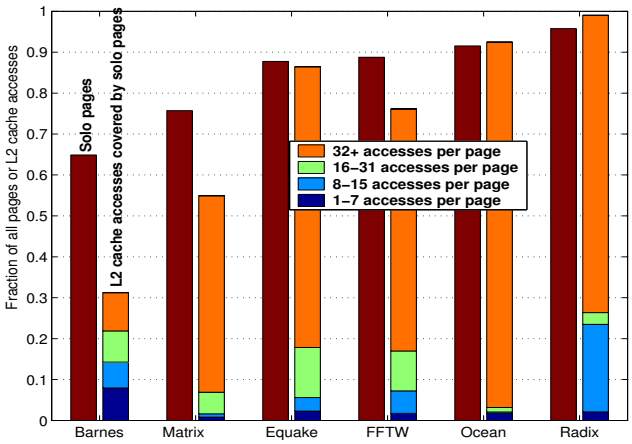

Figure 2. Page-grain (4 KB) solo access statistics.

To establish the key hypothesis of this work that there is significant access locality at the granularity of pages, Figure 2 presents some relevant data for a selected set of explicitly parallel shared memory applications running on an 8-core CMP with a 16 MB L2 cache similar to the one shown in Figure 1.[2] For each application, the bar on the left shows the fraction of pages having accesses from exactly one core during a 0.1 million-cycle sample period averaged across all such non-intersecting samples. The bar on the right shows the fraction of L2 cache accesses covered by these "solo pages". This fraction is further divided into four categories where each category shows the contribution coming from the solo pages that have number of accesses falling in that category. The statistics are very encouraging. For five of the six applications, more than 75% of pages accessed in a sample period are solo pages and these pages cover more than 50% of L2 cache accesses. Since a major proportion of L2 cache accesses is covered by solo pages with 32 or more accesses, a migration algorithm that migrates an entire page to a local bank of a core after observing the access pattern to a part of the page will be able to compensate the migration overhead by enjoying local reuse to the migrated page.

Motivated by these statistics we explore fully hardwired dynamic page migration (Section 3) as well as OS-assisted static page placement mechanisms (Section 4) to improve physical locality of data in large shared CMP NUCAs. Our simulation results (Sections 5 and 6) on a small-scale 8-core CMP with a shared 16 MB L2 cache employing a bidirectional ring interconnect show that fully hardwired dynamic page migration reduces execution time by up to 26.9% and on average (geometric mean) 18.7% for a set of shared memory parallel applications. For a set of representative multiprogrammed workloads, these numbers are 16.4% and 12.6%, respectively. This excellent performance of the page migration scheme comes at the cost of an extra storage overhead of only 4.8% to learn the reference patterns and maintain the migrated page addresses. Interestingly, we show that the storage overhead of a performance-optimized block migration scheme, which delivers performance reasonably close to page migration for most of the workloads, is as high as 28.5%. We also derive a simple analytical model (Section 7) to argue about the performance trends in the presence of data migration.

[2] The details of the simulation environment and the applications will be presented in Section 5.

1.1. Related Work

The impact of wire delay on large caches was presented in [19]. Instead of designing the entire cache in a monolithic way where the access time is governed by the worst case delay, the authors outline a non-uniform cache architecture (NUCA) that can take advantage of proximity of data from the accessing processor. Subsequently, the NuRAPID proposal decoupled the tag and data placement in a NUCA by augmenting each tag and data block with a forward and reverse pointer to the corresponding data block and tag, respectively [11].

Cache block migration in a CMP NUCA was first explored in [5]. However, the two-phase multicast tag search required to locate a block in the NUCA along with the gradual migration protocol used in that proposal may significantly increase the amount of dynamic energy expended per migration. CMP-NuRAPID [12] resolves the tag search issue by leveraging the forward and reverse pointers of NuRAPID. However, it replaces the single L2 cache tag array of NuRAPID by per-core private L2 cache tag array. This introduces two new problems, namely, the significant area overhead due to replication of the entire L2 cache tag array in each core and the hardware needed to keep the private tag arrays coherent. In contrast, our page migration proposal does not require per-core L2 cache tag arrays and offers solutions to the locality problem with a significantly lower area overhead. Also, our proposal, instead of using gradual migration, swaps the source and destination data of migration to minimize the number of copy operations. We implement a suitable block migration scheme to fit our architecture and show that these schemes impose significantly higher storage overhead than the proposed page migration scheme. In this paper, we do not implement data replication (as in [6, 32]) and propose other solutions to improve the locality of shared data without affecting the cache coherence protocol.

Various static and dynamic block-to-bank mapping algorithms to vary the sharing degree of a shared NUCA are explored in [17]. The authors find dynamic block-grain migration useful for improving performance. However, the dynamic schemes require a distributed partial tag search to locate a block in the cache. The authors in [13] evaluate a round-robin placement of demand pages on cache slices or banks and introduce the concept of virtual multicore where a round-robin page placement can be done by the OS within a group of cores constituting the virtual multicore. Concurrent to our effort, a dynamic page migration mechanism to improve capacity allocation and physical data locality in last-level caches is proposed in [3]. This work extends the OS-assisted page coloring technique proposed in [13] with hardware mechanisms to re-color pages. Since a migration only changes the color of a page and does not copy the page from the old location to the new location within the last-level cache, the first access to a block belonging to a migrated page suffers from a miss in the entire cache hierarchy. However, this proposal shows how to cleverly use the unused shadow bits in the upper part of the physical address to save a portion of the book-keeping storage, which our proposal needs for maintaining one of the address map tables.

Our proposal is most influenced by the page migration and replication studies on traditional distributed shared memory mul-

tiprocessors [8, 31]. These proposals looked at OS-assisted mechanisms to carry out dynamic page migration and replication in large-scale non-uniform memory access (NUMA) machines by using the cache miss and TLB miss counts to infer the page affinity of the computing nodes. We also note that the SGI Origin 2000 [21] integrates a dynamic page migration algorithm with its directory-based cache coherence protocol, while the Sun Wildfire [24] implements a page migration and replication daemon that takes migration/replication decisions by monitoring the volume of cache misses to remote home. All these designs require OS assistance in remapping the migrated pages, copying the physical data, and shooting down the TLBs.[3] However, the overhead of doing OS-assisted dynamic migration in a NUCA environment would be hard to compensate for because in our simulated architecture the round-trip latency to the farthest L2 cache bank is slightly over ten nanoseconds higher than the local bank access latency. As a result, we design our dynamic page migration scheme as a fully hardwired mechanism. We show that this scheme is superior to OS-assisted static data placement mechanisms such as first-touch and application-directed page placement.

2. Baseline L2 Cache Architecture

We assume a baseline architecture similar to the one shown in Figure 1 with a two-level inclusive cache hierarchy. We will refer to an L2 cache bank as local to a core if it has the smallest round-trip wire delay among all the L2 cache banks from that core. Note that a core may have more than one local L2 cache banks. The private L1 caches are kept coherent with the help of an invalidation-based MESI directory protocol. The directory entries are kept with the tags of the L2 cache blocks. A miss or write-back request from an L1 cache is first forwarded to the home bank where the tag and the directory entry of the block are looked up and appropriate coherence actions are taken. A replacement from the L2 cache invalidates all the sharers' L1 cache blocks belonging to the replaced L2 cache block or retrieves the pertinent L1 cache blocks from the owner so that inclusion is maintained. The miss/refill/writeback requests from/to the L2 cache banks are handled by four on-die memory controllers each handling one quarter of the physical address space.

Given a physical address, there are various choices for selecting the L2 cache bank (i.e., the home bank) holding this physical address. Managing locality at a page grain requires an entire page to belong to a bank. Therefore, the bank number can be formed with bits selected from the cache index that fall outside the page offset. We use a page-interleaved mapping scheme where consecutive physical pages are assigned to consecutive L2 cache banks i.e., the bank number comes from the bits right next to the page offset. We find that this mapping scheme delivers performance and energy-efficiency similar to the popular block-interleaved mapping scheme. It is important to note that the home bank number is a function of the physical address generated by the L1 cache controller and the home bank of an L2 cache block changes as the block undergoes migration from one bank to another. At any point in time, the home bank of an L2 cache block is the bank where the block resides. The home bank of an L2 cache block derived from its original OS-assigned physical address will be referred to as the "original home bank" or the "original home".

3. Hardware Mechanisms

The dynamic page and block migration schemes presented in this section execute completely in hardware without any OS intervention. They do not move any data in the physical memory and the per-core L1 caches, and do not change the translations cached in the TLBs resident in the cores. The entire migration process is kept hidden from the virtual to physical address mapping layer of the OS.

3.1. Dynamic Page Migration

The migration algorithm involves three major steps, namely, deciding when to migrate a page, deciding where to migrate a page, and the actual migration protocol. Another important concern is how to locate a block in the L2 cache if it belongs to a migrated page. We discuss these four aspects in the following.

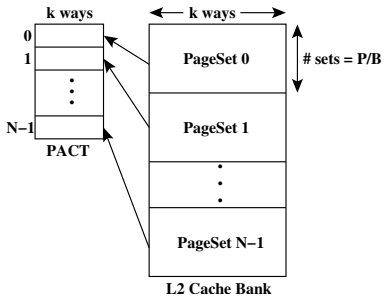

Figure 3. An L2 cache bank and the associated PACT. P is the page size and B is the L2 cache block size.

3.1.1. Migration Decision Algorithm

An L1 cache miss request (read, read-exclusive, or upgrade) from a core R is routed to the L2 cache bank holding the requested address. While the tag array is looked up, a migration decision is made based on the access pattern observed by the page holding the requested cache block, provided the current bank is remote to the requesting core R. The information needed to carry out this migration decision is stored in a set-associative page access counter table (PACT) indexed and tagged respectively with the lower and the remaining upper bits of the physical page frame number of the requested address after removing the bank number (each bank has a separate PACT). The PACT has the same number of ways as the L2 cache bank. The number of sets in the PACT is equal to the number of sets in the L2 cache bank divided by the number of L2 cache blocks per page (see Figure 3). All pages residing in PageSet n have their access information mapped to set n of PACT. Conflicts within a PACT set are resolved by traditional LRU replacement policy. An invalid way in set n of PACT necessarily corresponds to empty space equal to a page size in PageSet n of the L2 cache bank.

Logically each PACT entry contains several fields. Other than the valid bit, the tag, and the LRU states, the most important ones are an array of saturating counters for keeping track of the number of accesses to a page by a pair of adjacent cores (a pair of adjacent cores will be referred to as a cluster), the running maximum and second maximum access counts and the corresponding cluster ids, a sharer bitvector of length equal to the number of cores and the population count of the vector[4], a saturating counter for keeping track of the number of accesses to a page from the time the last sharer was added for that page, and an idle counter to decide when

[3] The poison states in the Origin directory help reduce the critical path.

[4] This sharer vector is per page and is different from the one maintained in the directory entry per cache block. This per-page sharer vector is not updated on coherence events such as invalidations because coherence events take place at block grain.

to invalidate an idle PACT entry. It is important to note that invalidating a PACT entry does not necessitate invalidating all the blocks belonging to the corresponding page. Also, a replacement from the L2 cache does not require any change in the PACT contents. Although apparently the PACT does not seem to scale well as the number of cores increases, it is always possible to maintain information at a coarser grain. For example, instead of maintaining access count per pair of adjacent cores, it can be maintained for a cluster of k cores that are close to each other depending on the underlying interconnect topology. Also, the per page sharer vector can be made coarse so that each entry of the vector refers to a cluster of cores.

We use a simple migration decision algorithm. If the cache block being accessed belongs to a page that is currently shared by more than one core and the difference between the maximum and the second maximum access counts is below a pre-defined threshold T_1 (signifying the absence of a single frequently accessing cluster), the "sharer mode" of the decision algorithm is invoked. In this mode a potential migration is flagged provided the number of accesses to this page since the addition of the last sharer exceeds a pre-defined threshold T_2 (signifying a stable set of sharers). On the other hand, the "solo mode" of migration is invoked if the page is currently accessed by a single core or the difference between the maximum and the second maximum access counts is more than or equal to T_1, and the requesting core R belongs to the cluster using the page most. To determine T_1 and T_2, we first execute a binary search for T_1 while holding T_2 fixed at a moderate value (e.g., 16). Once we arrive at the best value of T_1, we do a binary search for T_2 on both sides of the fixed value. We observe that the performance usually improves as T_2 increases up to a certain point. This is expected since premature migrations before the sharer set gets enough time to stabilize lead to an overall larger number of migrations. On the other hand, T_1 has little impact on the migration of solo pages because these pages get migrated on the second access to the page. However, too small a value for T_1 may lead to a wrong premature solo mode migration for a shared page followed by subsequent sharer mode migrations. Our experiments use $T_1 = 9$ and $T_2 = 29$.

3.1.2. Destination of Migration

Once a potential migration is flagged, the destination bank and a region within that bank where the migrated page will be placed have to be decided. For sharer mode migration, the page is migrated to a bank that minimizes the access time for the sharing cores on average, assuming uniform access distribution across the sharing cores. Every bank maintains a Proximity ROM, which stores, for each sharer combination (or sharing cluster combination if scalability becomes an issue), the four bank ids that have the least average access latencies for that sharer combination. The sharer vector is used to index into this ROM. Among the four banks read out from the ROM, the one with the least "load" is chosen as the destination. The load on a bank is defined as the number of pages mapped on that bank. This is updated at the end of a page migration and at the time of handling a page fault. However, the migration is canceled if the L2 cache bank currently holding the page happens to be one of the four banks read out from the ROM. The destination bank of a solo mode migration is chosen to be the least-loaded bank among all the local banks of the requesting core R.

Next, the algorithm proceeds to pick a physical page frame number mapping to the destination bank. This frame number will be assigned to the migrated page. The algorithm first tries to locate an unused index range covering a page in the destination bank by looking for an invalid entry in the destination bank's PACT. This is achieved by maintaining a small bitvector of length equal to the number of sets in the PACT (16 in our case). A bit position is set if the corresponding set in the PACT has an invalid way. This is updated at the time of a new allocation in the PACT. If one such PACT set exists, the algorithm generates a physical page frame number mapping to that set in the destination bank outside the installed physical memory range (to avoid coherence issues). If no PACT set in the destination bank has an invalid way, the algorithm first picks a non-MRU random PACT set and then selects the LRU page in that set as the target frame. The data residing in this frame is swapped with the data at the source of the migration. Note that these pages continue to have their original physical frame numbers outside the L2 cache.

3.1.3. Locating a Block in the L2 Cache

Before presenting the actual migration process, we first discuss how a block is located in the L2 cache. Since the migration process is hidden from the OS, the TLBs, and the L1 caches, the addresses coming out of the L1 cache controllers are the original OS-generated physical addresses. Therefore, each core maintains two tables that map the original physical frame number of an instruction or data page to the migrated frame number. These tables will be referred to as the iL1Map and dL1Map, respectively. Every L2 cache transaction generated by the L1 cache looks up the appropriate L1Map before it can get routed to the correct L2 cache bank. The latency of this lookup is hidden under the delay to enqueue the request in the port scheduler of the local switch on the ring. The organization of the L1Maps is exactly same as the TLBs. An entry in the appropriate L1Map is loaded from the unified second-level map table (discussed below) when the corresponding page table entry is loaded in the TLB at the time of a TLB miss. If a map is not found in the second-level table, an identity map is inserted in the L1Map. On a migration, the new physical frame number of a page is sent to the sharing cores (with the help of the sharing vector in the PACT) so that they can update their L1Maps appropriately. We would like to note that in rare situations, a writeback request (a dirty eviction from the L1 cache) may fail to find a map in the dL1Map because an L1Map replacement (done at the time of replacing the mirror entry in the corresponding TLB) does not flush the blocks belonging to that page from the L1 cache. In such situations, the block is first sent to its original home bank where the second-level map table is looked up and subsequently the block is re-routed to its current bank (if that is different from the original home). Each L1Map entry holds a tag, a mapped frame number, and a valid bit.

Each bank of the L2 cache needs two map tables. These are organized as tagged set-associative tables (details on sizes are presented in Section 5). The first table (referred to as the inverse L2Map) is accessed before sending an L2 cache response to the L1 caches or a request to the memory controllers. This table is indexed using the lower few bits of the physical page frame number (after dropping the bank number bits) extracted from the L2 cache address and, on a hit, returns the original OS-assigned physical frame number. The second table (referred to as the forward L2Map) is accessed before accepting external requests (including refills and interventions) from the memory controllers. This table is indexed using the lower few bits of the OS-assigned physical page frame number (after dropping the bank number bits) and, on a hit, returns the migrated page frame number within the L2 cache. The TLB miss handler looks up this table to fill in the L1Map entries. If the forward L2Map of the home bank of page p contains the map $p \rightarrow q$ (meaning all the blocks belonging to the original page p can now be found at page address q in the L2 cache), the inverse L2Map of the home bank of page q will contain the map $q \rightarrow p$. We would like to note that the L2 cache miss path gets slightly lengthened because an L2 cache miss request needs to

look up the inverse L2Map at the originating bank before it can get routed to the appropriate memory controller and similarly, a refill needs to look up the forward L2Map at the original home before it can get routed to the appropriate L2 cache bank. However, data migration can easily compensate for this loss by improving the latency of the L2 cache hits. The number of L2 cache hits is much larger than the misses. The MSHR for an L2 cache miss is always allocated at the original home bank so that migration of pages with outstanding misses can be smoothly handled. However, the refill transaction of such a miss must be held up at the original home until the migration is completed. The complete address translation flow is shown in Figure 4. Each L2Map entry holds a tag, a mapped page frame number, LRU states, and a valid bit.

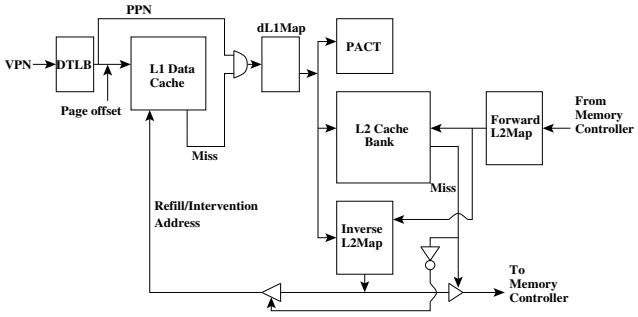

Figure 4. The translation flow showing how the dL1Map of a core and the forward and inverse L2Maps of an L2 cache bank interact with various other components. VPN and PPN stand for virtual and physical page numbers.

Finally, we note that L2Map replacement is a costly operation requiring movement of the involved page P from its migrated location to its original location. The page currently residing in the original location of P has to be replaced from the L2 cache. Therefore, it is necessary to size the L2Maps appropriately so that the volume of L2Map replacements is not too high.

3.1.4. Migration Protocol

The actual migration process involves a few steps. The inverse L2Map at the source/destination bank is looked up with the source/destination page number (call them S and D) to obtain their actual physical frame numbers (call them s and d). Next, s and d are used to index into the forward L2Maps of the home banks of s and d, respectively, and the corresponding two maps are swapped. Two entries of the inverse L2Maps of S and D are also swapped. This entire process involves updating one entry each in the inverse L2Maps of the two banks participating in the migration and one entry each in forward L2Maps of two other banks, and requires only one read port and one write port in each L2Map. Finally, the two new forward maps ($s \rightarrow D$ and $d \rightarrow S$) are sent to the relevant cores so that they can update their L1Maps. These cores acknowledge this update. Once these L1Map update acknowledgments are received at the L2 cache, the cache blocks (data, coherence states, and directory entry) belonging to the source/destination page are read out from the source/destination bank and sent to the destination/source bank. During the entire migration process the source and the destination L2 cache banks do not accept any request as the read/write ports in these two banks remain busy copying the involved pages. This "interlock" protocol requires careful deadlock analysis. Apparently, the completion of the migration protocol does not depend on the completion of any other events. However, the migration phase may evict cache blocks from the L2 cache to make room for the migrated blocks (a perfect swap may not be possible always). Since L2 cache evictions must send notifications to the sharer/owner L1 caches to maintain inclusion, the migration phase potentially requires space in the L2-L1 invalidation/intervention virtual queues. However, through a cycle-free design of the virtual queue dependence graph, one can guarantee that the L2-L1 invalidation/intervention virtual queues will drain irrespective of how the migration phase is making progress. The details of this design technique are beyond the scope of this paper. In summary, our protocol is deadlock-free by design.

3.2. Dynamic Cache Block Migration

Past proposals on data migration in CMP NUCA evaluated dynamic cache block migration. For comparison with our dynamic page migration proposal, we also implement a dynamic cache block migration scheme suitable for our architecture. In the following, we discuss the pertinent algorithms.

The block migration decision algorithm is found to deliver the best performance for $T_1 = 4$ and $T_2 = 27$. The block access counter table (BACT) replaces the PACT. However, the BACT is tightly coupled with the L2 cache tag array and does not require its own tag, valid bit, and LRU states.

The destination bank of a migration is decided using the same algorithm as in page migration. However, in this case the load on a bank is defined as the number of cache blocks allocated in a bank. We experiment with two algorithms for selecting the destination block of migration in the destination bank. The first algorithm picks the first set with an invalid way in the destination bank (can be done by maintaining a bitvector as discussed for page migration) and uses the invalid way as the target frame. In a pathological situation, this algorithm may end up selecting all the ways of a set as the targets for consecutive migrations leaving no room for the non-migrated blocks that may map to the same set in near-future. Applications with large data sets show increased volume of conflict misses when this algorithm is implemented. Our second algorithm solves this problem by picking the next round-robin set with an invalid way in the destination bank, thereby distributing the migration load evenly in a bank. If no set with an invalid way is found, both the algorithms pick the LRU way within a randomly selected set.

For locating a block in the L2 cache, the forward and inverse L2Maps must have one entry per L2 cache block. The organization of the forward L2Map in terms of the number of sets, ways, and banks is exactly same as the main L2 cache. To avoid more than doubling the total L2 cache access latency of the external interventions due to serialized access to the forward L2Map followed by the L2 bank tag array, each entry of the forward L2Map stores not only the new bank and the new index of the migrated block, but also the target way at the new index. This ensures that a direct lookup in the new L2 cache bank's target tag and data way would be enough. However, this may require some extra circuitry and a re-organization of parts of the cache tag array, since the conventional tag array organizations of set-associative caches do not allow direct lookup of a particular way while maintaining the LRU states of all the ways correctly. Of course, the way information is not used by the refills. Each entry of the forward L2Map also needs to store a tag. The inverse L2Map shares the tag array with the main cache and stores only the original home bank number and the original index within that bank.

The L1Maps contribute to most of the storage overhead of block migration. The L1Map in each core must replicate the entire forward L2Map. On completion of an L2 cache refill, the bank number, the set index, and the way allocated to the refilled block must be broadcast to all the cores so that this block can be located efficiently in the L2 cache on subsequent L1 cache misses to this

block. It is important to note that such a broadcast is unnecessary in the case of page-grain migration. To offer a lower bound for the storage overhead of block-grain migration, in Section 5, we consider a floorplan-oblivious design where only one replica of the forward L2Map is placed in such a way that it is somehow physically close to all the cores. In both the designs, the organization of the L1Map is same as the forward L2Map.

4. OS-assisted Mechanisms

The OS mechanisms modify the default virtual to physical address assignment algorithm to improve locality (default is the bin-hopping algorithm [18]). In the following, we briefly discuss how first-touch page placement and application-directed page placement can improve data locality in CMP NUCAs. In all the mechanisms (including baseline and hardware mechanisms), the pages belonging to the stack space of a thread are allocated in the banks local to the thread.

The first-touch page placement algorithm assigns a physical page frame number to a virtual page such that the physical page maps to an L2 cache bank local to the core touching the page for the first time. The first-touch core id must be passed on to the page fault handler. The page fault handler picks the next free physical page frame from the least-loaded L2 cache bank that is local to the first-touch core. If all the page frames mapping to the local banks of the first-touch core are exhausted, the page fault handler selects the next free page frame from the globally least-loaded L2 cache bank. The load on a bank is measured as the number of pages mapped on that bank.

In application-directed page placement we allow the application to specify the affinity of a range of virtual pages via an Irix-style system call (see Chapter 2 of [27]). The system call specifies three arguments, namely, the starting virtual address (as a pointer to a data structure), the ending virtual address (again a pointer to a data structure), and a core id. The OS maps the pages in the specified range on banks local to the specified core. In this paper, we modify the applications by hand so that all the page placement system calls are inserted just before the main parallel computation begins. Notice that it is a static technique where each partition of a data structure can use a one-time system call to specify its best possible affinity to a core for the entire parallel execution. While the private pages can specify their affinity accurately, the pages belonging to the shared data structures are placed in a round-robin fashion across the L2 cache banks local to the sharing cores.

At the time this affinity-based mapping takes place, several pages may already have been assigned physical frames according to the default policy. For such a page, the OS invalidates the old translations from all the TLBs, flushes the cache blocks belonging to the page out of the entire cache hierarchy and writes the dirty ones back to memory, copies the page from old frame to new frame in physical memory, and inserts the new translation in the page table. We found that as the cache size increases, this cache flush severely impacts the performance of some of the applications. So we explore an alternate design that leverages the L1Map and L2Map tables of the hardwired page migration mechanisms to avoid flushing the caches and copying the pages in physical memory. In this design, the OS, on completing a page placement system call, registers the new maps in the L1Map and L2Map tables, copies the affected blocks from the old location to the new location within the L2 cache only, and does not modify the TLBs, L1 caches, or the physical memory. Subsequently, the map tables correctly translate between the old and new addresses at the L2 cache boundary.

5. Simulation Environment

We simulate a MIPS ISA-based eight-core chip-multiprocessor using a detailed in-house execution-driven simulator. A high-level floorplan was shown in Figure 1. A directory entry is maintained per L2 cache block and is kept with its tag. The directory maintains the M, S, and I states (the M and E states are merged into the M state [21]) and has an eight-bit vector to maintain the sharer list. The salient features of our simulated system are shown in Table 1.

Table 1. Simulated system

Parameter	Value
Number of cores	8, clocked at 3.2 GHz
Process/V_{dd}/V_t	65 nm/1.1 V/0.18 V
Pipe stages	18
Front-end/Commit width	4/8
BTB	256 sets, 4-way
Branch predictor	Tournament (Alpha 21264)
Br. mispred. penalty	14 cycles (minimum)
ROB/RAS/Branch stack	128/32/32 entries
Integer/FP registers	160/160
Integer/FP/LS queue	32/32/64 entries
ALU/FPU	8 (two for addr. calc.)/3
Int. mult./div. latency	6/35 cycles
FP mult. latency	2 cycles
FP div. latency	12 (SP)/19 (DP) cycles
ITLB, DTLB	64/fully assoc./Non-MRU
TLB miss penalty	ROB flush+65 cycles+ PTE reload [15]
Page size	4 KB
Private L1 Icache	32 KB/64B/4-way/LRU
Private L1 Dcache	32 KB/32B/4-way/LRU
Store buffer	32
L1 MSHR	16+1 for retiring stores
L1 cache hit latency	3 cycles
Shared L2 cache	16 MB/128B/16-way/LRU
L2 MSHR	16 per bank$\times$16 banks
L2 bank tag latency	7 cycles
L2 bank data latency	3 cycles (one way)
Switch latency of ring	1 cycle, 2 cycles at source
Switch-to-switch wire delay	2 cycles (horizontal segment) 1 cycle (vertical segment)
L2 cache prefetcher	Seq., two blocks look-ahead
Memory cntr. freq.	1.6 GHz
System bus width/freq.	64 bits/1.6 GHz
SDRAM bandwidth	6.4 GB/s per controller
SDRAM access time	70 ns (row buffer miss) 30 ns (row buffer hit)

We size the L2 cache such that the die area is close to 435 mm^2 with 65 nm process [26]. We conservatively estimate the out-of-order multiple issue core size to be 4 mm$\times$4 mm [20]. We design the ring interconnect using the "semi-global" M5 metal layer [4]. Assuming the M5 layer wiring pitch of 330 nm [26], we estimate that we can comfortably fit a bidirectional ring of width 1074 bits (1024 bits of data matching the width of the L2 cache block, 40 bits of address, and 10 bits of control for carrying request/response opcode and source/destination core or bank id) in

an area of 20 mm×1.5 mm. We design each wire segment (horizontal and vertical) between the switches with optimally placed repeaters [1] and compute the wire delay assuming the mid-level metal capacitance and resistance presented in [1]. Assuming every <core, L2 cache bank> pair to be equally likely, the average NUCA routing latency in our simulated architecture is 15 cycles at 3.2 GHz. Therefore, the round-trip L2 cache hit latency is (L1 cache tag latency + 15 + L2 cache tag and data latency + 15 + one cycle critical word delivery time in L1 cache) or 44 cycles or 13.75 ns. The maximum and minimum round-trip L2 cache hit latencies are 64 cycles (20 ns) and 24 cycles (7.5 ns), respectively.

We assume the basic memory cell size of the L2 cache to be 0.624 μm^2 [26]. We estimate from CACTI [16] the area devoted to the bit cells in a 1 MB 16-way set-associative bank to be 47% of the total area of the bank when equipped with one read port and one write port. Therefore, the total area of one bank is $0.624\ \mu\text{m}^2 \times (2^{23} + 36 \times 2^{13})/0.47$ or 11.5 mm^2. Note that each block has 36 bits of tag and state (20 bits of tag, four state bits for modified, valid, pending, and dirty in L1, eight bits of sharer vector, and four bits of LRU state). Including the L2 cache controllers and the four on-die integrated memory controllers, we estimate the chip size to be at most 20 mm×22 mm.

All the cache latencies are determined with CACTI. We assume serial tag and data access for L2 cache banks. The L2 cache tag array is assumed to have one read port, one write port, and one read/write port for read-modify-write operations to the directory. We assume that each data way of a bank is organized as a direct-mapped cache[5] and one of the direct-mapped ways is accessed after the tag hit/miss completes.

Our dynamic power model is significantly influenced by Wattch [7], but includes the power models of various off-core components including the ring interconnect, the port buffers, the virtual queues, and the port schedulers. All simulations in this paper are done with aggressive clock gating enabled.

Our subthreshold and gate leakage power models are developed based on the techniques proposed in [10] and [25], respectively. However, we have improved these models wherever possible by cross-validating the leakage current values with HSPICE simulations. Our L2 cache data RAM banks implement the drowsy cells [14], which switch a 128 KB subbank to a low voltage supply while retaining the data, if the subbank is not accessed for 1000 clock cycles.

Finally, our approximate DRAM energy model is developed based on Micron technical notes [22, 23]. We assume that each memory controller connects to a 1 GB DIMM built out of 18 512 Mb x4 DRAM chips, two of which are for ECC. The DRAM energy model uses published figures for a highly loaded DDR2-400 512 Mb x4 chip scaled up to 400 MHz. Our modeled peak power drain per DRAM access (at 400 MHz) turns out to be 4.3 W.

The details of the shared memory parallel applications and the multiprogrammed workloads used in our evaluation are shown in Table 2. All the shared memory applications use hand-optimized array-based queue locks and scalable tree barriers. We run the shared memory applications from beginning to completion. The multiprogrammed workloads are prepared from SPEC 2000 and BioBench [2] suites. The alternate cores are activated in the 8-core chip for running these 4-way workloads. For each of the constituent applications, we extract the representative sample of one billion dynamic instructions using the SimPoint toolset [28]. The SPEC 2000 applications use the `ref` input sets, while the BioBench applications use `sitchensis.fa`, `tufa420.seq`, and `tufa420.phy` as inputs. We allow each constituent application to commit one billion instructions. We report the average execution time of the workload, which is the arithmetic mean of the execution times of the constituent applications.

Table 3 shows the sizes of various storage structures required by the hardwired migration mechanisms. The number of sets in the L2Maps is kept half of the number of sets in the L2 cache bank for page-grain migration. This configuration is found to give satisfactorily low volume of L2Map replacements. In ideal block migration (last column), only one copy of the L1Map is shared across all the cores. Although this design is not feasible in reality, it does offer a lower bound on the storage needed by block-grain migration. Overall, page-grain migration requires less than 5% of total storage budget (total budget includes overhead added to the L2 cache storage; L2 cache storage is 2^{27} data bits + 36×2^{17} tag, directory, and state bits or 16960 KB). Block-grain migration devotes more than a quarter of the total budget, while the ideal design brings it down to slightly more than 10%.

Table 2. Simulated workloads

Shared memory applications		
Name	Input	Source
Barnes	16384 bodies	SPLASH-2
Matrix[6]	8192×8192 matrix, 256K non-zeros, 50 iterations	DIS [30]
Equake	MinneSPEC, ARCHduration 0.5	SPEC OMP
FFTW	1024×16×16 complex doubles	FFTW
Ocean	258×258 grid	SPLASH-2
Radix	2M keys, radix 32	SPLASH-2
Multiprogrammed workloads		
Name	Composition	
MIX1	ammp, gzip, vortex, wupwise	
MIX2	apsi, ammp, equake, mesa	
MIX3	apsi, twolf, gzip, mesa	
MIX4	apsi, vpr, equake, mesa	
MIX5	ammp, tigr, clustalw, phylip-protdist	
MIX6	gzip, tigr, clustalw, phylip-protdist	
MIX7	twolf, tigr, clustalw, phylip-protdist	
MIX8	vpr, tigr, clustalw, phylip-protdist	

6. Simulation Results

We present our detailed simulation results in this section. We compare the hardwired and OS-assisted locality management schemes in terms of performance as well as energy consumption. We close this section with an evaluation of how L1 cache prefetching compares and interacts with our proposal of dynamic page migration.

6.1. Performance Comparison

Figure 5 presents the execution time (upper panel) and the aggregate load/store stall cycles (lower panel)[7] of the shared memory applications normalized to the baseline CMP with static NUCA.

[5] This is different from the CACTI serial access model, which organizes the entire data array as a direct-mapped cache.

[6] This is a sparse solver using iterative conjugate gradient.

[7] These are the cycles where the instruction retirement unit of a core could not commit any instruction due to an incomplete load or store operation at the head of the ROB. We enforce sequential consistency (as in the MIPS R10000).

Table 3. Storage overhead of hardwired migration mechanisms

Component	Page-grain	Block-grain	Block-grain ideal
Forward L2Map	256 sets×16 ways×16 banks ×49 bits = 392 KB	512 sets×16 ways×16 banks ×38 bits = 608 KB	512 sets×16 ways×16 banks ×38 bits = 608 KB
Inverse L2Map	256 sets×16 ways×16 banks ×49 bits = 392 KB	512 sets×16 ways×16 banks ×13 bits = 208 KB	512 sets×16 ways×16 banks ×13 bits = 208 KB
L1Maps	2×8 cores×64 entries ×57 bits = 7.1 KB (i & d)	8 cores×2^{17} entries ×38 bits = 4864 KB (unified)	2^{17} entries× 38 bits = 608 KB (unified)
PACT/BACT	16 sets×16 ways×16 banks ×98 bits = 49 KB	512 sets×16 ways×16 banks ×68 bits = 1088 KB	512 sets×16 ways×16 banks ×68 bits = 1088 KB
Proximity ROM	256 entries×16 banks ×16 bits = 8 KB	256 entries×16 banks ×16 bits = 8 KB	256 entries×16 banks ×16 bits = 8 KB
Total	848.1 KB (4.8%)	6776 KB (28.5%)	2520 KB (12.9%)

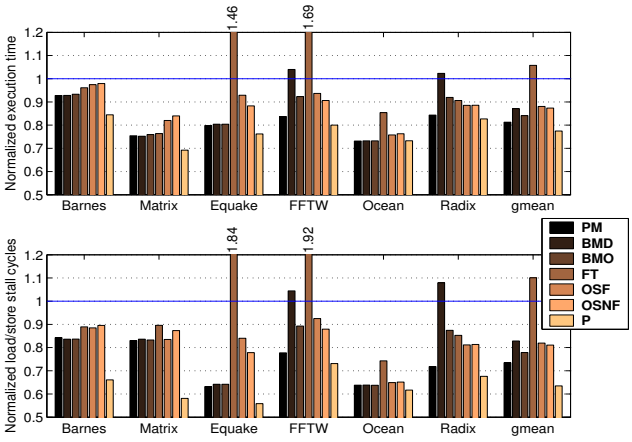

Figure 5. Execution time (top panel) and aggregate load/store stall cycles (bottom panel) normalized to baseline static NUCA for shared memory parallel applications.

We do not present the L2 cache miss rate data because different optimizations lead to execution of different numbers of instructions and cause varying numbers of L2 cache accesses rendering the miss rate data very misleading in some cases. Instead, we present the load/store stall cycle statistics, since the overall effect of any cache optimization is best captured by this data. For each application, we present the performance of seven algorithms, namely, page-grain migration (PM or PageMigration), cache block-grain migration with default destination set selection algorithm which picks the first set with an invalid way (BMD or BlockMigrationDef), cache block-grain migration with optimized destination set selection algorithm which considers all sets with an invalid way before picking the same set twice (BMO or BlockMigrationOpt), OS-assisted first-touch placement (FT or FirstTouch), application-directed page placement with OS-assisted cache flush (OSF or OSFlush), application-directed page placement without cache flush and DRAM page copy (OSNF or OSNoFlush), and an ideal placement where each L2 cache access is charged local bank access latency (P or Perfect). Page migration is able to reduce execution time by 7.2% (Barnes) to 26.9% (Ocean), averaging (geometric mean) at 18.7% (see the last group of bars). Although Matrix has almost 45% of L2 cache accesses to shared pages (see Figure 2), it is encouraging to note that our page migration scheme achieves a 24.6% reduction in execution time without resorting to any data replication. Default and optimized block migration algorithms deliver performance equivalent to page migration in all applications except FFTW and Radix. In these two applications, the default policy suffers from extra conflict misses in the L2 cache (already explained in Section 3.2) and the optimized policy fails to convert a sufficient number of remote bank accesses to local accesses. The primary reason for this is that compared to page migration, block migration has to be less aggressive to keep the migration overhead low, which cannot be amortized across several blocks as in page migration.

Among the OS-assisted mechanisms, first-touch placement performs poorly for the applications where a single thread initializes all the data (Equake, FFTW, Ocean). Application-directed page placement with cache flush does not deliver performance as good as the best of the hardwired mechanisms. The primary reason for this is cache flush and DRAM copy overhead, and Equake and FFTW are the most affected ones. In these two applications, the volume of L2 cache miss increases due to cache flush. The second reason is that the pages that exhibit different sharing patterns in different phases cannot be handled by static placement mechanisms. Application-directed placement without cache flush and DRAM copy helps nullify the first effect (see Equake and FFTW), but still fails to deliver the best performance due to the presence of shared pages. In some applications, turning off cache flush increases the execution time slightly due to some extra accidental conflict misses resulting from the new virtual to physical address translations.

Finally, the Perfect bar shows that Matrix has the maximum potential of performance improvement (30.8% reduction in execution time). On average, Perfect reduces execution time by 22.5% and dynamic page migration comes surprisingly close achieving a reduction of 18.7%. Optimized block migration reduces execution time by 15.9% and application-directed page placement without cache flush achieves a 12.6% reduction in execution time on average.

We note that the aggregate load/store stall cycles directly correspond to the trend in execution time discussed above. On average, dynamic page migration reduces load/store stall cycles by 26.5%, while optimized block migration comes very close with an average reduction of 22.1%. The poor performance of first-touch placement can be easily attributed to 10% increased load/store stall cycles. The application-directed page placement techniques reduce load/store stall cycles by about 18%.

Figure 6 presents the normalized average execution time and load/store stall cycles for the multiprogrammed workloads. In these results we do not include application-directed page placement because it would have performance similar to first-touch placement due to the absence of sharing. Dynamic page migration reduces execution time by 9.5% (MIX4) to 16.4% (MIX1) with an average reduction of 12.6%. Default block migration performs poorly for a large number of workload mixes. Due to much larger data footprint than the shared memory applications, the multiprogrammed workloads are very sensitive to the destination set selection algorithm. The optimized block migration algorithm helps

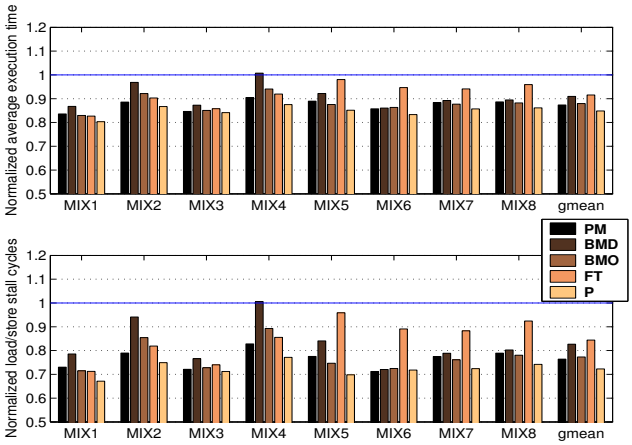

Figure 6. Execution time (top panel) and aggregate load/store stall cycles (bottom panel) normalized to baseline static NUCA for multiprogrammed workloads.

resolve this issue and performs close to dynamic page migration on average. However, it performs poorly for MIX2 and MIX4. This will be explained below with the help of Table 4. The first-touch placement delivers performance worse than page migration, except for MIX1. However, it delivers much better relative performance on the multiprogrammed workloads than on the shared memory applications. On average, first-touch placement achieves 8.5% reduction in execution time. Due to large data sets in MIX4, MIX5, MIX6, and MIX7, the first-touch placement fails to statically place all the data touched by each thread in local banks and resorts to frequent spilling to non-local banks. On average, Perfect shows a potential of 15.2% reduction in execution time, while page migration delivers the best performance by achieving a reduction of 12.6%, followed closely by optimized block migration. We note that the multiprogrammed workloads benefit less from the dynamic migration techniques compared to the shared memory parallel applications. The primary reason for this is that the multiprogrammed workloads suffer from a bigger volume of L2 cache misses, thereby diminishing the importance of on-chip physical locality (Amdahl's Law effect). This phenomenon brings out an important trend, namely, increasing (decreasing) the shared L2 cache capacity will lead to an increase (decrease) in the importance of on-chip cache locality optimization techniques. Finally, we would like to point out that the trends in execution time discussed above are reflected clearly in the load/store stall cycle statistics (lower panel of Figure 6).

To further understand the performance results, Table 4 presents the percent local L2 cache accesses in baseline static NUCA, page migration, optimized block migration, first-touch, and application-directed page placement without cache flush. For the shared memory applications, on average (harmonic mean), dynamic page migration succeeds in making more than 80% of accesses local, while block migration falls short mostly due to FFTW and Radix (already explained). Interestingly, for Barnes and Matrix, page migration is able to achieve similar percentage of local bank accesses (about 62%), but the reduction in execution time for Barnes is much less than that in Matrix. This is mostly because Matrix spends more time in the cache hierarchy than Barnes and as a result, cache hierarchy optimizations help Matrix more than Barnes. First-touch placement can only offer slightly over 40% local accesses. Application-directed page placement cannot handle the shared pages optimally and can achieve only 54.1% of local accesses. This relative trend of the hardwired and OS-assisted mechanisms is clearly reflected in the execution time. For the multiprogrammed workloads, page and block migration enjoy about 85%

Table 4. Percent local L2 cache accesses

App.	Base	Page	Block	FT	App.-dir.
Barnes	21.5	62.3	57.6	41.6	38.1
Matrix	19.9	62.1	60.2	33.9	34.0
Equake	22.0	96.6	94.6	25.0	54.6
FFTW	20.3	93.3	76.9	57.0	76.2
Ocean	21.1	98.8	98.3	87.3	98.7
Radix	21.1	99.0	66.8	59.7	73.6
Hmean	**21.0**	**81.7**	**72.6**	**43.1**	**54.1**
MIX1	16.9	82.3	90.5	86.7	–
MIX2	21.1	70.3	61.0	71.8	–
MIX3	25.2	94.0	95.6	95.2	–
MIX4	24.7	86.0	78.6	74.2	–
MIX5	19.1	86.5	88.0	52.1	–
MIX6	22.3	91.2	90.0	67.5	–
MIX7	22.2	89.2	91.2	68.7	–
MIX8	24.6	87.9	89.4	58.9	–
Hmean	**21.6**	**85.3**	**84.0**	**69.6**	–

local L2 cache accesses, while first-touch delivers 69.6% of local accesses. First-touch fails to do well in the last four workloads, which have applications from the BioBench suite. Among these applications, tigr has a very large data footprint. First-touch could not allocate all the data of this thread locally and had to resort to spilling on non-local banks. For MIX2 and MIX4, block migration fails to convert as many number of remote accesses into local accesses as page migration does. As a result, these are the only two workloads where optimized block migration fails to deliver performance close to page migration.

6.1.1. Isolating the Performance Factors

In the discussion above, we have shown that optimized dynamic block migration delivers performance close to dynamic page migration except for a few cases, namely, FFTW and Radix among the shared memory parallel programs and MIX2 and MIX4 among the multiprogrammed workloads. While we have already explained the performance difference of these cases with the help of local and remote access statistics, in the following we look for more fundamental reasons, if any. Apparently, there are two benefits of page-grain migration that block-grain migration fails to offer. First, an entire page is migrated after observing the accesses to a part of the page. So the accesses to the rest of the page should enjoy a prefetching effect. However, as already hinted at in Figure 2, we found that after a page or a block is migrated, it is accessed a significant number of times, thereby making the prefetching effect of page migration insignificant. In other words, if a block is accessed locally N_p times after the page containing the block is migrated in the page-grain scheme and the same block is accessed locally N_b times after the block is migrated in the block-grain scheme, we find that $|1 - \frac{N_p}{N_b}|$ is insignificantly small.

The second apparent benefit of page migration is that it can transfer a number of blocks in a bundle thereby opening up the opportunity of pipelining the transfers. In the block-grain scheme such a pipeline cannot be set up because only one block is transferred at a time. Also, the number of page migration attempts is expected to be less than that of block migration, provided the page-grain access pattern is stable enough. However, one should keep in mind that page migration communicates 32 times more data compared to block migration per transfer and hence, demands higher peak bandwidth. But it enjoys a latency advantage per transfer due to pipelining. To confirm this phenomenon, we nullified all

migration overhead, re-tuned T_1 and T_2 so that block-grain migration can now be more aggressive, and executed the applications/workloads with both page and optimized block migration. As expected, we found that the two techniques now delivered similar performance under this setting. Therefore, we conclude that the performance gap between page and block migration stems primarily from the inability of block-grain migration techniques to pipeline the transfers, thereby failing to amortize the performance overhead across multiple blocks.

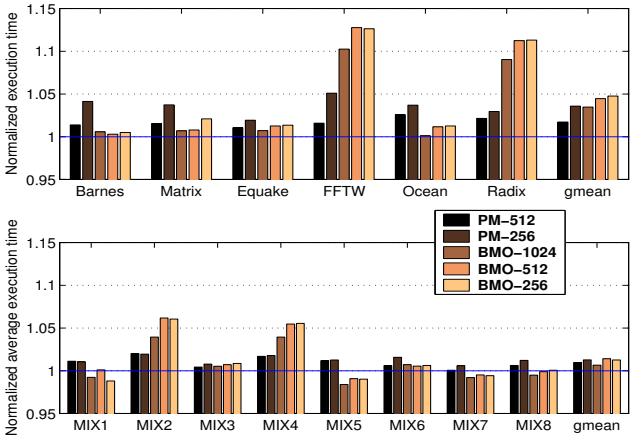

Figure 7. Execution time normalized to page migration on a 1024-bit data ring for shared memory parallel applications (top panel) and multiprogrammed workloads (bottom panel). PM-n represents page-grain migration on an n-bit data ring and BMO-n represents optimized block migration on an n-bit data ring.

Finally, as hinted at in the above discussion, there is an important trade-off that must be understood when taking the decision of implementing fine-grain or coarse-grain migration. The trade-off is among latency, bandwidth, and storage. Coarse-grain migration requires less storage, takes less time to migrate the same amount of data (an effect of pipelined transfer), but makes the interconnect traffic bursty and demands higher peak bandwidth, since at the time of migration a large amount of data should be moved in a pipeline within a small amount of time. To understand this bandwidth demand, we evaluate the page-grain and the optimized block-grain migration schemes in a constrained bandwidth environment. The results presented till now assume a bidirectional data ring of width 1024 bits in each direction. Figure 7 shows the results for 512-bit and 256-bit bidirectional data rings normalized to page-grain migration running on a 1024-bit data ring. As expected, constraining the bandwidth of the interconnect affects page-grain migration more than block-grain migration. Also, the impact on the shared memory applications is more pronounced than the multiprogrammed workloads because the former type of applications must execute a larger number of migrations to handle dynamic sharing of pages. Overall, with a 256-bit data ring, the page-grain scheme suffers from a nominal 3.6% average increase in execution time for the shared memory applications compared to a 1024-bit data ring. For the multiprogrammed workloads, the corresponding increase is only 1.3%. The block-grain scheme is relatively more tolerant to bandwidth variation.

6.2. Energy Overhead

Figures 8 and 9 present the total energy consumption (includes the on-chip components and off-chip DRAM) of different mechanisms (excluding Perfect) normalized to baseline. We show the

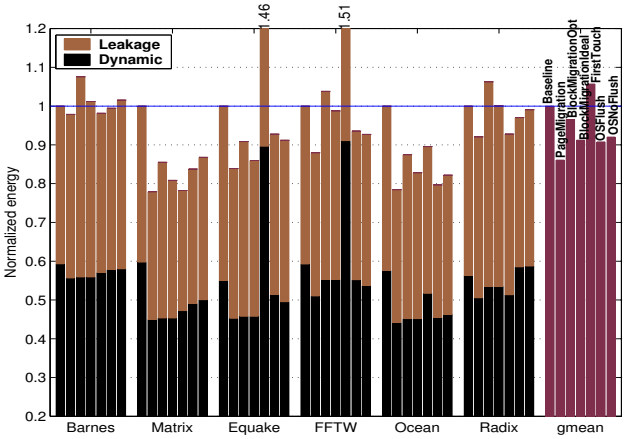

Figure 8. Energy consumption normalized to baseline static NUCA for shared memory applications.

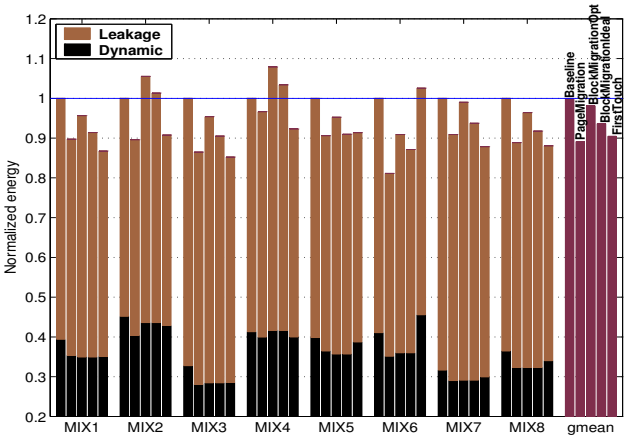

Figure 9. Energy consumption normalized to baseline static NUCA for multiprogrammed workloads.

division of total energy into dynamic and leakage energy. The last group of bars presents the normalized average of total energy. For block migration, we present the results for the optimized design and the ideal design that was introduced in Table 3. The latter is expected to save some leakage energy.[8]

Since the locality management policies reduce the execution time, we should expect to save total energy provided the extra dynamic and leakage energy overhead is low. On average, dynamic page migration turns out to be the most energy-efficient design. It saves 13.9% and 10.8% energy compared to the baseline static NUCA design for the shared memory applications and the multiprogrammed workloads, respectively. Optimized block migration suffers due to extra leakage energy consumption resulting from its high storage requirement. It saves only 3.5% and 1.9% energy for the shared memory and multiprogrammed workloads, respectively. The ideal block migration design does help save more energy, but fails to come close to page migration. As expected, the first-touch placement policy increases energy consumption by 5.7% in the shared memory applications due to increased volume of L2 cache misses and DRAM accesses, but it comes close to dy-

[8] The multiprogrammed workloads have higher proportions of leakage energy consumption compared to the shared memory applications because the irregular access patterns in the former reduce the efficiency of the subbank-grain drowsy technique.

namic page migration for the multiprogrammed workloads. It is encouraging to note that the energy-efficiency of the application-directed page placement policy (OSFlush and OSNoFlush) is similar to the ideal block migration design for the shared memory applications.

6.3. Impact of L1 Cache Prefetching

L1 cache prefetching also attacks the same problem that the locality management policies aim at solving. We integrate a multi-stream stride prefetcher into the L1 cache controller of each core. The prefetcher keeps track of sixteen read and sixteen write (i.e. read-exclusive and upgrade) streams per core. The stream size is fixed at 4 KB matching the page size. The prefetcher, after seeing the same stride twice in a row, prefetches six blocks ahead into the stream. Some of these six blocks may cross the designated physical page boundary of the stream, but the first among these six blocks must belong to the designated page. However, a prefetch request to an unmapped address is dropped. Table 5 compares the effectiveness of L1 cache prefetching against our proposed dynamic page migration scheme for the shared memory applications (ShMem) and the multiprogrammed workloads (MultiProg).

Table 5. Average (gmean) reduction in execution time compared to baseline static NUCA

Workload	L1 Pref.	Dyn. Page Mig.	Both
ShMem	14.5%	18.7%	25.1%
MultiProg	4.8%	12.6%	13.0%

For the shared memory applications, L1 cache prefetching turns out to be quite effective and reduces execution time by 14.5% on average compared to the baseline. However, dynamic page migration is more effective than L1 cache prefetching and reduces execution time by 18.7%. When both L1 cache prefetching and dynamic page migration are enabled, a 25.1% reduction in execution time is achieved. The multiprogrammed workloads do not appear to be easy to prefetch with a multi-stream stride prefetcher. For these workloads, L1 cache prefetching fails to improve performance much. Prefetching and page migration together reduce the execution time by 13% on average for these workloads.

7. An Analytical Model

Before we conclude the paper, in the following, we present a simple, yet useful, analytical model to estimate the performance benefits of data migration (the grain of migration is not captured in this model). For a certain workload, let the number of L2 cache hits be h and the number of misses be m. We will refer to the L2 cache miss rate as $r = \frac{m}{m+h}$ (this is really the local miss rate as opposed to the global miss rate). Let mx be the number of cycles on the critical path to serve the m L2 cache misses i.e., for these many cycles, the retirement of the requesting thread on the critical path of the parallel application is stalled with an L2 cache miss at the head of the ROB. Similarly, let hy be the number of cycles on the critical path to serve the h L2 cache hits. Let us further assume that due to data migration, the number of cycles on the critical path to serve the L2 cache hits improves to hys with $0 < s < 1$. Let the number of busy cycles on the critical path of the parallel execution be C. Therefore, the execution time with migration normalized to that without migration is given by $N = \frac{mx+hys+C}{mx+hy+C}$. We assume that the volume of L2 cache hits and misses remains unchanged after introducing data migration and we ignore the overhead of doing data migration. Substituting $\frac{x}{y} = A$, $\frac{C}{mx+hy+C} = t$ i.e. $C = \frac{t(mx+hy)}{1-t}$, and $\frac{m}{h} = \frac{r}{1-r}$, we get $N = \frac{rA+(1-r)[s+t(1-s)]}{rA+1-r}$. We make a few simple observations from this expression. First, if r is one or s is one, data migration does not offer any advantage, as expected. Second, as A grows larger signifying off-chip latency getting bigger compared to on-chip L2 cache hit latency, data migration loses importance. This trend nicely captures the effect of data migration on applications with large working sets. These applications will not benefit much from data migration. Finally, for an application, if t is large (compute intensive), data migration will fail to offer much benefit.

To predict the benefit of data migration in typical situations, we need to fix the values of the parameters. We set y to the average round-trip L2 cache hit latency i.e. 13.75 ns (as discussed in Section 5). We set x to 100 ns (mean of uncontended row buffer hit and row buffer miss latency + 20 ns for DRAM bus transfer + 10 ns for system bus transfer + 13.75 ns round-trip between L2 cache and L1 cache + 6.25 ns miscellaneous). Therefore, we get $A = 7.2728$. The average value of s can be computed as the fastest round-trip L2 cache hit latency over the harmonic mean of round-trip L2 cache hit latency provided the baseline bank access distribution is uniform. When the switch and wire latencies presented in Table 1 are taken into consideration, the floorplan shown in Figure 1 has the average value of s equal to 0.6195. The ranges $r \in [0.04, 0.06]$ and $t \in [0.4, 0.5]$ capture the most commonly encountered values of r and t in our workloads when running on the baseline static NUCA. We find that for these values of r and t with $s = 0.6195$, the reduction in execution time predicted by the model varies from 18% to 13%. It is very encouraging to note that these predicted values closely resemble the average reduction in execution time of the shared memory applications and the multiprogrammed workloads, as discussed in Section 6.1.

8. Summary

We have explored hardwired and OS-assisted locality management policies for large CMP NUCAs. Dynamic page migration emerges as the best policy in terms of execution time, energy consumption, and storage overhead for a set of shared memory parallel applications and multiprogrammed workloads running on an 8-core CMP with a shared 16 MB L2 cache. Compared to a baseline static NUCA, it reduces execution time by 18.7% (shared memory) and 12.6% (multiprogrammed), and energy by 13.9% (shared memory) and 10.8% (multiprogramed). This excellent performance comes at the cost of only 4.8% extra storage out of the total L2 cache and book-keeping budget. Although L1 cache prefetching succeeds in hiding a significant fraction of the on-chip latency in the shared memory applications, it fails to do so for the multiprogrammed workloads. L1 cache prefetching and dynamic page migration together save more than a quarter of the execution time for the shared memory applications.

In summary, the results of this paper clearly point to more detailed study into coarse-grain data migration in NUCAs. While a page appears to be a natural migration grain, a fixed grain may not be suitable for different working sets. An adaptive-grain migration policy that can switch between page-grain and block-grain over different execution phases may bring more benefit. In this paper, we have shown how to avoid cache flushes for application-directed page placement. This can be exploited in a light-weight application-directed dynamic page placement technique that migrates pages at barrier exit points or flag set points (i.e., a subset of release boundaries) where sharing patterns are likely to change in shared memory parallel applications.

Acknowledgments

The author thanks the Intel Research Council for funding this effort. Special thanks go to Gautam Doshi for useful suggestions throughout the course of this study. The author also thanks Sreenivas Subramoney, Sriram Vajapeyam, Arkaprava Basu, and the anonymous reviewers for useful feedback.

References

[1] V. Agarwal et al. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] K. Albayraktaroglu et al. BioBench: A Benchmark Suite of Bioinformatics Applications. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 2–9, March 2005.

[3] M. Awasthi et al. Dynamic Hardware-assisted Software-controlled Page Placement to Manage Capacity Allocation and Sharing within Large Caches. In *Proceedings of the 15th International Symposium on High-Performance Computer Architecture*, February 2009.

[4] J. D. Balfour and W. J. Dally. Design Trade-offs for Tiled CMP On-chip Networks. In *Proceedings of the 20th Annual International Conference on Supercomputing*, pages 187–198, June/July 2006.

[5] B. M. Beckmann and D. A. Wood. Managing Wire Delay in Large Chip-multiprocessor Caches. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 319–330, December 2004.

[6] B. M. Beckmann, M. R. Marty, and D. A. Wood. ASR: Adaptive Selective Replication for CMP Caches. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 443–454, December 2006.

[7] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-level Power Analysis and Optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 83–94, June 2000.

[8] R. Chandra et al. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–24, October 1994.

[9] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 264–276, June 2006.

[10] X. Chen and L-S. Peh. Leakage Power Modeling and Optimization in Interconnection Networks. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 90–95, August 2003.

[11] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance Associativity for High-performance Energy-efficient Non-uniform Cache Architecture. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 55–66, December 2003.

[12] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 357–368, June 2005.

[13] S. Cho and L. Jin. Managing Distributed , Shared L2 Caches through OS-level Page Allocation. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 455–468, December 2006.

[14] K. Flautner et al. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 148–157, May 2002.

[15] J. Gibson et al. FLASH vs. (Simulated) FLASH: Closing the Simulation Loop. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–58, November 2000.

[16] HP Labs. CACTI 4.2. Available at http://www.hpl.hp.com/personal/Norman_Jouppi/cacti4.html.

[17] J. Huh et al. A NUCA Substrate for Flexible CMP Cache Sharing. In *Proceedings of the 19th International Conference on Supercomputing*, pages 31–40, June 2005.

[18] R. E. Kessler and M. D. Hill. Page Placement Algorithms for Large Real-indexed Caches. In *ACM Transactions on Computer Systems*, **10**(4): 338–359, November 1992.

[19] C. Kim, D. Burger, and S. W. Keckler. An Adaptive Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, October 2002.

[20] R. Kumar, V. V. Zyuban, and D. M. Tullsen. Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads, and Scaling. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 408–419, June 2005.

[21] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[22] Micron Technology Inc. DDR2 Offers New Features and Functionality. *Micron Technical Note TN-47-02*.

[23] Micron Technology Inc. Calculating Memory System Power for DDR2. *Micron Technical Note TN-47-04*.

[24] L. Noordergraaf and R. van der Pas. Performance Experiences on Sun's Wildfire Prototype. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, November 1999.

[25] R. M. Rao et al. Efficient Techniques for Gate Leakage Estimation. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 100–103, August 2003.

[26] S. Rusu et al. A Dual-core Multi-threaded Xeon Processor with 16 MB L3 Cache. In *Proceedings of the International Solid State Circuits Conference*, February 2006.

[27] SGI Technical Publication. Origin2000 and Onyx2 Performance Tuning and Optimization Guide. *Document Number 007-3430-002*. Available at http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi/0640/bks/SGI_Developer/books/OrOn2_PfTune/.

[28] T. Sherwood et al. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support on Programming Languages and Operating Systems*, pages 45–57, October 2002.

[29] E. Speight et al. Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 346–356, June 2005.

[30] Tital Corporation. Data Intensive Systems (DIS) Benchmark Performance Summary. *Air Force Research Laboratory Technical Report AFRL-IF-RS-TR-2003-198*, August 2003.

[31] B. Verghese et al. Operating System Support for Improving Data Locality in ccNUMA Compute Servers. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–289, October 1996.

[32] M. Zhang and K. Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 336–345, June 2005.