Leveraging On-Chip Networks for Data Cache Migration in Chip Multiprocessors

Noel Eisley
eisley@princeton.edu
Dept. of EE
Princeton University
Princeton, NJ 08544

Li-Shiuan Peh
peh@princeton.edu
Dept. of EE
Princeton University
Princeton, NJ 08544

Li Shang
li.shang@colorado.edu
Dept. of ECE
University of Colorado
Boulder, CO 80309

ABSTRACT

Recently, chip multiprocessors (CMPs) have arisen as the *de facto* design for modern high-performance processors, with increasing core counts. An important property of CMPs is that remote, but on-chip, L2 cache accesses are less costly than off-chip accesses; this is in contrast to earlier chip-to-chip or board-to-board multiprocessors, where an access to a remote node is just as costly if not more so than a main memory access. This motivates on-chip cache migration as a means to retain more data on-chip. However, previously proposed techniques do not scale to high core counts: they do not leverage the on-chip caches of all cores nor have a scalable migration mechanism. In this paper we propose ascalable in-network migration technique which uses hints embedded within the router microarchitecture to steer L2 cache evictions towards free/invalid cache slots in any on-chip core cache, rather than evicting it off-chip. We show that our technique can provide an average of a 19% reduction in the number of off-chip memory accesses over the state-of-the-art, beating the performance of a pseudo-optimal migration technique. This can be done with negligible area overhead and a manageable traffic overhead of 13.4%.

Categories and Subject Descriptors: C.2.1 [Network Architecture and Design]: Packet-Switching Networks; C.2.2 [Network Protocols]: Protocol Architecture (OSI Model)

General Terms: Design, Theory

Keywords: Chip-multiprocessor, CMP, Interconnection network, NoC, Migration, Network-driven computing

1. INTRODUCTION

Chip multiprocessors were first proposed more than ten years ago [17] as a design alternative to combat the increased complexity and diminishing marginal performance gains experienced by superscalar uniprocessors. Since then, as device scaling has continued on its historic trajectory, CMPs have become the *de facto* design for modern high-performance processors, with many commercial CMPs available [22, 24, 25]. Yet CMPs have their own challenges and tradeoffs and research into CMPs is still in the early stages. In particular, many of the design choices which have been adopted by early CMP designers work well on 2 or 4-core CMPs, but they face problems when scaled to tens or even hundreds of cores.

Of particular importance is that, in moving a multiprocessor to a single chip, the relative cost of accessing main memory, as compared to a remote node's cache, increases by approximately one to two orders of magnitude. The number of off-chip accesses will worsen as CMPs scale to higher core counts, particularly if the number of cores grows more quickly than the total on-chip storage (*i.e.* the per-core cache shrinks), thus causing increased local cache pressure and evictions. This is worsened by the constrained bandwidth that can be delivered by current off-chip I/Os. We are hitting a limit in the amount of I/O bandwidth that can be supplied to a chip due to limits in pin count and electrical signaling bandwidths [2]. While alternative technologies exist, such as optical I/Os and 3D stacking, they still face significant challenges before practical deployment. As a result, it is beneficial to keep as many cache lines as possible on-chip; migrating evicted cache lines from one on-chip node to another is one way to accomplish this goal.

The key to cache line migration lies in the judicious choice of a target node for migrating the evicted line to. Ideally, such a target node will have available space for the evicted line so the migration does not trigger additional cache evictions; and the target node will be placed close to future accesses so even on-chip access time is minimized. The choice of such a target node relies on up-to-date information about L2 cache occupancy. To explore the potential of cache line migration, we simulated a pseudo-optimal migration strategy where the cache states of all on-chip L2 caches are known perfectly, and the nearest node with an invalid cache line slot to which the specific evicted L2 cache line would map is selected as the target node for each L2 eviction. Figure 1 shows that with such a pseudo-optimal strategy, cache line migration can attain an average of a 26.8% reduction in the number of off-chip main memory accesses, over the next-best configuration, in a 64-node CMP (see Section 4 for detailed experimental setup).

In recent years, there has been a flurry of research into alternative techniques in on-chip cache migration (see Section 2). However,

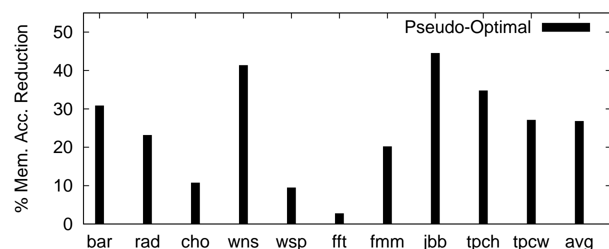

Figure 1: Potential main memory access reduction.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PACT'08, October 25–29, 2008, Toronto, Ontario, Canada.
Copyright 2008 ACM 978-1-60558-282-5/08/10 ...$5.00.

these techniques do not lend themselves well to future many-core chips, either because they do not use all remote caches as migration targets, losing out substantial opportunities in future many-core chips, or because their implementations do not scale beyond high tens or hundreds of cores. This motivated our research: a cache migration technique that is highly scalable and leverages the entire on-chip capacity of many-core chips. A unique aspect of our proposal is that we implement it *within* the on-chip network fabric: upon a cache line eviction, the cache line will be injected into the network, which will not route the line towards a specific destination as in conventional on-chip networks, but will instead route the line towards a remote L2 cache with an available cache line slot. In essence, each router keeps track of L2 cache occupancies of immediate neighbors, and the routing algorithm uses this information to steer the evicted line towards appropriate router ports.

Our simulation results show in-network migration reducing average memory access latencies over state-of-the-art techniques; in Section 4 we see that the 19% reduction in off-chip accesses leads to a 6.0% average read latency reduction compared to a strictly private L2 cache scheme (L2P) in 64-core CMPs. This is achieved without impacting router pipeline delay, at negligible area overhead (of just 0.1% of on-chip cache area), and with a manageable traffic overhead of 13.4%.

The rest of this paper is organized as follows. In Section 2, we present background information and discuss related work. In Section 3 we describe our in-network migration scheme in detail, its mechanics and implementation details, and design-space considerations. In Section 4 we present our results and discussion and finally we conclude in Section 5.

2. BACKGROUND AND RELATED WORK

2.1 Shared and private L2 caches

While there have been dance-hall CMP architectures with processing elements on one side of the chip and a monolithic shared L2 cache on the other, such CMPs have only two, four, or eight nodes [15, 9, 25] and face challenges in scaling up to higher core counts. This is because minimum L2 hit latency increases with CMP core count, with request messages having to pass through a number of routers before even reaching the large L2. Hence, for CMPs with higher core counts, grid-like CMP architectures where each core has its own L1 and L2 caches are more appropriate. In such architectures, there are basically two ways of organizing the L2: shared or private.

Shared L2 Caches: L2S. In the L2S configuration, each local L2 cache behaves as one slice of a monolithic shared L2. Logically, the behavior is identical to a CMP which has a single L2 NUCA [11]. Each node serves as the home node for a statically divided portion of the physical address space. Added to each L2 cacheline is a bit vector which specifies which nodes are actively caching copies of that data. Upon an off-chip memory access, the requested data is first stored at the L2 cache of the home node for that data, and then it is forwarded on to the L1 cache of the requesting node. In addition, cachelines which are evicted from L1 caches are not retained at the local L2 caches, but instead are sent back to the home nodes to update the sharing bit vectors. As a result, any line of data can exist in only one on-chip L2 cache and as many on-chip L1 caches as are actively using that line.

Because L1 evictions are not retained in the local L2 caches, the number of L1 misses which ultimately hit in remote L2 caches is relatively high. A remote L2 hit is more costly than a local L2 hit, yet the L2S configuration usually results in fewer off-chip memory accesses, since no cache line is replicated in more than one L2, and therefore more unique cache lines can be kept on-chip at any time.

Private L2 Caches: L2P. In the private L2 configuration, the same cache line can be replicated at any number of L1 *or* L2 caches. In order to do this, the bit vectors used to maintain coherence are separated from the L2 cache and maintained in their own *directory cache* structure. Here the cache coherence protocol encapsulates both the L2 and L1 caches as one, such that, if a directory cache entry indicates that a particular cache line is shared by Node A, then that line may exist either in Node A's L2 or L1 cache. As in L2S, when an off-chip access returns from main memory it first goes to the home node so that the sharing list can be updated, but the line is *not* stored at the home node's L2. Also as in L2S, the data is then forwarded to the requesting node and written to its L1 cache. However, unlike in L2S, when an L1 cache line is evicted, it is retained by the local node's L2. Only when it is evicted from the L2 is an eviction message sent back to the home node and the sharer removed from the directory cache entry.

L2P's main advantage over L2S is that the local L2 hit rate is much higher since L1 evictions are retained at the local node. While the off-chip memory access rate is higher due to the duplicate L2 cache lines, this effect is usually dominated by the first. As a result, the overall performance of L2P is usually significantly better than L2S, as has been shown in previous work [20, 6] and here in Section 4 as well.

2.2 Home node migration and replication

Victim replication (VR) [20] is a recent work, by Zhang and Asanovic, which is a type of hybrid configuration between L2S and L2P, but is derived from L2S. Just as in L2S, when data is brought from off-chip, it is written to the home node's L2 cache and forwarded on to the requesting node's L1 cache. The difference is in what happens when there is an L1 eviction. In VR, the local node attempts to retain a copy of the data in its L2. However, this attempt is not always successful. A victim copy may overwrite an invalid L2 line, another victim, or a global line (*i.e.* a line for which this node is the home node) with no sharers, in that order of preference. If no such lines exist then the attempt is abandoned, and an eviction message is sent back to the home node, as in L2S.

VR's main advantage over L2S is that many of the L1 evictions are retained by the local L2, and therefore many future accesses will hit in the local L2 rather than in a remote L2. However, its main drawback is that there are many redundant L2 copies in the network. Even if a line of data is not shared by more than one node, it will exist in two nodes (unless the sharer node is the same as the home node). Thus, the effective total L2 capacity is almost halved, which leads to more conflicts and hence more off-chip memory accesses. In Section 4, we see that with a modestly sized L2 cache per node, VR actually suffers a 32% memory access latency penalty as compared to L2P.

Follow-on work, Victim Migration (VM) [21], addressed this effective total caching restriction by removing the duplicated cache lines at the home node's L2. VM duplicates the directory structures at each node; now there are directory entries (bit vectors) embedded in the L2 cache (as in L2S), as well as a separate directory cache along with duplicated tag arrays (as in L2P). Thus, an on-chip copy of data is only cached at its home node if there are no active nodes using the data *and* there is room for it at the home node's L2 cache. Again the motivation is that retaining data on-chip when there is space available may prevent a future off-chip main memory access.

Another hybrid between L2S and L2P is Adaptive Selective Replication (ASR) [1]. Like VR, ASR attempts to keep evicted L1 cachelines in the local L2 caches. However, Beckmann, Marty, and Wood make the observation that a particular replication policy's effectiveness is a function of both address and time. Thus, ASR makes each individual replication decision based on a marginal cost/benefit analysis. Relevant statistics are kept which are used to calculate these marginal values. This adaptive behavior allows ASR to outperform VR [20]; however, ASR still limits on-chip data to either the caches of an active sharer or of its home node. Our approach permits data to reside in *any* on-chip L2 cache.

2.3 Centralized vs. distributed cache line migration

Our proposal is a distributed approach to approximate the pseudo-optimal migration described in Section 1. A centralized approximation can be realized using a technique such as Cooperative Caching for CMPs [4] as described by Chang and Sohi. By duplicating the tag arrays of each of the nodes' caches in a centralized structure and maintaining directory information there, decisions can be made using global knowledge about which cache lines to replicate or migrate, and to which nodes. However, the work by Chang and Sohi assumes a CMP with just 4 nodes, making such a centralized structure practical. In this work we explore CMPs with 16 and 64 nodes, with the belief that even more may be possible in the future. There are two factors which prohibit the scalability of such a centralized design. The first is that all memory traffic which enters the network must travel through the directory. This is likely to cause an artificial traffic hotspot at the centralized directory. The second reason is that in order to make a decision as to where to place a particular line of data, each of the directory structures (one for each node) must be accessed, and the results compared. So as the number of cores in the CMP increases, the number of comparisons also increases, and in turn the latency of each decision increases. As we show in Section 4, the performance of our distributed approach comes very close to such a centralized structure.

CMP-NuRapid [6] is a more distributed approach which allows replica cachelines to exist not only in the closest bank of an on-chip L2 cache (*i.e.* the "local" L2), but any other as well. As cachelines are evicted from local banks, they are successively moved further away until there is no room on-chip. A node will also only initially replicate an on-chip copy if it makes at least two requests for that data. However, the evaluation of this scheme is also limited to CMPs with a small number of nodes because CMP-NuRapid assumes not only a fully-connected crossbar between each node and each bank of the centralized, on-chip L2 cache, but also a memory bus on which each processor snoops (through their private L2 tag arrays). Requiring bus broadcasts and snooping protocols severely limits the scalability of such a scheme.

2.4 In-network code and data management techniques

Implementing data management functionality in or near the network fabric has been investigated in a number of contexts in the past. To our knowledge, the first work to integrate cache coherency with the network layer was that by Mizrahi *et al.* [16]. There, the authors implement the entire L2 caches within the switches which is feasible in a chip-to-chip or board-to-board multiprocessor, but not in a CMP. Furthermore, their protocol dictates that only a single copy of any cacheline can exist in the network at any time and do not address the capacity/latency tradeoff of replication. Other works which implement in-network data management functionality include that by Iyer, *et al.* [12], who proposed maintaining just the cache coherence directories in the network layer; the Wisconsin Multicube [8], which utilized in-transit cacheline snooping; GLOW [13], which used a tree structure embedded within the network interface layer in order to optimize cache coherence behavior; and In-Network Cache Coherence [7], which combined the in-transit optimization opportunities of a tree-based structure with the low-latency redirection of in-network implementation in order to achieve lower network memory access latencies. All of the above do not address migration or replication, however. Finally, others have explored implementing functionality within the network fabric of CMPs in order to perform hardware/software co-optimization of thread management [5]. In general, as CMP core counts increase, more and more functionality may be embedded within the network fabric in order to address scalability concerns.

3. NETWORK MIGRATION

3.1 Overview

The thesis of our proposed approach is simple: when a cache line is evicted from the L2 cache of a particular node, instead of either evicting the line off the chip entirely, or migrating it back to the home node, we will try to migrate the line to another on-chip node. We use hints embedded in the network fabric to adaptively steer the migrating cache lines towards underutilized L2 caches. Based on these hints, at each hop a migrating cache line makes the decision to either eject from the network and attempt to relocate to the L2 cache of the current node, or to continue on and seek another target node which is more likely to have room for it (see the walkthrough below). The hints are updated periodically through messages from immediate neighbors and are based on estimates of L2 cache utilization in a particular direction. If a migration succeeds at a particular node, then a message is sent to the home node to update the sharer list, and an acknowledgement is sent back to the migrant's new node to finalize the migration. If the migration does not succeed, then an eviction message is sent back to the home node and the cache line is discarded as if the migration had not been attempted in the first place.

3.2 Mechanics

3.2.1 Network messages

Throughout this paper, we assume a standard directory-based MESI protocol [10], but our approach can be applied to any directory protocol. We do not modify the cache line state diagram; we only add a few message types to support migration. Table 1 summarizes the messages used in our scheme, with our deviations from the standard protocol shown in bold and described in more detail below:

- `MIGRATE` . A `MIGRATE` message encapsulates the evicted data cache line and adaptively routes through the network until it finds a suitable node for relocation. The adaptive routing algorithm is constrained such that deadlocks are prevented.
- `MIGREQ` . Once the `MIGRATE` message has found a suitable node, it is morphed into a `MIGREQ` message within the router and is sent to the home node for the particular cache line. At the home node, a `MIGREQ` message triggers a lookup in the directory cache. If no entry exists, then it must have been evicted during the migration, and therefore the `MIGREQ` message simply deletes itself, effectively terminating the migration attempt. If the directory entry still exists, the new node is added to the sharer list, and the old node is removed.
- `MIGREPLY` . After the `MIGREQ` message successfully adds the new sharer to the directory entry, it is again morphed into a `MIGREPLY` message which routes back to the new node. When it reaches its destination, the data is written to the local node's L2 and the migration is complete.

Note that it is possible that during the request/reply, the local L2 may no longer have an available (invalid) line. If this happens there are two possible actions. First, the migrant can still copy itself to the L2, evicting a new line in the process; second, the `MIGREPLY` message can delete itself and send a new `EVICT` message to the home node so that the new sharer can be deleted from the directory entry. We choose the former because of its simplicity; there is minimal difference in the performance of the two because on average, fewer than 1% (and often fewer than 0.1%) of migrations run into this conflict.

3.2.2 Implications to cache coherence

It is important that the addition of the `MIGRATE, MIGREQ and MIGREPLY` messages do not break the coherence provided

Table 1: Protocol messages.

Message	Description
READREQ	Initial read request message. Always routes towards home node
READREPLY	Reply message sent by home node or remote sharer to the requesting node
WRITEREQ	Initial write request message. Always routes towards home node
WRITEREPLY	Reply message sent by home node to the requesting node
INV	Invalidation message sent from home node to sharers. Invalidates L2 and L1 copies
INVACK	Invalidation message acknowledgement sent back to the home node
EVICT	Message sent to the home node when an L2 cache line is evicted due to conflict Similar to INVACK, but not caused by INV
MIGRATE	**Message which encapsulates evicted data and searches for a new node for it**
MIGREQ	**Message sent from migrant's new node to home node to add new node to sharer list**
MIGREPLY	**Message sent from home node to migrant's new node to finalize migration**

by a standard directory protocol. Given that the baseline protocol is coherent, then if the atomic behavior of the read and write requests of our protocol are equivalent to those of the baseline, then we can be sure that the property of coherence is maintained.

The concern arises from the metastate[1] of the aggregate on-chip caching caused by the migrations. During a migration, a cacheline is evicted from one node and in transit to another; yet at the home node, the directory entry points to the original node as one location where it can be found. But this scenario can actually occur in the baseline protocol as well. When a cacheline is evicted, the data is sent back to the home node, but the home node still points to the node from which it was evicted until that message reaches the home node. This is not a problem because any read which is forwarded to the sharer node will fail upon reaching it (finding that there is no copy there) and simply retry (route to the home node again). So the two are equivalent in this respect. Additionally, the migrated data must not be read until the new node is pointed to by the directory entry at the home node. We ensure this property by not committing the migration until a message has been sent to the home node to update the sharer list and a message has been sent back to the new node acknowledging that it is now able to commit the migration.

3.2.3 Network hints

As addressed briefly above, the behavior of the MIGRATE message is guided by hints, or scores, embedded in a table in the network pipeline architecture, as depicted in Figure 2. At each node, each scores table entry contains one score for the local node's processing element (PE score), and one score for each outgoing link (link scores). The PE score gives an estimate of the utilization of a portion of the local node's L2 cache to which the given migrant's address maps (the addresses-to-entry mapping is many-to-one as seen in Figure 3). The link scores give rougher estimates of the L2 cache utilizations of the nodes in the direction of that link, not just the immediate neighbor. In addition, the scores table is indexed by a number of consecutive bits from the memory address, such that $Score_{PE,N,S,E,W} = f(addr)$. These bits are a subset of the same bits used to index into the L2 cache and are depicted in Figure 3 as the shaded region of the address. In order to keep the information in the score tables up-to-date, the PE scores are periodically updated directly by snooping the reads and writes to the local L2 cache, and the link scores are periodically updated by their neighboring nodes. The microarchitectural implementation for updating and propagating the link scores values is shown in Figure 3. When updated, a link score is a function of its neighboring node's scores (using the

9 [1]Not to be confused with the actual MESI state of each individual cacheline.

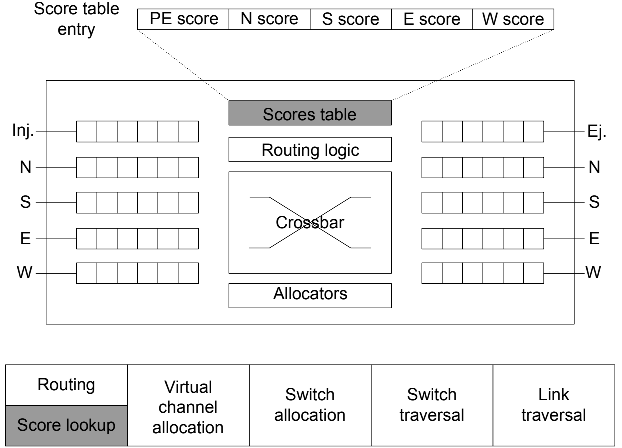

Figure 2: Proposed router architecture, highlighting the addition of the scores table and the table lookup stage, performed in parallel with the routing stage. In order to accomodate simultaneous reads from any combination of input queues, five separate read ports are required. However, since updates are infrequent, a single write port with a small queue is sufficient.

east link to be updated as an example):

$$S_{N_A,E}(a) = f(S_{N_B,PE}(a), S_{N_B,N}(a), S_{N_B,S}(a), S_{N_B,E}(a)). \quad (1)$$

where N_A and N_B are two neighboring nodes, and $S_{N_x,y}(a)$ is the score of type y at node x for address a. While there are an infinite number of possible functions of this type, we want our function to have two important properties:

1. The neighboring PE score should be the dominant term.
2. The link scores should have significant and equal (to each other) influence.

For these reasons, we define the link score function as

$$S_{N_A,y}(a) = 0.5 \cdot S_{N_B,PE}(a) + \frac{1}{6} \cdot \sum_{w}^{w \in \{N,S,E,W\}, w \neq \overline{y}} S_{N_B,w}(a). \quad (2)$$

where $\overline{y}$ is the (directional) opposite of y (*e.g.* N is the opposite of S), and the limitation on w arises from the fact that a migrant can never route back on its own path.[2] The coefficients were chosen such that the PE and links (collectively) have equal weight, and

9 [2]Observe that there is no term dependent on the west link score

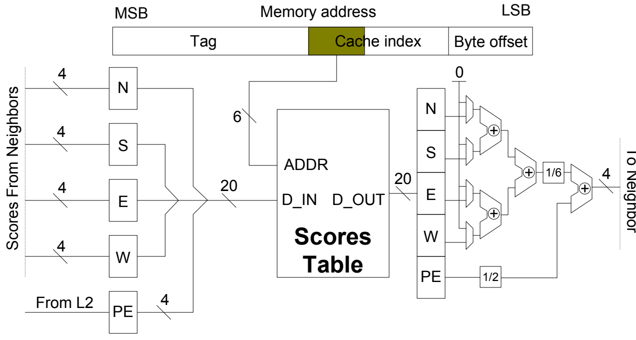

Figure 3: Scores Table updates. On the right side, the individual scores from each entry (depicted vertically) are weighted and summed, according to Eq. 2. The result is sent to the adjacent node, either encapsulated in a small message or sent over a dedicated link. The logic can either be replicated for each neighboring node to generate each score in parallel, or multiplexors can be utilized to generate each score in series (we assume the former). On the left side, the scores are collected from each neighbor, as well as the local L2, and are aggregated when writing to the scores table.

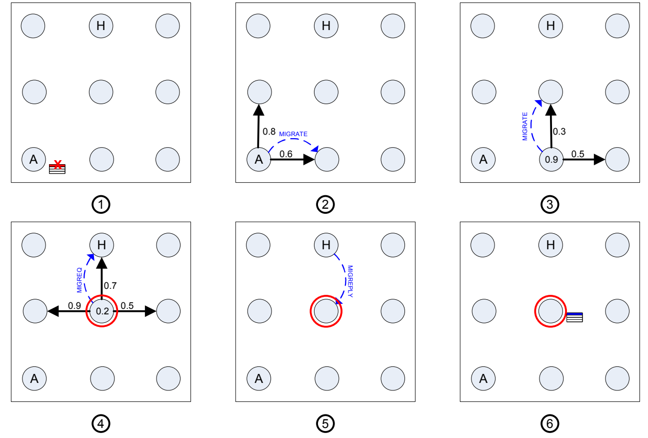

Figure 4: Walkthrough showing a complete migration operation in a 3x3 CMP. Scores of PEs and links are shown alongside the specific PEs and links. The node chosen to be the target for migration is highlighted with concentric circles. Physical links are not shown except when the link's score is in consideration. The threshold PE score in this example is 0.6.

the link scores each have equal weights since the network is symmetrical in both dimensions and since the memory space is evenly distributed among the cores. In this way, information about the utilization of each L2 cache propagates through the entire network, although diluted at each hop, encouraging the migrants to seek new homes closer to their original nodes, based on more accurate estimates. This method of information propagation also allows for it to be done in a way which does not require messages to traverse more than one link; this simplifies the implementation greatly as we describe in our discussion of the overhead in Section 3.5.

As can be seen from Figure 3 and Equation 2, the east link score for N_A is calculated at N_B and then transmitted to N_A. The link scores are only sent between immediate neighbors (*i.e.* each link score traverses exactly one physical link), which permits scores table update optimizations and tradeoffs, as discussed below:

- **Dedicated links.** One obvious way to perform the link score updates is to encapsulate them in standard network packets and to transmit them through the already existing output and input buffers, arbitrating for the links with all other network traffic. However, since the updates are only neighbor-to-neighbor, the updates can traverse dedicated links between nodes. By using separate links, buffers are not required and the update messages do not affect the regular network traffic.
- **Load balancing.** The scores tables need not be updated in consecutive cycles. If on average, each scores table is updated once every 10,000 cycles, but it takes just 100 cycles to send the necessary link scores from one node's table to a neighboring node's table, it can be accomplished by sending one line every 100 cycles, distributing the updates across time. This has a more significant benefit if update messages share links with regular network traffic, minimizing the impact to congestion, but even with dedicated links there is a potential power/temperature benefit to this type of load balancing.
- **Greedy updates.** Another possible congestion and energy-minimizing approach would be to not calculate and update link scores if the local PE score has not changed since the last update. A single bit in each PE score can be set whenever one of the lines of the L2 cache which that score covers is modified, and it can be unset whenever that PE score is used to calculate new link scores for the neighboring nodes. In this way, fewer cycles will be spent transmitting updated link scores.

These are just a few of the possible variations in an actual implementation of link score updates. In our simulations for this work, the details of the updates are transparent (except for the frequency), so we do not quantitatively evaluate them here.

3.3 Walkthrough Example

To tie everything together, we present a walkthrough of a typical migration, shown in Figure 4. Each step is described in detail below:

- **Step 1:** A cache line is evicted from the L2 cache of node A.
- **Step 2:** A `MIGRATE` message is created and populated with the evicted cacheline. All outgoing links' scores are compared, and the east link has the lowest score, so the `MIGRATE` message routes in that direction.
- **Step 3:** At the next node, the PE score is higher than the threshold, so again the lowest link score is selected (notice that the west link is not specified because the message is not allowed to turn back on itself). In this case, the north link promises the best chance for a vacant cache line so the `MIGRATE` message routes in this direction.
- **Step 4:** At the next node, the PE score is lower than the threshold, so the `MIGRATE` message attempts to make this node the new location of the evicted data by creating a `MIGREQ` message and routing it towards the home node for this cache line address. At this point the cacheline remains in a queue in the network interface of the migrant's new node and it is not sent along with the `MIGREG` message. If the queue were full, the migration would be aborted at this point.
- **Step 5:** At the home node, the `MIGREQ` message removes node A from the directory entry's sharer list, and adds the new node to it. A `MIGREPLY` message is then created and sent back towards the cache line's new node.

of node B in Equation 1.

- **Step 6:** Finally when the MIGREPLY message reaches its destination, it finds the waiting cacheline in the queue by comparing against all the addresses, the cache line is then written to the L2, and the migration is finalized.

3.4 Design Space

While the design space of our proposed scheme is essentially limitless, we highlight three parameters which we explore later in Section 4; these parameters are important because as we see below, they have a direct effect on the overhead of our design. Afterwards, we briefly discuss some of the other possible variations and their likely effects.

Hash function or set precision (number of entries). The precision of the hash function depends on the number of entries in the scores table. At one extreme, there is an entry for every set in the L2 cache. In this case, any PE score less than 1.0 actually translates to a 100% chance of there being an invalid line. As an example, consider the case where the L2 is 4-way set associative. The PE score for a set will be 1.0 only if all four entries are used (valid). So if the score is less than 1.0, at least one entry is invalid. Since any address which maps to this set can occupy any of the set's cache lines, then any MIGRATE message which maps to this set will be guaranteed an invalid line. At the other extreme, there is only a single PE score for the entire L2 cache (zero index bits to the scores table in Figure 3). In this case, the PE score approximates the actual probability that the L2 cache will have an invalid line to which a MIGRATE message would map, assuming a uniformly randomly generated address. Since the PE score is more informative and deterministic the more entries there are, we expect that performance will be better the more entries there are in the scores table.

Score precision (number of bits per entry). The precision of each score depends on the number of bits representing that score. For example, suppose there is just a single PE score representing the entire L2 cache which has 4K entries. Full precision requires 12 bits; in this case the exact number of invalid cachelines, but nothing about their distribution, is known. However, given that the distribution is not known, perhaps it is just as effective to know whether the L2 cache is 0, 25, 50, or 75% full, which only requires 2 bits for the score. We expect that, as alluded to in the above example, the score precision will have a larger effect on performance the more entries there are in the scores table.

Update frequency. It is straight forward that the frequency with which the link scores are updated directly affects the staleness of the scores. What is unclear is the quantitative effect of scores' staleness effect on performance. We explore this in Section 4.

3.5 Overhead

There are two types of overhead which our design introduces: area overhead in implementing the score tables, and traffic overhead. The traffic overhead can further be divided into extra traffic due to the actual migrations (MIGRATE, MIGREQ, and MIGREPLY messages) and traffic due to score table updates.

Area overhead. We will use the storage space (in bits) required as a proxy for estimating the area overhead of the scores table. Since there are two design space axes which affect the bits overhead, as discussed above, we can express the number of bits required for our design as a function of the number of entries per score table (set precision), as well as the number of bits per entry (score precision). Quite simply, the number of bits per node is

$$N_{bits}\left(N_T, N_S\right) = 5 \cdot N_T \cdot N_S, N_T \in \{1, 2, 4, ..., N_{L2}\} \quad (3)$$

where N_T is the number of entries in the score table, N_S is the number of bits per entry, the range of N_T is powers of 2 from 1 to N_{L2}, the number of entries in the L2 cache, and the factor of 5 is a result of our assumed mesh topology where there is one PE score and four link scores. However, it is interesting to define the number of bits required not in terms of bits per score, but in terms of precision. If we assume the maximum precision[3], then we can express N_S in terms of N_T, and we have

$$N_{bits}\left(N_T\right) = 5 \cdot N_T \cdot log_2\left(\frac{N_{L2}}{N_T}\right), N_T \in \{1, 2, 4, ..., N_{L2}\} \quad (4)$$

Varying the score precision is achieved by using fewer bits per score, though it is convenient to parameterize the desired precision, p (maximum, half, quarter, *etc.*), in order to visualize the family of curves of Eq. 4 above:

$$N_{bits}\left(N_T, p\right) = \begin{cases} 5 \cdot N_T \cdot log_2\left(\frac{N_{L2}}{N_T} \cdot p\right) & if N_T < p \cdot N_{L2}, \\ 5 \cdot N_T & if N_T \geq p \cdot N_{L2}. \end{cases}$$
$$for\ N_T \in \{1, 2, 4, ..., N_{L2}\}, p \in \{1, \tfrac{1}{2}, \tfrac{1}{4}, ..., \tfrac{2}{N_{L2}}\}.$$

With our baseline design (see Table 2), the scores table requires just 1280 storage bits per node. This is compared to more than 1M storage bits which comprise each L2 cache, or less than 0.1% of the area of the L2 cache. Furthermore, this figure compares favorably with other proposed in-network techniques. For example, the virtual tree caches of Eisley *et al.* [7] require more than 100K bits per node. So the size of the scores table does not lead to a significant area overhead.

Delay overhead. Since the area is approximately 1% of that of the tree caches, cited above, which take two cycles to access, we expect that a score table lookup can be readily achieved in one cycle. This is especially true since no tag comparisons are required. Simulations with Cacti [18] confirm that even assuming a much larger 1K entry scores table with 2 bits per score (for a total of 10 bits per entry), the data lookup can be performed in less than half a cycle, assuming 500MHz clock frequency at the $0.18\mu m$ technology point. Hence, this lookup can be done in parallel with the routing stage of a typical router pipeline, as shown in the bottom of Figure 2, and does not impact the router pipeline length. Even with a larger scores table than assumed here, or more aggressive clock frequencies, if the lookup takes multiple cycles, this will not significantly impact performance, since the only messages which make the table lookup are MIGRATE messages, and migrations are not on the critical path of memory accesses.

Traffic overhead. For the set of benchmarks described in Section 4, we measured the breakdown of all the traffic in the network by message type. MIGRATE, MIGREQ, and MIGREPLY combine to account for 7.5% of the total traffic, on average. So the migration messages themselves do not substantially increase the total traffic load on the network. In fact, even with migration, the average link utilization across the entire CMP and across time is approximately 8%, well below the point where saturation would be of concern. When deployed in many-core chips that demand high bandwidth, these messages can be assigned low priorities so they do not affect more critical coherence traffic.

A more interesting tradeoff occurs with the increased traffic caused by the propagation of the link scores from node to node. If we denote by P the number of phits[4] it takes to transmit a single score table (or, just one link of the score table, since each portion is transferred, but just one portion to each neighboring node), by L the number of links (and therefore link score tables), and by f the interval, in cycles, between updates, then the total network bandwidth used is equal to

$$BW = \frac{L \cdot P}{f} \quad (5)$$

9 [3] for an L2 cache with 4K entries and 4-way set associativity, the maximum precision for a 1-entry scores table would be 12 bits (since each score is covering 4K L2 entries), and the maximum precision for a 1K-entry (one entry per L2 set) scores table would be 2 bits per entry (since one score would be covering 4 L2 entries).

9 [4] A phit is the amount of data which can be transfered across a physical link in a single cycle.

If we assume 64-bit phits, and 48 links for a 4x4 mesh CMP, then Eq. 5 can be simplified to

$$BW = \frac{0.75 \cdot N_{bits}}{f} \frac{phits}{cycle} \quad (6)$$

where N_{bits} is given above. Assuming a maximum table size of 2K bits per link, sending updates every 100,000 cycles results in an additional 12% traffic overhead, on average, while a more appealing 1.2% overhead mandates updating only every 1,000,000 cycles. For scores tables as implemented in Table 2 (256 bits per link), which take 4 cycles to transfer an entire link's portion of a score table from one node to its neighbor, then an update interval of 100,000 cycles results in a traffic overhead of only about 1%. We explore this design parameter further in Section 4 to see how much performance is dependent on the frequency of updates.

4. RESULTS

Our experiments are run on a trace-driven cycle-accurate CMP network and cache simulator which we implemented for this work. Each node is modeled with a two-level cache, and the size, associativity, and cache line length of each level is separately parameterizable. The traces represent memory accesses which come from the processing element pipelines. The interconnection network is modeled in detail: cache coherence messages are modeled discretely and on a cycle-by-cycle basis; contention is captured by modeling the crossbars in the routers as well as the input port queues; and although each stage of the router pipeline is not modeled in detail, the pipeline length is modeled. The parameters of our baseline configuration are given in Table 2. The address space is statically divided among all the nodes, using the least significant bits of the tag, to determine which node is the home (directory) node for a given address. Analysis shows that this distribution is quite even. We chose to use a trace-driven simulator, rather than implementing our idea into an execution-driven simulator for tractability. Even in our simulator, some of the longest traces take twelve hours to complete in a 16-node CMP. As a consequence of trace-driven simulation, however, it is less reliable to report an overall measure of execution time, *e.g.* simulation cycles. So in this work we focus on average memory access time and the number of off-chip memory accesses, although even in our simulator the reduction in simulated cycles is proportionate to the reduction in average memory access time.

Some of the memory access traces[5] are generated by running a set of SPLASH-2 benchmarks [23] in Simics [26]. The configuration for Simics for trace generation is a 16-node CMP arranged in a 4x4 mesh topology. The other traces[6] are generated by running these commercial benchmarks on PharmSim [3], an execution-driven simulator developed at the University of Wisconsin. These are also configured to run in a 16-node CMP arranged in a 4x4 mesh.

The simulator which we implemented tracks memory access latencies, both reads and writes. The latency includes node-to-node routing, queueing delays, directory cache contention delays, directory cache access time, remote cache access time, and off-chip memory access time. An off-chip memory access is modeled in the following way. First, a message is sent from the current node to the nearest edge of the chip (*i.e.* to the nearest pin) at which point the message is removed from the network and a return message is injected back into the network after some number of cycles representing the time to actually access the memory. Simulated I/O bandwidth was unlimited, as we use the number of off-chip memory accesses as a measure of I/O pressure.

We ran the 16 traces for each benchmark in 4x4 simulated CMPs

Table 2: Simulated memory-network configuration.

Baseline Data Cache Configuration	
Line size	8 words (32 Bytes)
Eviction policy	LRU (see Section 2)
L2 data cache size (baseline)	4K entries (128KB)
L2 data cache associativity	4-way
L2 data cache access latency	6 cycles
L1 data cache size	256 entries (8KB)
L1 data cache associativity	Direct-mapped
L1 data cache access latency	1 cycle
Baseline Network-Memory Configuration	
Router pipeline	3 cycles
Routing algorithm	X-Y routing
Main memory access latency	200 cycles
Baseline Directory Cache Configuration	
Entries	4K
Associativity	16-way
Access latency	2 cycles
Eviction policy	LRU
Baseline Network Migration Configuration	
Entries	64
Link score update interval (cycles)	100000
PE score threshold	0.4
Bits / entry	2

as well as 8x8 CMPs. For the 8x8 configurations, the traces were run in one quadrant of the CMP (experiments were also run with the traces in the central 4x4 section of the CMP, with approximately equal results). As the number of transistors per unit area increases, and thus the number of cores per chip increases, this is a likely scenario. The peak power consumption of four times as many cores will necessitate limiting the number of active cores (L2 caches can remain active while their respective cores are inactive) except for short periods).[7]

4.1 Performance and Scalability

We implemented the configurations described in Section 2: L2P, L2S, our network migration (NM) scheme, and victim replication (VR) [20] for comparison. In addition, we implemented two variants of our NM scheme to function as oracles or yardsticks for comparison:

- Pseudo-optimal migration ("**Opt**"). When a cache line is evicted from L2, the migrant's destination is selected as the nearest node which contains an invalid line to which the migrant address maps. It is pseudo-optimal because even though global information is used at the start of the migration, it is possible that the node selected may not have any invalid lines in the applicable set once the migrant actually reaches the node.
- Random migration ("**Rnd**"). At each node, a migrant randomly picks a direction and travels one hop. With 50% probability, the migrant will either eject into the local node and *attempt* to migrate to the current node or not, at which point it will again pick a random direction and continue on its way, although routing is limited to one-turn paths, to avoid deadlock.

The primary effect of migration is to reduce the number of memory accesses which need to retrieve data from off-chip. However, first

9[5]`barnes`, `radix`, `cholesky`, `water-`n^2, `water-spatial`, `fft`, and `fmm`

9[6]`specjbb`, `tpch`, and `tpcw`

9[7]Conversely, streaming applications which place little demand on caches could turn off L2 caches while using all cores.

we take a higher-level view by observing this effect on the overall average memory access latency, and then we investigate the root cause before discussing a few other factors as well as a small design exploration for our network migration.

Average read latency. Because an on-chip access will almost certainly take less time than an off-chip access, we expect that enabling more data to remain on-chip will have a positive impact on the average read latency. Here we show that this is indeed the case; moreover, this impact increases for larger CMPs. The average read latencies for the 16-thread benchmarks in 16- and 64-node networks are shown in Figures 5(a) and 5(c), respectively. While not shown (so that the performance differences between the configurations can be more easily visualized), the average read times for L2S are over 360% and 332% higher than for L2P in 16- and 64-node networks, a significant difference which has been shown previously [20, 21]. The primary reason for this difference is the increased traffic latency for read requests. Once a cache line is evicted from a node's local L1 cache, then even if the data still remains at the L2 of that line's home node, there is a full round-trip latency between the reader node and the home node. Even victim replication suffers 79% and 32% penalties compared to L2P here; this is markedly higher than the average difference reported in the work by Zhang and Asanovic, where the overall performance of VR is "usually within 5% of L2P [20]." The reason for this is the L2 cache sizes assumed. In their work, the architecture is an 8-node CMP with 1MB L2 cache per node, but here we assume 16- or 64-node CMPs with just 128KB L2 cache per node. We believe that as CMPs are composed of more and more nodes, the per-node L2 cache size will out of necessity decline. Since VR effectively halves the total L2 cache size, smaller caches suffer increased local L2 misses under VR as compared to L2P, and thus the behavior is closer to L2S than with larger caches. We see this effect later in Section 4.3 when we explore the performance of NM under different cache sizes. Finally, since NM is derived from L2P, we expect the performance to match or exceed that of L2P. Indeed, NM provides 1.7% and 6.0% average read latency reduction for 16- and 64-node CMPs respectively. For the 64-node case, this reduction is up to 17.4% (for `specjbb`). `Specjbb` owes its excellent performance to the fact that it has a read-to-write (r/w) access ratio of 26.7; since writes invalidate on-chip copies of data, they can effectively negate the benefit of migrated cachelines, thus keeping the performance of NM close to that of L2P. `Tpch` and `tpcw` have r/w ratios of 13.4 and 12.8, respectively, and they perform well; the maximum r/w ratio of all other benchmarks is 4.28. These results highlight not just the performance advantage our scheme has over the next best design, but they also highlight its scalability in the face of near-future CMPs with tens of nodes.

Also of note is that the performance of the baseline network migration configuration approaches that of the pseudo-optimal migration configuration. For the 4x4 and 8x8 networks, Opt provides average memory access times which are 1.4% and 8.4% better than L2P. Finally, our network migration configuration performs better than a random migration, as described above. In 4x4 and 8x8 CMPs, the average read latencies for Rnd are 0.5% and 3.8% lower than L2P across all benchmarks. So particularly in larger CMPs where the effects are more pronounced, our proposed scheme performs better than a random implementation and nearly as well as a pseudo-optimal approach.

Average write latency. While keeping data on-chip reduces a future read's latency, it may actually increase the latency of a future write to the same address, since that copy must be invalidated before the write can proceed. The average write times for 4x4 and 8x8 networks are shown in Figures 5(b) and 5(d). We do not observe a measurable difference between NM and L2P, despite our concerns above. However, we do see lower write latencies for L2S (again not shown) and VR. While L2S exhibits the highest average read times for most of the benchmarks, it exhibits the lowest average write request times: 20.3% and 16.8% lower than L2P in 4x4 and 8x8 CMPs respectively. But both trends can be explained for the same reason. For memory requests whose addresses all map to the same set, the contention will be greatest at the directory cache of the home node for L2P, assuming that the requests are fairly evenly distributed amongst the nodes. But for L2S, while the point of contention will still be at the home node, it can be either at the directory cache *or* the L2 cache, depending on which has a lower associativity. In this case, the directory caches are 16-way set associative, whereas the L2 caches are only 4-way, so the contention is greatest at the L2. Hence, there are more L2 evictions for L2S. Since there are more evictions, it is less likely that when a write request misses in the local node, it will reach the directory node and find any valid data on-chip which has to be invalidated, and thus it will be able to return immediately to the requesting node, without paying any additional delay which could be tens of cycles. So it makes sense that the benchmarks which experience the greatest increase in L2 invalidations, and therefore off-chip memory accesses, such as `cho` and `fft`, experience the greatest increase in average read request time, but also the greatest decrease in average write request time. We see the same trend in the comparison between L2P and VR with comparably smaller magnitude: the average write request latencies under VR for 4x4 and 8x8 CMPs are 6.7% and 1.6% lower than those for L2P. Finally, although not shown, the trend of the overall average access (read and write) latencies closely tracks that of the reads: 1.4% and 3.9% average reduction in 4x4 and 8x8 CMPs. While again this may seem unexpected (since, as read latencies decrease, write latencies increase, as seen above), many more cycles are spent completing read requests than write requests. On average, approximately 80% of the cycles spent filling memory requests (from main memory accesses to local L1 hits) are for reads.

Memory accesses. As explained previously, migration's primary benefit is that it can reduce the number of off-chip memory accesses by retaining evicted L2 cachelines on-chip. As a result, we expect that the number of off-chip accesses should be significantly reduced under NM. Here we show that this is indeed the case. Figure 6 shows the total number of off-chip memory accesses for each of the benchmarks under the baseline configuration as specified in Table 2 and in 8x8 CMPs. For each benchmark, the number of accesses for each design is normalized to that of L2P. From the figure, we can see that the average increase in the number of off-chip memory accesses for VR compared to L2P is about 42%, compared with just a 32% increase in the average read latency. Similarly, NM experiences a 19% *reduction* in the number of off-chip accesses for the baseline configuration, which results in a 6.0% reduction, as seen above, in the average read latency for the 8x8 CMP. So, as expected, we see that the trend of the number of read requests resulting in off-chip memory accesses tracks that of the average read latency, but by a significantly larger magnitude, as the savings are amortized over all read accesses.

4.2 Effect of off-chip memory access time

To asses the potential impact of an architecture which has a higher main memory access latency, we repeated the above experiments with an access time of 400 cycles, twice that of the baseline assumption, and we observed the impact on the average read and write latencies. Since we have seen that the primary benefit of network migration is a result of fewer off-chip memory accesses, we expect the performance relative to L2P to increase. In the interests of space, we have done these experiments only on 8x8 CMPs and we omit figures.

The average read latency reduction (compared to L2P) across all benchmarks for NM is now 10.9% compared to a 6.0% reduction as seen above. Write latencies remain relatively unchanged as expected: 1.9% increase in the average write latency for 400 cycle main memory accesses, as compared to a 1.8% increase for 200

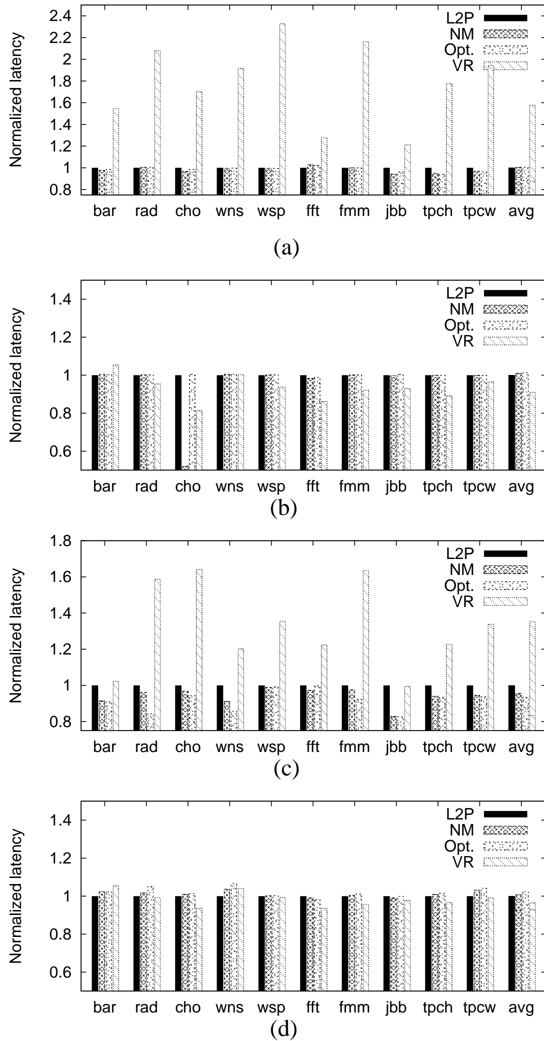

Figure 5: Normalized (to performance under L2P scheme) average latencies for (a) reads in 16-node CMP, (b) writes in 16-node CMP, (c) reads in 64-node CMP, (d) writes in 64-node CMP.

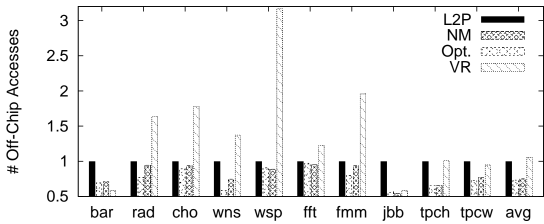

Figure 6: Normalized (to L2P) total off-chip memory accesses.

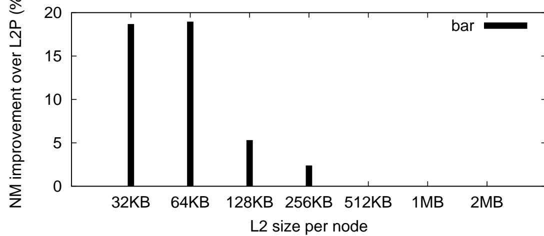

Figure 7: Benefit of migration as a function of L2 size. 128KB is our default L2 size.

cycle main memory accesses. Since we have seen that VR results in more off-chip memory accesses in our experiments, we would expect that it would diverge from L2P in the opposite direction. Indeed, assuming a 400 cycle off-chip access latency, the average read latency under VR is 34% higher than L2P, a modest increase from 32% for 200 cycle main memory access. So if the memory wall [19] continues to increase, network migration will provide an even greater benefit.

4.3 Effect of L2 Cache Size

Varying the cache sizes of each node has a direct effect on the improvement that network migration has over the baseline private cache configuration. Since the benefit of NM comes from the ability to retain cache lines on-chip when they would otherwise be evicted, it follows that NM should perform best when resources are constrained. We demonstrate this behavior by varying the size of the L2 caches for the `barnes` benchmark in an 8x8 CMP.

Figure 7 shows that indeed the performance of NM relative to L2P decreases as the size of the L2 increases. This is because as the L2 size increases, there are fewer and fewer evictions, which in turn means that the *opportunity* for improvement shrinks dramatically. However, as mentioned previously, we believe that as the number of cores per CMP increases, it will be difficult to feasibly implement L2 caches on the order of 1MB per node.

As we reduce the L2 cache size below 128KB, marked performance improvements of close to 20% against L2P is observed for 64 and 32KB. However, if we were to extend Figure 7 to the left (*i.e.* to even smaller L2 caches), we would observe the relative benefit of NM decreasing again. This is because the L2 caches become so small that there are an unmanageable number of evictions. Successful migrations are evicted before they can ever be used. However, this region is not very interesting because it is not likely that general-purpose CMPs will be designed with L2 caches so small since the absolute performance would be unacceptably low.

4.4 Network Migration Design Exploration

In order to find the optimal configuration for our network migration scheme, we performed a detailed design space exploration of the parameters described in Section 3.

First, we determined the effect of varying the frequency with which the link scores are updated. This has a direct effect on the network traffic overhead of network migration. [8] The update frequency should also have an indirect effect on the performance of NM since routing decisions will be made based on stale data if the updates are infrequent. However, it turns out that this is not the case. We varied the update interval from 10 cycles to 100K cycles for `barnes`, `wns`, and `wsp`. There is no clear trend and the variation is negligible: the range of read latency variation is only 0.01% of its average. The reason for this is that the average age of all valid cachelines in the network is significantly higher than the maximum update interval tested: approximately 3M cycles (after 35M cycles of simulation). In other words, the turnover in the L2 caches is very low, so even updating the link scores every 100K cycles results in few actual altered link scores. We therefore chose 100K cycles as the upper bound because this interval, in combination with the baseline parameters, results in an insignificant 1% traffic overhead (Section 3.5). Reducing this overhead further gains little benefit, at the cost of possible performance degradation for applications which may have a higher cache turnover.

It is worth noting here that given such a low cache turnover, a more demand-driven update scheme may reduce traffic and energy

9 [8]Though recall that since the link scores never travel more than one hop, they do not traverse the crossbars and can greedily traverse the links during idle cycles, and so will not increase congestion. Thus, the overhead is just in link energy/power consumption, not router energy-delay or network throughput.

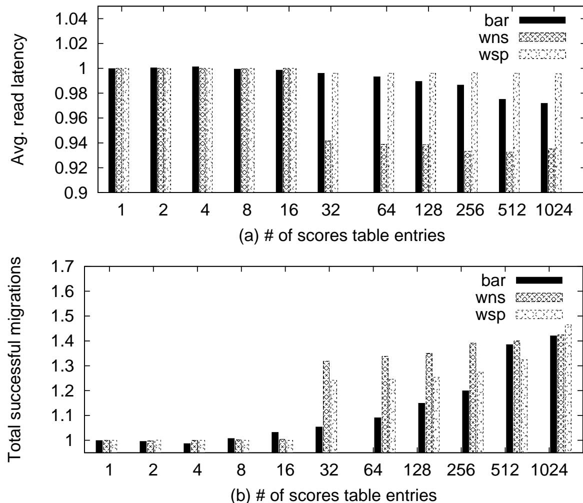

Figure 8: (a) Average memory latency and (b) number of successful migrations when the number of entries in the scores table is varied. Benchmark is `barnes` **in an 8x8 CMP.**

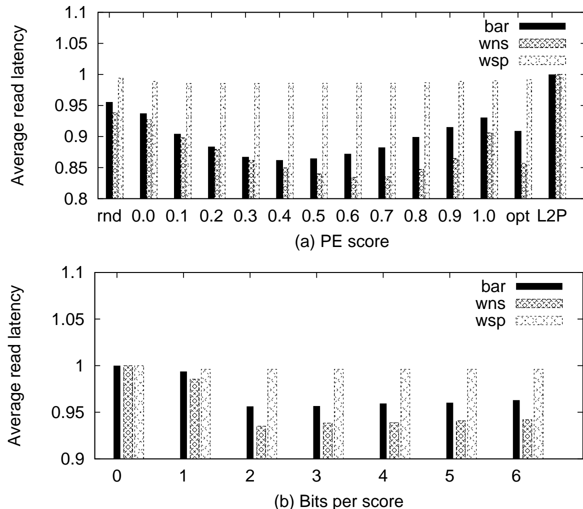

Figure 9: (a) Average read latency when the PE score is varied and (b) Average read latency when the number of bits per score is varied for `barnes` **in an 8x8 CMP.**

overhead even further. Although we do not implement it here, such a scheme would, instead of updating *all* link scores every 100K cycles, only update those scores *which have been modified* since the last update. We leave this analysis to future work.

Next, we varied the number of entries in the scores tables, from 1 to 1024 (the number of L2 cache sets in our baseline configuration) in powers of 2. The results are presented in Figure 8(a), which shows that the behavior is dependent on the application, but in general the average read latency increases as expected as the number of entries is reduced. The exception for this set is `wsp`, but this makes sense since we saw from Figure 5(c) that `wsp` did not experience measurable improvement over L2P. Furthermore, `wns` exhibits a bimodal performance owing to its particular memory access patterns. Also, when we plot the number of successful migrations against the size of the scores tables (Figure 8(b)), we see the similar trends. For `barnes`, the number of successful migrations drops off steadily as the number of entries is reduced from 1024 to 16 and then levels off, as in Figure 8(a). But in this case both `wns` and `wsp` exhibit bimodal behavior. This highlights the fact that migrations may be successful, but may not be subsequently accessed, but in general we see the direct corellation between the number of entries and the success of the migrations, as well as how this translates into reduced memory latency. Since the magnitude of the variation is not dramatic in this case, we chose the number of entries with overhead as the first-order constraint. We chose tables with 64 entries as they are above the bimodal discontinuities of `wns` and `wsp`, yet small enough such that they do not take many cycles to transfer from one node to its neighbor. It is clear that this choice is somewhat arbitrary, yet choosing smaller values for this and other parameters demonstrates that good performance improvements can be obtained even under tight hardware constraints; in general the parameters should be as large as hardware constraints will allow.

To determine the optimal PE score, we varied this parameter while keeping the others constant as shown in Table 2. The results are shown in Figure 9(a). From this figure, we can see that the best performance comes when the PE score threshold is around 0.4 (meaning the local L2 cache is 40% full for the relevant region) for `barnes` and 0.7 for `wns`. Again `wsp` doesn't exhibit a performance difference as the PE score is varied. For the first two, the trend is exactly as we would expect, with the performance decreasing both as the PE threshold is increased, and as it is decreased. If the threshold is 1.0, then any migration message will try to migrate to its current node, and success is not predictable (*i.e.* it is random); conversely, if the threshold is 0.0, then the migrant is not likely to run into any nodes which have completely empty cache sets (and the larger the region, the less likely it will be entirely empty), resulting in migrants routing throughout the network until routing limitations dictate that they must give up seeking and attempt to migrate to their current nodes. This also approximates random migration. Note also that a number of threshold values outperform Opt, demonstrating that it really is *pseudo*-optimal, as discussed previously.

Finally, we varied the number of bits per score. As with the number of entries per scores table, this parameter varies the precision of the technique, so we would expect that as the scores become less precise (fewer bits per score), the latency should increase. Figure 9(b) shows the results when the numbers of bits is varied from 0 (ignoring the scores) to 7 (the maximum given 64 scores table entries, each covering 128 lines of the L2 cache). What we see again is a bimodal behavior for `barnes` and `wns`, and minimal variation for `wsp`. For the first two, using 0 or 1 bits results in significantly worse performance than 2 to 7 bits per score, but within each group there is very little variation. As a result, we choose to use just 2 bits per score, the lowest number of bits which will provide equivalent performance to the full-precision configuration.

5. CONCLUSIONS

In this paper, we proposed embedding functionality within the network fabric in order to efficiently perform cache migration. Our technique is a fully distributed approach, which affords it scalability. There have been a number of previous works on replication and migration in the context of CMPs, but all face challenges when scaling to CMPs with tens of nodes or more. We have shown how our migration technique retains more data on-chip and therefore results in fewer off-chip memory accesses; this directly translates to lower average memory access latency. We have seen that in a 64-node CMP, an average of 19% and up to 44.5% off-chip memory access reduction leads directly to a 6.0% and up to 15.3% overall memory latency reduction, compared to the next-best option, while incurring negligible area overhead and a manageable 13.4% traffic overhead. In a broader sense, we see this technique as just one piece of the puzzle in taking advantage of on-chip networks to provide dynamic in-transit functionality and to provide new opportunities for optimization.

Acknowledgments

9We would like to thank Natalie Enright-Jerger and the rest of the PharmSim group at the university of Wisconsin for providing us with trace files of `specjbb, tpch,` and `tpcw`. We would also like to thank Shougata Ghosh for providing us with trace files from Simics for the SPLASH-2 benchmarks. This work was supported in part by MARCO Gigascale Systems Research Center and NSF CNS-0509402, as well as NSERC Discovery Grant No. 388694-01.

6. REFERENCES

[1] B. M. Beckmann *et al. ASR: Adaptive Selective Replication for CMP Caches*. In Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 443–454, December, 2006.

[2] D. Burger *et al. Memory Bandwidth Limitations of Future Microprocessors*. In Proc. of the 23rd Annual International Symposium on Computer Architecture, pp. 78–89, May, 1996.

[3] H. Cain *et al. Precise and Accurate Processor Simulation*. In Proc. of the 5th Workshop on Computer Architecture Evaluation Using Commercial Workloads, pp. 13–22, February, 2006.

[4] J. Chang *et al. Cooperative Caching for Chip Multiprocessors*. In Proc. of the 33rd Annual International Symposium on Computer Architecture, pp. 264-276, May, 2006.

[5] J. Chen *et al. Hardware-Modulated Parallelism in Chip Multiprocessors*. In DASCMP, November, 2005.

[6] Z. Chishti *et al. Optimizing Replication, Communication, and Capacity Allocation in CMPs*. In Proc. of the 32nd Annual International Symposium on Computer Architecture, pp. 357–368, May, 2005.

[7] N. Eisley *et al. In-Network Cache Coherence*. In Proc. of the 39th Annual International Symposium on Microarchitecture, pp. 321-332, December, 2006.

[8] J. R. Goodman and P. J. Woest. *The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor*. in Proc. of the 15th International Symposium on High Performance Computer Architecture, pp. 422–431, June, 1988.

[9] L. Hammond *et al. The Stanford Hydra CMP*. In IEEE Micro, Vol. 20, No. 2, pp. 71–84, 2000.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers, Inc., 2003.

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

[12] R. Iyer *et al. Using Switch Directories to Speed up Cache-to-Cache Transfers in CC-NUMA Multiprocessors*. In Proc. of the 14th International Parallel and Distributed Processing Symposium, pp. 721-728, May, 2000.

[13] S. Kaxiras and J. R. Goodman. *The GLOW Cache Coherence Protocol Extensions for Widely Shared Data*. In Proc. of the 10th International Conference on Supercomputing, pp. 35–43, May, 1996.

[14] L. Lamport. *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs*. In IEEE Transactions on Computing, Vol. c-28, No. 9, pp. 690–691, September, 1979.

[15] A. Mendelson *et al. CMP Implementation in Systems Based on the Intel Core Duo Processor*. In Intel Technology Journal, Vol. 10, No. 2, May, 2006.

[16] H. E. Mizrahi *et al. Introducing Memory into the Switch Elements of Multiprocessor Interconnection Networks*. In Proc. of the 16th International Symposium on Computer Architecture, pp. 158–166, June, 1989.

[17] K. Olukotun *et al. The Case for a Single-Chip Multiprocessor*. In IEEE SIGPLAN Notices, Vol. 31, No. 9, pp. 2–11, 1996.

[18] S. J. E. Wilton and N. P. Jouppi. *An Enhanced Access and Cycle Time Model for on-Chip Caches*. DEC Western Research Laboratory, No. 93/5, 1994.

[19] W. A. Wulf and S. A. McKee. *Hitting the Memory Wall: Implications of the Obvious*. In SIGARCH Computer Architecture News, Vol. 23, No. 1, pp. 20–24, 1995.

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

[21] M. Zhang and K. Asanovic. *Victim Migration: Dynamically Adapting between Private and Shared CMP Caches*. MIT Technical Report MIT-CSAIL-TR-2005-064, MIT-LCS-TR-1006, October, 2005.

[22] http://www-128.ibm.com/developerworks/power/library/pa-expert1.html

[23] http://www-flash.stanford.edu/apps/SPLASH/

[24] http://www.intel.com/multi-core/

[25] http://www.sun.com/processors/throughput/

[26] http://www.virtutech.com/