Judicious Thread Migration When Accessing Distributed Shared Caches

Keun Sup Shim, Mieszko Lis, Omer Khan and Srinivas Devadas
Massachusetts Institute of Technology

***Abstract*—Chip-multiprocessors (CMPs) have become the mainstream chip design in recent years; for scalability reasons, designs with high core counts tend towards tiled CMPs with physically distributed shared caches. This naturally leads to a Non-Uniform Cache Architecture (NUCA) design, where on-chip access latencies depend on the physical distances between requesting cores and home cores where the data is cached. Improving data locality is thus key to performance, and several studies have addressed this problem using data replication and data migration.**

In this paper, we consider another mechanism, hardware-level thread migration. This approach, we argue, can better exploit shared data locality for NUCA designs by effectively replacing multiple round-trip remote cache accesses with a smaller number of migrations. High migration costs, however, make it crucial to use thread migrations judiciously; we therefore propose a novel, on-line prediction scheme which decides whether to perform a remote access (as in traditional NUCA designs) or to perform a thread migration at the instruction level. For a set of parallel benchmarks, our thread migration predictor improves the performance by 18% on average and at best by 2.3X over the standard NUCA design that only uses remote accesses.

I. Background

In the recent years, transistor density has continued to grow [13] and Chip Multiprocessors (CMPs) with four or more cores on a single chip have become common in the commodity and server-class general-purpose processor markets [25]. To further improve performance and use the available transistors more efficiently, architects are resorting to medium and large-scale multicores both in academia (e.g., Raw [31], TRIPS [26]) and industry (e.g., Tilera [12], [4], Intel TeraFLOPS [29]), and industry pundits are predicting 1000 or more cores in a few years [5].

With this trend towards massive multicore chips, a tiled architecture where each core has a slice of the last-level on-chip cache has become a popular design, and these physically distributed per-core cache slices are unified into one large, logically shared cache, known as the Non-Uniform Cache Architecture (NUCA) [18]. In the pure form of NUCA, only one copy of a given cache line is kept on chip, maximizing the effective on-chip cache capacity and reducing off-chip access rates. In addition, because only one copy is ever present on-chip, no two caches can disagree about the value at a given address and cache coherence is trivially ensured. A private per-core cache organization, in comparison, would need to rely on a complex coherence mechanism (e.g., a directory-based coherence protocol); these mechanisms not only pay large area costs but also also incur performance costs because repeated cache invalidations are required for shared data with frequent writes. NUCA obviates the need for such coherence overhead.

The downside of NUCA designs, however, is high on-chip access latency, since every access to an address cached remotely must cross the physical distances between the requesting core and the *home* core where the data can be cached. Therefore, various NUCA and hybrid designs have been proposed to improve data locality, leveraging data migration and replication techniques previously explored in the NUMA context (e.g., [30]). These techniques move private data to its owner core and replicate read-only shared data among the sharers at OS level [11], [15], [1] or aided by hardware [33], [8], [28]. While these schemes improve performance on some kinds of data, they still do not take full advantage of spatio-temporal locality and rely on remote cache accesses with two-message round trips to access read/write shared data cached on a remote core.

To address this limitation and take advantage of available data locality in a memory organization where there is only one copy of data, we consider another mechanism, *fine-grained hardware-level thread migration* [19], [20]: when an access is made to data cached at a remote core, the executing thread is simply migrated to that core, and execution continues there. When several consecutive accesses are made to data assigned to a given core, migrating the thread context allows the thread to make a sequence of local accesses on the destination core rather than pay the performance penalty of the corresponding remote accesses, potentially better exploiting data locality. Due to the high cost of thread migration, however, it is crucial to judiciously decide whether to perform remote accesses (as in traditional NUCA designs) or thread migrations, a question which has not been thoroughly explored.

In this paper, we explore the tradeoff between the two different memory access mechanisms and answer the question of when to migrate threads instead of performing NUCA-style remote accesses. We propose a novel, on-line prediction scheme which detects the first instruction of each memory instruction sequence in which every instruction accesses the same home core and decides to migrate depending on the length of this sequence. This decision is done at instruction granularity. With a good migration predictor, thread migration can be considered as a new means for memory access in NUCA designs, that is complementary to remote access.

In the remainder of this paper,

- we first describe two memory access mechanisms – *remote cache access* and *thread migration* – and explain

the tradeoffs between the two;
- we present a novel, PC-based migration prediction scheme which decides at instruction granularity whether to perform a remote access or a thread migration;
- through simulations of a set of parallel benchmarks, we show that thread migrations with our migration predictor result in a performance improvement of 18% on average and at best by 2.3X compared to the baseline NUCA design which only uses remote accesses.

II. Memory Access Framework

NUCA architectures eschew capacity-eroding replication and obviate the need for a coherence mechanism entirely by combining the per-core caches into one large logically shared cache [18]. The address space is divided among the cores in such a way that each address is assigned to a unique *home core* where the data corresponding to the address can be cached; this necessitates a memory access mechanism when a thread wishes to access an address not assigned to the core it is running on. The NUCA architectures proposed so far use a *remote access* mechanism, where a request is sent to the home core and the data (for loads) or acknowledgement (for writes) is sent back to the requesting core.

In what follows, we first describe the remote access mechanism used by traditional NUCA designs. We also describe another mechanism, *hardware-level thread migration*, which has the potential to better exploit data locality by moving the thread context to the home core. Then, we explore the tradeoff between the two and present a memory access framework for NUCA architectures which combines the two mechanisms.

A. Remote Cache Access

Since on-chip access latencies are highly sensitive to the physical distances between requesting cores and home cores, effective *data placement* is critical for NUCA to deliver high performance. In standard NUCA architectures, the operating system controls memory-to-core mapping via the existing virtual memory mechanism: when a virtual address is first mapped to a physical page, the OS chooses where the relevant page should be cached by mapping the virtual page to a physical address range assigned to a specific core. Since the OS knows which thread causes a page fault, more sophisticated heuristics can be used: for example, in a first-touch-style scheme, the OS can map the page to the core where the thread is running, taking advantage of data access locality. For maximum data placement flexibility, each core might include a Core Assignment Table (CAT), which stores the home core for each page in the memory space. Akin to a TLB, the per-core CAT serves as a cache for a larger structure stored in main memory. In such a system, the page-to-core assignment might be made when the OS is handling the page fault caused by the first access to the page; the CAT cache at each core is then filled as needed[1].

[1]Core Assignment Table (CAT) is not an additional requirement for our framework. Our memory access framework can be integrated with any data placement scheme.

Under the remote-access framework, all non-local memory accesses cause a request to be transmitted over the interconnect network, the access to be performed in the remote core, and the data (for loads) or acknowledgement (for writes) to be sent back to the requesting core: when a core C executes a memory access for address A, it must

1) compute the *home* core H for A (e.g., by consulting the CAT or masking the appropriate bits);
2) if $H = C$ (a *core hit*),
   a) forward the request for A to the cache hierarchy (possibly resulting in a DRAM or next-level cache access);
3) if $H \neq C$ (a *core miss*),
   a) send a remote access request for address A to core H;
   b) when the request arrives at H, forward it to H's cache hierarchy (possibly resulting in a DRAM access);
   c) when the cache access completes, send a response back to C;
   d) once the response arrives at C, continue execution.

Accessing data cached on a remote core requires a potentially expensive two-message round-trip: unlike a private cache organization where a coherence protocol (e.g., directory-based protocol) would take advantage of spatial and temporal locality by making a copy of the block containing the data in the local cache, a traditional NUCA design must repeat the round-trip *for every remote access*. Optimally, to reduce remote cache access costs, data private to a thread should be assigned to the core the thread is executing on or to a nearby core; threads that share data should be allocated to nearby cores and the shared data assigned to geographically central cores that minimize the average remote access delays. In some cases, efficiency considerations might dictate that critical portions of shared read-only data be replicated in several per-core caches to reduce overall access costs. For shared read/write data cached on a remote core (which are not, in general, candidates for replication), a thread still needs to perform remote accesses.

B. Thread Migration

In addition to the remote access mechanism, fine-grained, hardware-level thread migration has been proposed to exploit data locality for NUCA architectures [19], [20]. A thread migration mechanism brings the *thread* to the locus of the data instead of the other way around: when a thread needs access to an address cached on another core, the hardware efficiently migrates the thread's execution context to the core where the memory is (or is allowed to be) cached and continues execution there.

If a thread is already executing at the destination core, it must be evicted and migrated to a core where it can continue running. To reduce the necessity for evictions and amortize the latency of migrations, cores duplicate the architectural context (register file, etc.) and allow a core to multiplex execution among two (or more) concurrent threads. To prevent deadlock,

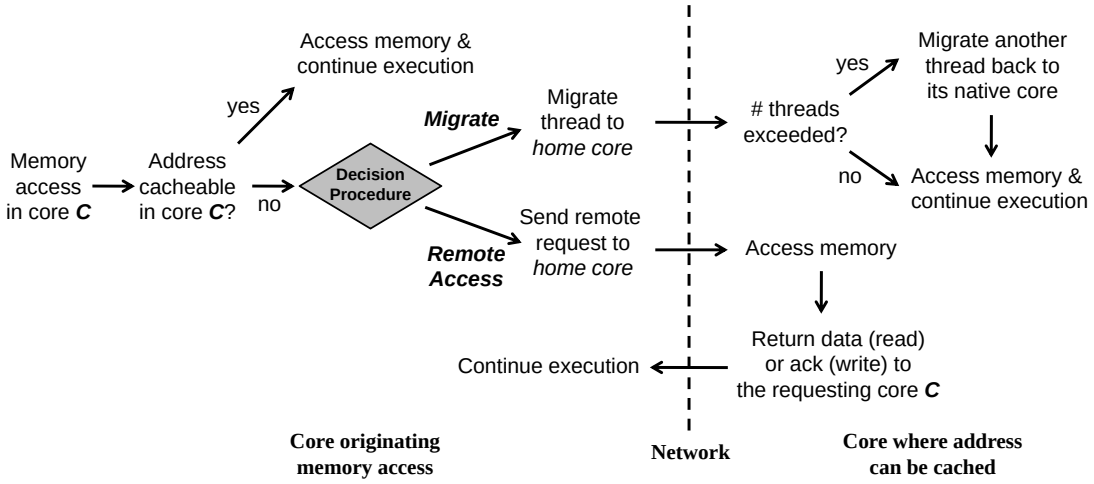

Fig. 1. In NUCA architectures, memory accesses to addresses not assigned to the local core result in remote data accesses. Here, we add another mechanism for accessing remote memory – migrating a thread to the home core where the data is cached. In this paper, we propose an on-line decision scheme which efficiently decides between a remote access and a thread migration for every memory access.

one context is marked as the *native context* and the other is the *guest context*: a core's native context may only hold the thread that started execution on that core (called the thread's *native core*), and evicted threads must migrate to their native cores to guarantee deadlock freedom [10].

Briefly, when a core C running thread T executes a memory access for address A, it must

1) compute the *home* core H for A (e.g., by consulting the CAT or masking the appropriate bits);
2) if $H = C$ (a *core hit*),
   a) forward the request for A to the cache hierarchy (possibly resulting in a DRAM access);
3) if $H \neq C$ (a *core miss*),
   a) interrupt the execution of the thread on C (as for a precise exception),
   b) migrate the microarchitectural state to H via the on-chip interconnect:
      i) if H is the native core for T, place it in the native context slot;
      ii) otherwise:
         A) if the guest slot on H contains another thread T', evict T' and migrate it to its native core N'
         B) move T into the guest slot for H;
   c) resume execution of T on H, requesting A from its cache hierarchy (and potentially accessing backing DRAM or the next-level cache).

Although the migration framework requires hardware changes to the baseline NUCA system (since the core itself must be designed to support efficient migration), it migrates threads directly over the interconnect network to achieve the shortest possible migration latencies, which is faster than other thread migration approaches (such as OS-level migration or Thread Motion [24], which uses special cache entries to store thread contexts and leverages the existing cache coherence protocol to migrate threads). In terms of a thread context size that needs to be migrated, the relevant architectural state in a 64-bit x86 processor amounts to about 3.1Kbits (16 64-bit general-purpose registers, 16 128-bit floating-point registers and special purpose registers, e.g., *rflags*, *rip* and *mxcsr*), which is the context size we are assuming in this paper. The thread context size may vary depending on the architecture; in the Tilera TILEPro64 [4], for example, it amounts to about 2.2Kbits (64 32-bit registers and a few special registers).

C. Hybrid Framework

We propose a hybrid memory access framework for NUCA architectures by combining the two mechanisms described: each core-miss memory access may either perform the access via a remote access as in Section II-A or migrate the current execution thread as in Section II-B. The hybrid architecture is illustrated in Figure 1. For each access to memory cached on a remote core, a decision algorithm determines whether the access should migrate to the target core or execute a remote access.

As discussed earlier, the approach of migrating the thread context can potentially better take advantage of spatiotemporal locality: where a remote access mechanism would have to make repeated round-trips to the same remote core to access its memory, thread migration makes a one-way trip to the core where the memory can be cached and continues execution there; unless every other word accessed resides at a different core, it will make far fewer network trips.

At the same time, we need to consider the cost of thread migration: given a large thread context size, the thread migration cost is much larger than the cost required by remote-access-only NUCA designs. Therefore, when a thread is migrated to another core, it needs to make several *local* memory accesses to make the migration "worth it." While some of this can be addressed via intelligent data layout [27] and memory access reordering at the compiler level, occasional "one-off" accesses

seem inevitable and migrating threads for these accesses will result in expensive back-and-forth context transfers. If such an access can be predicted, however, we can adopt a hybrid approach where "one-off" accesses are executed under the remote access protocol, and migrations handle sequences of accesses to the same core. The next section discusses how we address this decision problem.

III. Thread Migration Prediction

As described in Section II, it is crucial for the hybrid memory access architecture (remote access + thread migration) to make a careful decision whether to follow the remote access protocol or the thread migration protocol. Furthermore, because this decision must be taken on every access, it must be implementable as efficient hardware. Since thread migration has an advantage over the remote access protocol for multiple contiguous memory accesses to the same location but not for "one-off" accesses, our migration predictor focuses on detecting such memory sequences that are worth migrating.

A. Detection of Migratory Instructions

Our migration predictor is based on the observation that sequences of consecutive memory accesses to the same home core are highly correlated with the program (instruction) flow, and moreover, these patterns are fairly consistent and repetitive across the entire program execution. At a high level, the predictor operates as follows:

1) when a program first starts execution, it basically runs as on a standard NUCA organization which only uses remote accesses;
2) as it continues execution, it keeps monitoring the home core information for each memory access, and
3) remembers each first instruction of every sequence of multiple successive accesses to the same home core;
4) depending on the length of the sequence, marks the instruction either as a *migratory instruction* or a *remote-access instruction*;
5) the next time a thread executes the instruction, it migrates to the home core if it is a migratory instruction, and performs a remote access if it is a remote-access instruction.

The detection of *migratory instructions* which trigger thread migrations can be easily done by tracking how many consecutive accesses to the same remote core have been made, and if this count exceeds a threshold, marking the instruction to trigger migration. If it does not exceed the threshold, the instruction is marked as a remote-access instruction, which is the default state. This requires very little hardware resources: each thread tracks (1) *Home*, which maintains the home location (core ID) for the current requested memory address, (2) *Depth*, which indicates how many times so far a thread has contiguously accessed the current home location (i.e., the *Home* field), and (3) *Start PC*, which keeps record of the PC of the very first instruction among memory sequences that accessed the home location that is stored in the *Home* field. We separately define the depth threshold θ, which indicates the depth at which we determine the instruction as migratory. With a 64-bit PC, 64 cores (i.e., 6 bits to store the home core ID) and a depth threshold of 8 (3 bits for the depth field), it requires a total of 73 bits; even with a larger core count and a larger threshold, fewer than 100 bits are sufficient to maintain this data structure. When a thread migrates, this data structure needs to be transferred together with its 3.1Kbit context (cf. II-B), resulting in 3.2Kbits in total. In addition, we add one bit to each instruction in the instruction cache (see details in Section III-B) indicating whether the instruction has been marked as a migratory instruction or not, a negligible overhead.

The detection mechanism is as follows: when a thread T executes a memory instruction for address A whose PC = P, it must

1) compute the *home* core H for A (e.g., by consulting the CAT or masking the appropriate bits);
2) if $Home = H$ (i.e., memory access to the same home core as that of the previous memory access),
   a) if $Depth < \theta$,
      i) increment $Depth$ by one, then if $Depth = \theta$, $StartPC$ is marked as a migratory instruction.
3) if $Home \neq H$ (i.e., a new sequence starts with a new home core),
   a) if $Depth < \theta$,
      i) $StartPC$ is marked as a remote-access instruction[2];
   b) reset the entry (i.e., $Home = H$, $PC = P$, $Depth = 1$).

Figure 2 shows an example of the detection mechanism when $\theta = 2$. Suppose a thread executes a sequence of memory instructions, $I_1 \sim I_8$. Non-memory instuctions are ignored because they do not change the entry content nor affect the mechanism. The PC of each instruction from I_1 to I_8 is PC_1, PC_2, ... PC_8, respectively, and the home core for the memory address that each instruction accesses is specified next to each PC. When I_1 is first executed, the entry {Home, Depth, Start PC} will hold the value of {A, 1, PC_1}. Then, when I_2 is executed, since the home core of I_2 (B) is different from *Home* which maintains the home core of the previous instruction I_1 (A), the entry is reset with the information of I_2. Since the *Depth* to core A has not reached the depth threshold, PC_1 is marked as a remote-access instruction (default). The same thing happens for I_3, setting PC_2 as a remote-access instruction. Now when I_4 is executed, it accesses the same home core C and thus only the *Depth* field needs to be updated (incremented by one). After the Depth field is updated, it needs to be checked to see if it has reached the threshold θ. Since we assumed $\theta = 2$, the depth to the home core C now has reached the threshold and therefore, PC_3 in the *Start PC* field, which represents the first instruction (I_3) that

[2]Since all instructions are initially considered as remote-accesses, marking the instruction as a remote-access instruction will have no effect if it has not been classified as a migratory instruction. If the instruction was migratory, however, it reverts back to the remote-access mode.

accessed this home core C, is now classified as a migratory instruction. For I_5 and I_6 which keep accessing the same home core C, we need not update the entry because the first, migration-triggering instruction has already been detected for this sequence. Executing I_7 resets the entry and starts a new memory sequence for the home core A, and similarly, I_7 is detected as a migratory instruction when I_8 is executed. Once a specific instruction (or PC) is classified as a migratory instruction and is again encountered, a thread will directly migrate instead of sending a remote request and waiting for a reply.

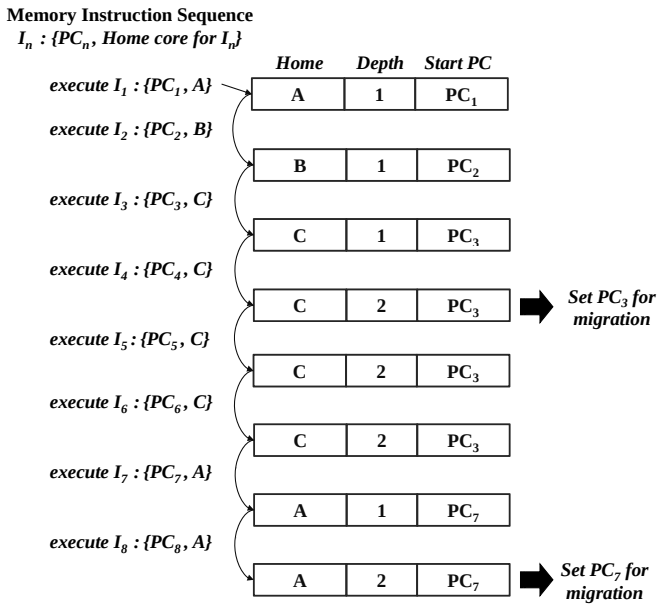

Fig. 2. An example how instructions (or PC's) which are followed by consecutive accesses to the same home location, i.e., *migratory instructions* are detected in the case of the depth threshold $\theta = 2$. Setting $\theta = 2$ means that a thread will perform remote accesses for "one-off" accesses and will migrate for multiple accesses (≥ 2) to the same home core.

Figure 3 shows how this migration predictor actually improves data locality for the example sequence we used in Figure 2. Suppose a thread originated at core A, and thus, it runs on core A. Under a standard, remote-access-only NUCA where the thread will never leave its native core A, the memory sequence will incur five round-trip remote accesses; among eight instructions from I_1 to I_8, only three of them (I_1, I_7 and I_8) are accessing core A which result in *core hits*. With our migration predictor, the first execution of the sequence will be the same as the baseline NUCA, but from the second execution, the thread will now migrate at I_3 and I_7. This generates two migrations, but since I_4, I_5 and I_6 now turn into *core hits* (i.e., local accesses) at core C, it only performs one remote access for I_2. Overall, five out of eight instructions turn into local accesses with effective thread migration.

B. Storing and Lookup of Migratory Instructions

Once a migratory instruction is detected, a mechanism to store the detection is necessary because a thread needs to

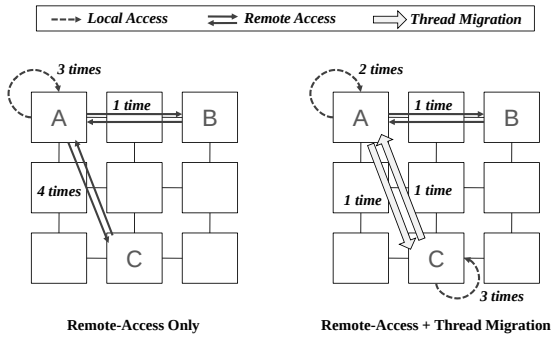

Fig. 3. The number of remote accesses and migrations in the baseline NUCA with and without thread migration.

migrate when it executes this instruction again during the program. We add one-bit called the "migratory bit" for each instruction in *the instruction cache* to store this information. Initially, these bits are all zeros; all memory instructions are handled by remote-accesses when the program first starts execution. When a particular instruction is detected as a migratory instruction, this migration bit is set to 1. The bit is set to 0 if the instruction is marked as a remote-access instruction, allowing migratory instructions to revert back to the remote-access mode. In this manner, the lookup of the migratory information for an instruction also becomes trivial because the migratory bit can be read together with the instruction during the instruction fetch phase with almost no overhead.

When the cache block containing a migratory instruction gets evicted from the instruction cache, we can choose either to store the information in memory, or to simply discard it. In the latter case, it is true that we may lose the migratory bit for the instruction and thus, a thread will choose to perform a remote access for the first execution when the instruction is reloaded in the cache from memory. We believe, however, that this effect is negligible because miss rates for instruction caches are extremely low and furthermore, frequently-used instructions are rarely evicted from the on-chip cache. We assume the migratory information is not lost in our experiments.

Another subtlety is that since the thread context transferred during migration does not contain instruction cache entries, the thread can potentially make different decisions depending on which core it is currently running on, i.e., which instruction cache it is accessing. We rarely observed prediction inaccuracies introduced by this, however. For multithreaded benchmarks, all worker threads execute almost identical instructions (although on different data), and when we actually checked the detected migratory instructions for all threads, they were almost identical; this effectively results in the same migration decisions for any instruction cache. Therefore, a thread can perform migration prediction based on the I-$ at the current core it is running on without the overhead of having to send the migratory information with its context. It is important to note that even if a misprediction occurs due to either cache eviction or thread migration (which is very rare), the memory

access will still be carried out correctly (albeit perhaps with suboptimal performance), and the functional correctness of the program is maintained.

IV. Evaluation

A. Simulation Framework

We use Pin [2] and Graphite [22] to model the proposed NUCA architecture that supports both remote-access and thread migration. Pin enables runtime binary instrumentation of parallel programs, including the SPLASH-2 [32] benchmarks we use here; Graphite implements a tile-based multicore, memory subsystem, and network, modeling performance and ensuring functional correctness. The default settings used for the various system configuration parameters are summarized in Table I.

Parameter	Settings
Cores	64 in-order, 5-stage pipeline, single-issue cores, 2-way fine-grain multithreading
L1/L2 cache per core	32/128 KB, 2/4-way set associative
Electrical network	2D Mesh, XY routing, 3 cycles per hop, 128b flits
	3.2 Kbits execution context size (cf. Section III-A)
	Context load/unload latency: $\left\lceil \frac{pkt\,size}{flit\,size} \right\rceil$ = 26 cycles
	Context pipeline insertion latency = 3 cycles
Data Placement	FIRST-TOUCH, 4 KB page size
Memory	30 GB/s bandwidth, 75 ns latency

TABLE I
SYSTEM CONFIGURATIONS USED

For data placement, we use the *first-touch after initialization* policy which allocates the page to the core that first accesses it after parallel processing has started. This allows private pages to be mapped locally to the core that uses them, and avoids all the pages being mapped to the same core where the main data structure is initialized before the actual parallel region starts.

B. Application benchmarks

Our experiments used a set of *Splash-2* [32] benchmarks: *fft*, *lu_contiguous*, *lu_non_contiguous*, *ocean_contiguous*, *ocean_non_contiguous*, *radix*, *raytrace* and *water-n*^{2}, and two in-house distributed hash table benchmarks: *dht_lp* for linear probing and *dht_sc* for separate chaining. We also used a modified set of *Splash-2* benchmarks [27]: *fft_rep*, *lu_rep*, *ocean_rep*, *radix_rep*, *raytrace_rep* and *water_rep*, where each benchmark was profiled and manually modified so that the frequently-accessed shared data are replicated permanently (for read-only data) or temporarily (for read-write data) among the relevant application threads. These benchmarks with careful replication[3] allow us to explore the benefits of thread migration on NUCA designs with more sophisticated data placement and replication algorithms like R-NUCA [15]. Rather than settling on and implementing one of the many automated schemes in the literature, we use modified Splash-2 benchmarks which implement all beneficial replications, and can serve as a reference placement/replication scheme.

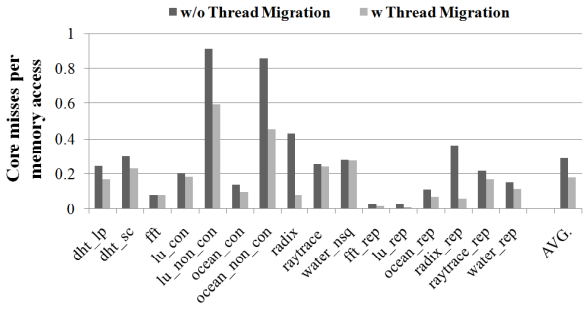

Fig. 4. The fraction of memory accesses requiring accesses to another core (i.e., core misses). The core miss rates decrease when thread migrations are effectively used.

Each application was run to completion using the recommended input set for the number of cores used. For each simulation run, we measured the average latency for memory operations as a metric of the average performance of the multicore system. We also tracked the number of memory accesses being served by either remote accesses or thread migrations.

C. Performance

We first compare the core miss rates for a NUCA system without and with thread migration: the results are shown in Figure 4. The depth threshold θ is set to 3 for our hybrid NUCA, which basically aims to perform remote accesses for memory sequences with one or two accesses and migrations for those with ≥ 3 accesses to the same core. We show how the results change with different values of θ in Section IV-D. While 29% of total memory accesses result in *core misses* for remote-access-only NUCA on average, NUCA with our migration predictor results in a core miss rate of 18%, which is a 38% improvement in data locality. This directly relates to better performance for NUCA with thread migration as shown in Figure 5. For our set of benchmarks, thread migration performance is no worse than the performance of the baseline NUCA and is better by up to 2.3X, resulting in 18% better performance on average (geometric mean) across all benchmarks.

Figure 6 shows the fraction of *core miss* accesses handled by remote accesses and thread migrations in our hybrid NUCA scheme. *Radix* is a good example where a large fraction of remote accesses are successfully replaced with a much smaller number of migrations: it originally showed 43% remote access rate under a remote-access-only NUCA (cf. Figure 4), but

[3]Our modifications were limited to rearranging and replicating the main data structures to take full advantage of data locality for shared data. Our modifications were strictly source-level, and did not alter the algorithm used.

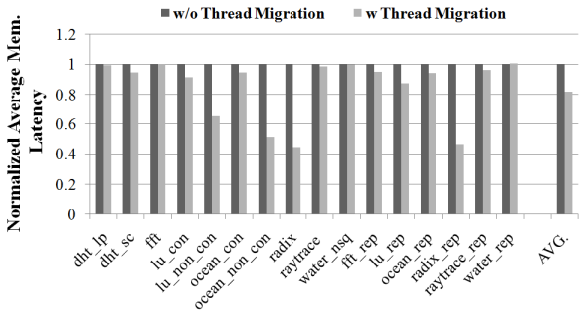

Fig. 5. Average memory latency of our hybrid NUCA (remote-access + thread migration) with $\theta = 3$ normalized to the baseline remote-access-only NUCA.

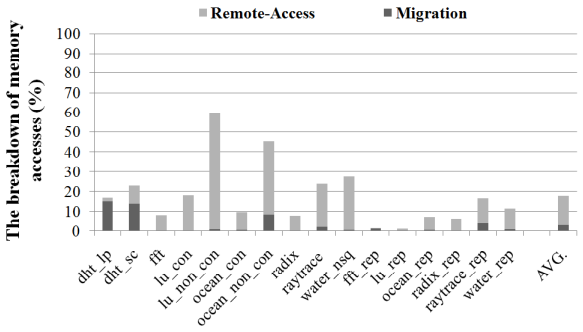

Fig. 6. The breakdown of core miss rates handled by remote accesses and migrations

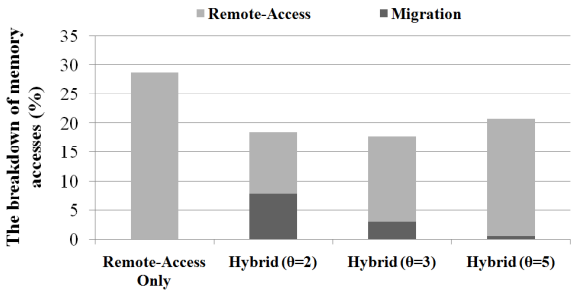

Fig. 7. The fraction of remote-accesses and migrations for the standard NUCA and hybrid NUCAs with the different depth thresholds (2, 3 and 5) averaged across all the benchmarks.

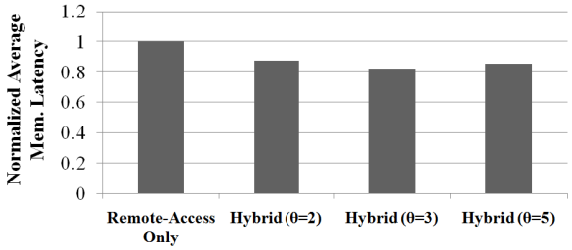

Fig. 8. Average memory latency of hybrid NUCAs with the different depth thresholds (2, 3 and 5) normalized to that of the standard NUCA averaged across all the benchmarks.

it decreases to 7.9% by introducing less than 0.01% of migrations, resulting in 5.4X less core misses in total. Across all benchmarks, the average migration rate is only 3% and these small number of thread migrations results in a 38% improvement in data locality (i.e., core miss rates) and an 18% improvement in overall performance.

D. Effects of the Depth Threshold

We change the value of the depth threshold $\theta = 2$, 3 and 5 and explore how the fraction of core-miss accesses being handled by remote-accesses and migrations changes. As shown in Figure 7, the ratio of remote-accesses to migrations increases with larger θ. The average performance improvement over the remote-access-only NUCA is 13%, 18% and 15% for the case of $\theta = 2$, 3 and 5, respectively (cf. Figure 8). The reason why $\theta = 2$ performs worse than $\theta = 3$ with almost the same core miss rate is because of its higher migration rate; due to the large thread context size, the cost of a single thread migration is much higher than that of a single remote access and needs, on average, a higher depth to achieve better performance.

V. Related Work

To provide faster access of large on-chip caches, the non-uniform memory architecture (NUMA) paradigm has been extended to single-die caches, resulting in a non-uniform cache access (NUCA) architecture [18], [9]. Data replication and migration, critical to the performance of NUCA designs, were originally evaluated in the context of multiprocessor NUMA architectures (e.g., [30]), but the differences in both interconnect delays and memory latencies make the general OS-level approaches studied inappropriate for today's fast on-chip interconnects.

NUCA architectures were applied to CMPs [3], [17] and more recent research has explored data distribution and migration among on-chip NUCA caches with traditional and hybrid cache coherence schemes to improve data locality. An OS-assisted software approach is proposed in [11] to control the data placement on distributed caches by mapping virtual addresses to different cores at page granularity. When adding affinity bits to TLB, pages can be remapped at runtime [15], [11]. The CoG [1] page coloring scheme moves pages to the "center of gravity" to improve data placement. The O^2 scheduler [6], an OS-level scheme for memory allocation and thread scheduling, improves memory performance in distributed-memory multicores by keeping threads and the data they use on the same core. Zhang proposed replicating recently used cache lines [33] which requires a directory to keep track of sharers. Reactive NUCA (R-NUCA) [15] obviates the need for a directory mechanism for the on-chip last-level cache by only replicating read-only data based on the premise that shared read-write data do not benefit from replication. Other schemes add hardware support for page migration support [8], [28]. Although manual optimizations of programs that take advantage of the programmer's application-level knowledge can replicate not only read-only data but also read-write shared data during periods when it is not being written [27], only read-only pages are candidates for replication for a NUCA substrate in general automated data

placement schemes. Instead of how to allocate data to cores, our work focuses on how to access the remote data that is not mapped to the local core, especially when replication is not an option. While prior NUCA designs rely on remote accesses with two-message round trips, we consider choosing between remote accesses and thread migrations based on our migration predictor to more fully exploit data locality.

Migrating computation to the locus of the data is not itself a novel idea. Hector Garcia-Molina in 1984 introduced the idea of moving processing to data in memory bound architectures [14]. In recent years migrating execution context has re-emerged in the context of single-chip multicores. Michaud shows the benefits of using execution migration to improve the overall on-chip cache capacity and utilizes this for migrating selective sequential programs to improve performance [21]. Computation spreading [7] splits thread code into segments and assigns cores responsible for different segments, and execution is migrated to improve code locality. A compile-time program transformation based migration scheme is proposed in [16] that attempts to improve remote data access. Migration is used to move part of the current thread to the processor where the data resides, thus making the thread portion local. In the design-for-power domain, rapid thread migration among cores in different voltage/frequency domains has been proposed to allow less demanding computation phases to execute on slower cores to improve overall power/performance ratios [24]. In the area of reliability, migrating threads among cores has allowed salvaging of cores which cannot execute some instructions because of manufacturing faults [23]. Thread migration has also been used to provide memory coherence among per-core caches [19], [20] using a deadlock-free fine-grained thread migration protocol [10]. We adopt the thread migration protocol of [10] for our hybrid memory access framework that supports both remote accesses and thread migrations. Although the hybrid architecture is introduced in [19], [20], they do not answer the question of how to effectively decide/predict which mechanism to follow for each memory access considering the tradeoffs between the two. This paper proposes a novel, PC-based migration predictor that makes these decisions at runtime, and improves overall performance.

VI. Conclusions and future work

In this manuscript, we have presented an on-line, PC-based thread migration predictor for memory access in distributed shared caches. Our results show that migrating threads for sequences of multiple accesses to the same core can improve data locality in NUCA architectures, and with our predictor, it can result in better overall performance compared to the traditional NUCA designs which only rely on remote-accesses.

Our future research directions include improving the migration predictor to better capture the dynamically changing behavior during program execution and to consider other factors than access sequence depths, such as distances or energy consumption. Furthermore, we will also explore how to reduce single-thread migration costs (i.e., the thread context size being transferred) by expanding the functionality of the migration predictor to predict and send only the useful part of the context in each migration.

References

[1] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *HPCA*, 2009.

[2] Moshe (Maury) Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. Analyzing parallel programs with pin. *Computer*, 43:34–41, 2010.

[3] M. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. In *MICRO*, 2004.

[4] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - processor: A 64-Core SoC with mesh interconnect. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 88–598, 2008.

[5] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of DAC*, pages 746–749, 2007.

[6] Silas Boyd-Wickizer, Robert Morris, and M. Frans Kaashoek. Reinventing scheduling for multicore systems. In *HotOS*, 2009.

[7] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Computation spreading: employing hardware migration to specialize CMP cores on-the-fly. In *ASPLOS*, 2006.

[8] M. Chaudhuri. PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches. In *HPCA*, 2009.

[9] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, 2003.

[10] Myong Hyon Cho, Keun Sup Shim, Mieszko Lis, Omer Khan, and Srinivas Devadas. Deadlock-free fine-grained thread migration. In *Proceedings of NOCS 2011*, pages 33–40, 2011.

[11] Sangyeun Cho and Lei Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *MICRO*, 2006.

[12] David Wentzlaff et al. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, Sept/Oct 2007.

[13] International Technology Roadmap for Semiconductors. Assembly and Packaging, 2007.

[14] H. Garcia-Molina, R.J. Lipton, and J. Valdes. A Massive Memory Machine. *IEEE Trans. Comput.*, C-33:391–399, 1984.

[15] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *ISCA*, 2009.

[16] Wilson C. Hsieh, Paul Wang, and William E. Weihl. Computation migration: enhancing locality for distributed-memory parallel systems. In *PPOPP*, 1993.

[17] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *ICS*, 2005.

[18] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *ASPLOS*, 2002.

[19] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Christopher W. Fletcher, Michel Kinsy, Ilia Lebedev, Omer Khan, and Srinivas Devadas. Brief announcement: Distributed shared memory based on computation migration. In *Proceedings of SPAA 2011*, pages 253–256, 2011.

[20] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Omer Khan, and Srinivas Devadas. Directoryless shared memory coherence using execution migration. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing*, 2011.

[21] P. Michaud. Exploiting the cache capacity of a single-chip multi-core processor with execution migration. In *HPCA*, 2004.

[22] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of HPCA 2010*, pages 1–12, 2010.

[23] Michael D. Powell, Arijit Biswas, Shantanu Gupta, and Shubhendu S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *Proceedings of ISCA 2009*, pages 93–104, 2009.

[24] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: Fine-grained power management for multi-core systems. In *Proceedings of ISCA 2009*, pages 302–313, 2009.

[25] S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Vora. A 45nm 8-core enterprise Xeon® processor. In *Proceedings of the IEEE Asian Solid-State Circuits Conference*, pages 9–12, 2009.

[26] K. Sankaralingam, R. Nagarajan, H. Liu, J. Huh, C. K. Kim, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture. In *International Symposium on Computer Architecture (ISCA)*, pages 422–433, June 2003.

[27] Keun Sup Shim, Mieszko Lis, Myong Hyo Cho, Omer Khan, and Srinivas Devadas. System-level Optimizations for Memory Access in the Execution Migration Machine (EM^2). In *CAOS*, 2011.

[28] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: increasing DRAM efficiency with locality-aware data placement. *SIGARCH Comput. Archit. News*, 38:219–230, 2010.

[29] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE J. Solid-State Circuits*, 43:29–41, 2008.

[30] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. *SIGPLAN Not.*, 31:279–289, 1996.

[31] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to Software: Raw Machines. In *IEEE Computer*, pages 86–93, September 1997.

[32] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[33] M. Zhang and K. Asanović. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *ISCA*, 2005.