Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2010-053 November 22, 2010

Scalable directoryless shared memory coherence using execution migration

Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Omer Khan, and Srinivas Devadas

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

CSAIL

Scalable directoryless shared memory coherence using execution migration

Mieszko Lis Keun Sup Shim Myong Hyon Cho Omer Khan Srinivas Devadas

Abstract—**We introduce the concept of deadlock-free migration-based coherent shared memory to the NUCA family of architectures. Migration-based architectures move threads among cores to guarantee sequential semantics in large multicores. Using a execution migration (EM) architecture, we achieve performance comparable to directory-based architectures without using directories: avoiding automatic data replication significantly reduces cache miss rates, while a fast network-level thread migration scheme takes advantage of shared data locality to reduce remote cache accesses that limit traditional NUCA performance.**

EM area and energy consumption are very competitive, and, on the average, it outperforms a directory-based MOESI baseline by 6.8% and a traditional S-NUCA design by 9.2%. We argue that with EM scaling performance has much lower cost and design complexity than in directory-based coherence and traditional NUCA architectures: by merely scaling network bandwidth from 128 to 256 (512) bit flits, the performance of our architecture improves by an additional 8% (12%), while the baselines show negligible improvement.

I. Background

Current trends in microprocessor design clearly indicate an era of multicores for the 2010s. As transistor density continues to grow geometrically, processor manufacturers are already able to place a hundred cores on a chip (e.g., Tilera Tile-Gx 100), with massive multicore chips on the horizon; many industry pundits are predicting 1000 or more cores by the middle of this decade [1]. Will the current architectures and their memory subsystems scale to hundreds of cores, and will these systems be easy to program?

The main barrier to scaling current memory architectures is the *off-chip memory bandwidth wall* [1], [2]: off-chip bandwidth grows with package pin density, which scales much more slowly than on-die transistor density [3]. Today's multicores integrate very large shared last-level caches on chip to reduce the number of off-chip memory accesses [4]; interconnects used with such shared caches, however, do not scale beyond relatively few cores, and the power requirements of large caches (which grow quadratically with size) exclude their use in chips on a 1000-core scale—for example, the Tilera Tile-Gx 100 does not have a large shared cache.

For massive-scale multicores, then, we are left with relatively small per-core caches. Since a programming model that relies exclusively on software-level message passing among cores is inconvenient and so has limited applicability, programming complexity considerations demand that the per-core caches must present a unified addressing space with coherence among caches managed automatically at the hardware level.

On scales where bus-based mechanisms fail, the traditional solution to this dilemma is directory-based cache coherence: a logically central directory coordinates sharing among the per-core caches, and each core cache must negotiate shared (read-only) or exclusive (read/write) access to each line via a complex coherence protocol. In addition to protocol complexity and the associated design and verification costs, directory-based coherence suffers from three other problems: (a) directory sizes must equal a significant portion of the *combined* size of the per-core caches, as otherwise directory evictions will limit performance [5]; (b) automatic replication of shared data significantly decreases the effective total on-chip cache size because, as the core counts grow, a lot of cache space is taken by replicas and fewer lines in total can be cached, which in turn leads to sharply increased off-chip access rates; and (c) frequent writes to shared data can result in repeated cache invalidations and the attendant long delays due to the coherence protocol.

Two of these shortcomings have been addressed by S-NUCA [6] and its variants. These architectures unify the per-core caches into one large shared cache, in their pure form keeping only one copy of a given cache line on chip and thus steeply reducing off-chip access rates compared to directory-based coherence. In addition, because only one copy is ever present on chip, cache coherence is trivially ensured and a coherence protocol is not needed. This comes at a price, however, as accessing data cached on a remote core requires a potentially expensive two-message round-trip: where a coherence protocol would take advantage of spatial and temporal locality by making a copy of the block containing the data in the local cache, S-NUCA must repeat the round-trip *for every access* to ensure sequential memory semantics. Various NUCA and hybrid proposals have therefore leveraged data migration and replication techniques previously explored in the NUMA context (e.g., [7]) to move private data to its owner core and replicate read-only shared data among the sharers at OS level [8], [2], [9] or aided by hardware [10], [11], [12], but while these schemes improve performance on some kinds of data, they still do not take full advantage of spatio-temporal locality and require either coherence protocols or repeated remote accesses to access read/write shared data.

To address this limitation and take advantage of available data locality in a memory organization where there is only one copy of data, we propose to allow computation threads to migrate from one core to another at a fine-grained instruction level. When several consecutive accesses are made to data assigned to a given core, migrating the execution 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. While computation migration, originally considered in the context of distributed multiprocessor architectures [13], has recently re-emerged at the single-chip multicores level, e.g., [14], [15], [16], for power management and fault-tolerance, we are unique in using migrations to provide memory coherence. We also propose a hybrid architecture that includes support for SNUCA-style remote access.

Specifically, in this paper we:

1) introduce the idea of using instruction-level execution migration (EM) to ensure memory coherence and sequential consistency in directoryless multicore systems with per-core caches;
2) present a provably deadlock-free hardware-level migration algorithm to move threads among the available cores with unprecedented efficiency;
3) combine execution migration (EM) with NUCA-style remote memory accesses (RA) to create a directoryless shared-memory multicore architecture which takes advantage of data locality.

II. Migration-based memory coherence

The essence of traditional distributed cache management in multicores is bringing data to the locus of the computation that is to be performed on it: when a memory instruction refers to an address that is not locally cached, the instruction stalls while either the cache coherence protocol brings the data to the local cache and ensures that the address can be safely shared or exclusively owned (in directory protocols) or a remote access is sent and a reply received (in S-NUCA).

Migration-based coherence brings the *computation* to the data: when a memory instruction requests an address not cached by the current core, the execution context (architecture state and TLB entries) moves to the core that is *home* for that data. As in traditional NUCA architectures, each address in the system is assigned to a unique core where it may be cached: the physical address space in the system is partitioned among the cores, and each core is responsible for caching its region.

Because each address can be accessed in at most one location, many operations that are complex in a system based on a cache coherence protocol become very simple: sequential consistency and memory coherence, for example, are ensured by default. (For sequential consistency to be violated, multiple threads must observe multiple writes in different order, which is only possible if they disagree about the value of some variable, for example, when their caches are out of sync. If data is never replicated, this situation never arises). Atomic locks work trivially, with multiple accesses sequentialized on the core where the lock address is located, and no longer ping-pong among core local caches as in cache coherence.

In what follows, we first discuss architectures based purely on remote accesses and purely on migration, and then combine them to leverage the strengths of both.

A. Basic remote-access-only (RA) architecture

In the remote-access (RA) architecture, equivalent to traditional S-NUCA, 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) 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 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) 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.

To avoid interconnect deadlock,[1] the system must ensure that all remote requests must always eventually be served; specifically, the following sequence, involving execution core C, home core H, and memory controller M, must always eventually make progress:

1) remote access request $C \rightarrow H$,
2) possible cache $\rightarrow$ DRAM request $H \rightarrow M$,
3) possible DRAM $\rightarrow$ cache response $M \rightarrow H$, and
4) remote access reply $H \rightarrow C$.

Since each step can only block and occupy resources (e.g., buffers) until the following steps complete, network messages induced by each later step must not be blocked at the network level by messages from a previous step belonging to another remote access. First, because steps 2 and 3 are optional, avoiding livelock requires traffic to be split into two independent virtual networks: one carrying messages for steps 1 and 4, and one for steps 2 and 3. Next, within each such subnetwork, the reply must have higher priority than the request. Finally, network messages between any two nodes within each subnetwork must be delivered in the order in which they were sent. With these rules, responses are always consumed and never blocked by requests, and the protocol always eventually makes progress.

B. Basic execution-migration-only (EM) architecture

In the execution-migration-only variant (EM), all non-local memory accesses cause the executing thread to be migrated to the core where the relevant memory address resides and executed there.

What happens if the target core is already running another thread? One option is to allow each single-issue core to round-robin execute several threads, which requires duplicate architectural state (register file, TLB); another is to evict the executing thread and migrate it elsewhere before allowing the new thread to enter the core. Our design features two execution contexts at each core: one for the core's *native* thread (i.e., the thread originally assigned there and holding its private data), and one for a *guest* thread. When an incoming guest migration encounters a thread running in the guest slot, this thread is evicted to its native core.

Thus, 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 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 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 DRAM).

Deadlock avoidance requires that the following sequence always eventually completes:

1) migration of T from $C \rightarrow H$,
2) possible eviction of T' from $H \rightarrow N'$,
3) possible cache $\rightarrow$ DRAM request $H \rightarrow M$, and
4) possible DRAM $\rightarrow$ cache response $M \rightarrow H$.

As with the remote-access-only variant from Section II-A, cache $\leftrightarrow$ memory controller traffic (steps 3 and 4) travels on one virtual network with replies prioritized over requests, and migration messages travel on another. Because DRAM $\rightarrow$ cache responses arrive at the requesting core, a thread with an outstanding DRAM request cannot be evicted until the DRAM response arrives; because this will always eventually happen, however, the eviction will eventually be able to proceed. Eviction migrations will always complete if (a) each thread T' has a unique native core N' which will always accept an eviction migration,[2] and (b) eviction migration traffic is prioritized over migrations caused by core misses. Since core-miss migrations can only be blocked by evictions, they will also always eventually complete, and the migration protocol is free of deadlock. Finally, to avoid migration livelock, it suffices to require each thread to complete at least one CPU instruction before being evicted from a core.

Because combining two execution contexts in one single-issue core may result in round-robin execution of the two threads, when two threads are active on the core they both experience a serialization effect: each thread is executing

[1]In the deadlock discussion, we assume that events not involving the interconnect network, such as cache and memory controller internals, always eventually complete, and that the interconnect network routing algorithm itself is deadlock-free or can always eventually recover from deadlock.

[2]In an alternate solution, where T' can be migrated to a non-native core such as T's previous location, a domino effect of evictions can result in more and more back-and-forth messages across the network and, eventually, deadlock.

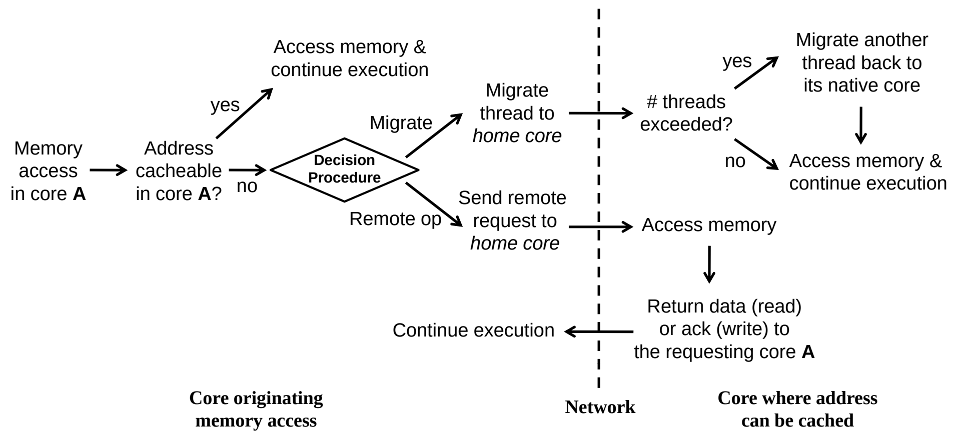

Memory access in core A
Address cacheable in core A?
yes
Access memory & continue execution
no
Decision Procedure
Migrate
Migrate thread to home core
Remote op
Send remote request to home core
# threads exceeded?
yes
Migrate another thread back to its native core
no
Access memory & continue execution
Access memory
Return data (read) or ack (write) to the requesting core A
Continue execution
Core originating memory access
Network
Core where address can be cached

Fig. 1. In the hybrid EM/RA architecture, memory accesses to addresses not assigned to the local core cause the execution context to be migrated to the core, or may result in a remote data access.

only 50% of the time. Although this seems like a relatively high overhead, observe that most of the time threads access private data and are executing on their native cores, so in reality the serialization penalty is not a first-order effect.

C. Hybrid architecture (EM/RA)

In the hybrid migration/remote-access architecture (EM/RA), 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. Because this decision must be taken on every access, it must be implementable as efficient hardware. In this paper, therefore, we consider and evaluate a simple heuristic scheme: the DISTANCE scheme. If the migration destination is the native core, the distance scheme always migrates; otherwise, it evaluates the hop distance to the home core. It migrates execution if the distance exceeds some threshold d else it makes a round-trip remote cache access.

In order to avoid deadlock in the interconnect, migrations must not be blocked by remote accesses and vice versa; therefore, a total of three virtual subnetworks (one for remote accesses, one for migrations, and one for memory traffic) are required. At the protocol level, evictions must now also wait for any outstanding remote accesses to complete in addition to waiting for DRAM $\rightarrow$ cache responses.

D. Migration framework

The novel architectural component we introduce here is fast, hardware-level migration of execution contexts between two cores via the on-chip interconnect network.

Since the core miss cost is dominated by the remote access cost and the migration cost, it is critical that the migrations be as efficient as possible. Therefore, unlike other thread-migration approaches (such as Thread Motion [17], which uses special cache entries to store thread contexts and leverages the existing cache coherence protocol to migrate threads), our architecture migrates threads directly over the interconnect network to achieve the shortest possible migration latencies.

Per-migration bandwidth requirements, although larger than those required by cache-coherent and remote-access-only designs, are not prohibitive by on-chip standards: in a 32-bit x86 processor, the relevant architectural state amounts, including TLB, to about 1.5 Kbits [17].

Figure 2 shows the differences needed to support efficient execution migration in a single-threaded five-stage CPU core. When both context slots (native and guest) are filled, execution round-robins between them to ensure that all threads can make progress. Register files now require wide read and write ports, as the migration logic must be able to unload all registers onto the network or load all registers from the network in relatively few cycles; to enable this, extra muxing logic connects the register files directly with the on-chip network router. The greater the available network bandwidth, the faster the migration. As with traditional S-NUCA architectures, the memory subsystem itself is connected to the on-chip network router to allow for accesses to the off-chip memory controller as well as for reads and writes to a remote cache (not shown in the figure).

E. Data placement

The assignment of addresses to cores affects the performance of EM/RA in three ways: (a) because context migrations pause thread execution and therefore longer migration distances will slow down performance; (b) because remote accesses also pause execution and longer round trips will also limit performance; and (c) indirectly by influencing cache performance. On the one hand, spreading frequently used addresses evenly among the cores ensures that more addresses are cached in total, reducing cache miss rates and, consequently, off-chip memory access frequency; on the other hand, keeping addresses accessed by the same thread in the same core cache reduces migration rate and network traffic.

As in standard S-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 are possible: for example, in a first-touch-style scheme, the OS can map the page to the thread's *native* core, taking advantage of data access locality to reduce the migration rate while keeping the threads spread among cores.

In EM/RA architectures, data placement is key, as it determines the frequency and distance of remote accesses and migrations. Although placement has been studied extensively in the context of NUMA architectures (e.g., [7]) as well as more recently in NUCA context (e.g., [2]), we wish to concentrate here on the potential of the EM/RA architecture and implement none of them directly. Instead, we combine a first-touch data placement policy [18], which maps each page to the first core to access it, with judicious profiling-based source-level modifications to our benchmark suite (see Section III-C) to provide placement and replication on par or better than that of available automatic methods.

III. Methods

A. Architectural simulation

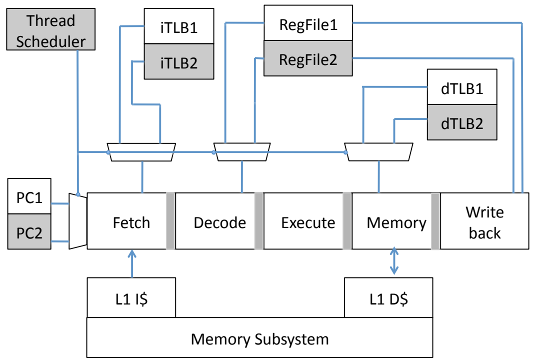
We use Pin [19] and Graphite [20] to model the proposed execution migration (EM), remote-access (RA) and hybrid (EM/RA) architectures as well as the cache-coherent (CC) baseline. Pin enables runtime binary instrumentation of parallel programs, including the SPLASH-2 [21] benchmarks we use here; Graphite 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; any deviations are noted when results are reported. In experiments comparing EM/RA architectures against CC, the parameters for both were identical, except for (a) the memory directories which are not needed for EM/RA and were set to sizes recommended by Graphite on basis of the total cache capacity in the simulated system, and (b) the 2-way multithreaded cores which are not needed for cache-coherent baseline.

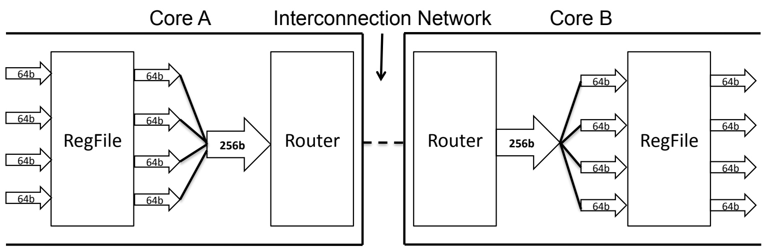
To exclude differences resulting from relative scheduling of Graphite threads, data were collected using a homogeneous cluster of machines.

B. On-chip interconnect model

Experiments were performed using Graphite's model of an electrical mesh network with XY routing with 128-bit flits. Since modern network-on-chip routers are pipelined [22], and 2- or even 1-cycle per hop router latencies [23] have been demonstrated, we model a 2-cycle per hop router delay; we also account for the appropriate pipeline latencies associated with loading and unloading a packet onto the network. In addition to the fixed per-hop



(a) The architecture of a 2-way single-issue slot multithreaded core for EM^2



(b) Microarchitecture for a single context transfer in EM^2

Fig. 2. (a) A single-issue five-stage pipeline with efficient context migration; differences from a single-threaded pipeline are shaded. (b) For a context transfer, the register file of the originating core is unloaded onto the router, transmitted across the network and finally loaded onto the home core's register file via the router.

Parameter	Settings
Cores	256 in-order, 5-stage pipeline, single issue cores
	2-way fine-grain multithreading
L1 instruction/L1 data/L2 cache per core	32/16/64 KB, 4/2/4-way set associative
Electrical network	2D Mesh, XY routing, 2 cycles per hop (+ contention), 128b flits
	1.5 Kbits execution context size (similar to [17])
	Context load/unload latency: $\left\lceil \frac{pkt\,size}{flit\,size} \right\rceil$ = 12 cycles
	Context pipeline insertion latency = 3 cycles
Data Placement scheme	FIRST-TOUCH, 4 KB page size
Coherence protocol	Directory-based MOESI, Full-map distributed directories = 8
	Entries per directory = 32768, 16-way set associative
Memory	30 GB/s bandwidth, 75 ns latency

TABLE I
SYSTEM CONFIGURATIONS USED

latency, contention delays are modeled using a probabilistic model similar to the one proposed in [24].

C. Application benchmarks

Our experiments used a set of SPLASH-2 benchmarks: FFT, LU_CONTIGUOUS, OCEAN_CONTIGUOUS, RADIX, RAYTRACE, and WATER-N^2. For the benchmarks for which versions optimized for cache coherence exist (LU and OCEAN [25], [21]), we chose the versions that were most optimized for directory-based cache coherence.

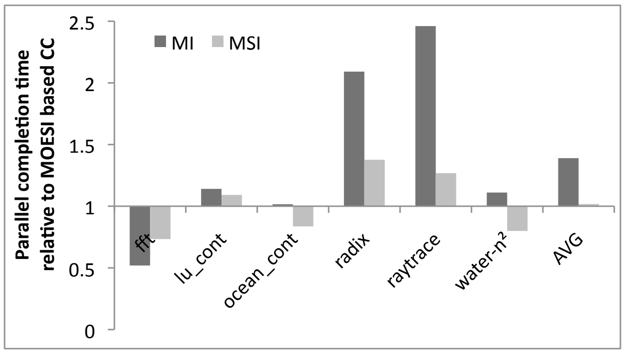
Application benchmarks tend not to perform well in RA architectures with simple striped data placements [2], and sophisticated data placement and replication algorithms like R-NUCA [2] are required for fair comparisons. Rather than picking one of the many automated schemes in the literature, we first profiled and manually modified each benchmark to detect shared data structures that caused many non-local memory accesses; then, we manually modified the frequently accessed shared data to replicate them permanently (for read-only data) or temporarily (for read-write data) among the relevant application threads. Since automated replication and placement methods cannot take advantage of such application-specific techniques as temporarily replicating read-write data, our scheme is in some sense superior to all available automatic methods. Our optimizations were limited to rearranging, and replicating the main data structures to take full advantage of the first-touch page allocation scheme. Our optimizations were strictly source-level, and did not alter the algorithm used. Together with a first-touch page placement policy implemented in our architectural simulator, our modifications serve as a reference placement/replication scheme. As such, the changes did not affect the operation of the cache coherence protocol, and the performance differences were negligible: on average, our EM/RA-optimized version was about 2 % faster than the cache-coherence-optimized variant when run on the cache-coherent baseline.

Each application was run to completion using the recommended input set for the number of cores used. For each simulation run, we tracked the total application completion time, the parallel work completion time, the percentage of memory accesses causing cache hierarchy misses, and the percentage of memory accesses causing migrations. While the total application completion time (wall clock time from application start to finish) and parallel work completion time (wall clock time from the time the second thread is spawned until the time all threads re-join into one) show the same general trends, we focused on the parallel work completion time as a more accurate metric of average performance in a realistic multicore system with many applications.

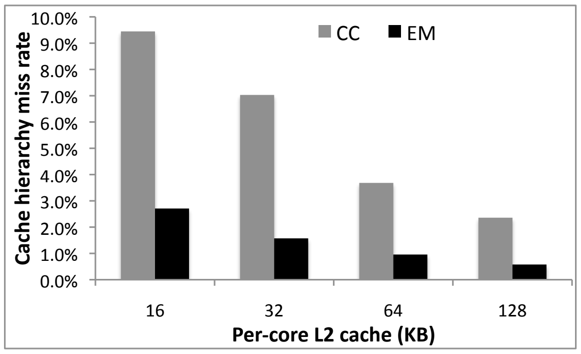
D. Directory-based cache coherence baseline selection

In order to choose a directory-based coherence (CC) baseline for comparison, we considered the textbook protocol with Modified/Shared/Invalid (MSI) states as well as two alternatives: on the one hand, data replication can be completely abandoned by only allowing modified or invalid states (MI); on the other hand, in the presence of data replication, off-chip access rates can be lowered via protocol extensions such as an *owned* and *exclusive* states (MOESI) combined with cache-to-cache transfers whenever possible.

To evaluate the impact of these variations, we compared the performance for various SPLASH-2 benchmarks under MSI, MI, and MOESI (using parameters from Table I). As



(a) parallel completion time under different CC protocols, normalized to MOESI



(b) cache miss rates under CC and NUCA at various cache sizes

Fig. 3. (a) Although parallel completion time for different coherence protocols varies somewhat across the benchmarks (notably, the high directory eviction rate in FFT leads to rampant invalidations in MSI and MOESI and favors MI), generally MOESI was the most efficient protocol and MI performed worst. (b) Cache hierarchy miss rates at various cache sizes show that, by eschewing replication, the EM/RA architecture achieves cache miss rates much lower than the CC baseline at all cache sizes. (Settings from Table I).

shown in Figure 3a, MI exhibits by far the worst memory latency: although it may at first blush seem that MI removes sharing and should thus improve cache utilization much like EM/RA, in actuality eschewing the *S* state only spreads the sharing—and the cache pollution which leads to capacity misses—over time when compared with MSI and MOESI. At the same time, MI gives up the benefits of read-only sharing and suffers many more cache evictions: its cache miss rates were $2.3\times$ greater than under MSI. The more complex MOESI protocol, meanwhile, stands to benefit from using cache-to-cache transfers more extensively to avoid writing back modified data to off-chip RAM, and take advantage of exclusive cache line ownership to speed up writes. Our analysis shows that, while cache-to-cache transfers result in many fewer DRAM accesses, they instead induce significantly more coherence traffic (even shared reads now take 4 messages); in addition, they come at a cost of significantly increased protocol, implementation and validation complexity. Nevertheless, since our simulations indicate (Figure 3a) that MOESI is the best-performing coherence protocol out of the three, we use it as a baseline for comparison in the remainder of this paper.[3]

Finally, while we kept the number of memory controllers fixed at 8 for all architectures, for the cache-coherence baseline we also examined several ways of distributing the directory among the cores via Graphite simulations: central, one per memory controller, and fully distributed. On the one hand, the central directory version caused the highest queueing delays and most network congestion, and, while it would require the smallest total directory size, a single directory would still be so large that its power demands would put a significant strain on the 256-core chip.[4] On the other end of the spectrum, a fully distributed directory would spread congestion among the 256 cores, but each directory would have to be much larger to allow for imbalances in accesses to cache lines in each directory, and DRAM accesses would incur additional network latencies to contact the relatively few memory controllers. Finally, we considered the case of 8 directories (one for each of the 8 memory controllers), which removed the need for network messages to access DRAM and performed as well as the best-case fully distributed variant. Since the 8-directory configuration offered best performance and a good tradeoff between directory size and contention, we used this design in our evaluation.

E. *Remote-access NUCA baseline selection*

To compare against an RA architecture baseline, we considered two approaches: the traditional S-NUCA approach where the L1 and L2 caches are shared (that is, a local L1 or L2 may cache only a subset of the address space), and a hybrid NUCA/coherence approach where private L1 caches are maintained via a coherence protocol. Although the hybrid variant offers some relief from remote accesses to frequently used locations, the L1 caches must keep very large full-map directories (significantly larger than total cache on the core [2]!): if the directories are too small, the L1's will suffer frequent invalidations due to directory evictions and the combined performance will revert towards a remote-access-only design. Based on these considerations we chose to compare our hybrid architecture to a fully shared L1/L2 remote-access-only baseline.

[3]We use MSI in our analytical model (Section IV-A, Appendix) for simplicity.

[4]power demands scale quadratically with SRAM size

Component	#	Total area (mm^2)	Read energy (nJ/instance)	Write energy (nJ/instance)	Details
Register file	256	2.48	0.005	0.002	4-Rd, 4-Wr ports; 64x24 bits
Router	256	7.54	0.011	0.004	5-Rd, 5-Wr ports; 128x20 bits
Directory cache	8	9.06	1.12	1.23	1MB cache (16-way assoc)
L2 Cache	256	26.65	0.086	0.074	64KB (4-way assoc)
L1 Data Cache	256	6.44	0.034	0.017	16KB cache (2-way assoc)
Off-chip DRAM	8	N/A	6.333	6.322	1GB RAM

TABLE II
AREA AND ENERGY ESTIMATES

F. Cache size selection

We ran our SPLASH-2 simulations with a range of cache sizes under both an execution-migration design and the cache-coherent baseline. While adding cache capacity improves cache utilization and therefore performance for both architectures, cache miss rates are much lower for the migration-based approach and, with much smaller on-chip caches, EM/RA achieves significantly better results (Figure 3). When caches are very large, on the other hand, they tend to fit most of the working set of our SPLASH-2 benchmarks and both designs almost never miss the cache. This is, however, not a realistic scenario in a system concurrently running many applications: we empirically observed that as the input data set size increases, larger and larger caches are required for the cache-coherent baseline to keep up with the migration-based design. To avoid bias either way, we chose realistic 64KB L2 data caches as our default configuration because it offers a reasonable performance tradeoff and, at the same time, results in a massive 28Mbytes of on-chip total cache (not including directories for CC).

G. Instruction cache

Since the thread context transferred in an EM architecture does not contain instruction cache entries, we reasoned that the target core might not contain the relevant instruction cache lines and a thread might incur an instruction cache miss immediately upon migration. To evaluate the potential impact of this phenomenon, we compared L1 instruction cache miss rates for EM and the cache-coherent baseline in simulations of our SPLASH-2 multithreaded benchmarks.

Results indicated an average instruction cache miss rate of 0.19% in the RA design as compared to 0.27% in the CC baseline. The slight improvement seen in RA is due to the fact non-memory instructions are always executed on the core where the last memory access was executed (since only another memory reference can cause a migration elsewhere), and so non-memory instructions that follow references to shared data are cached only on the core where the shared data resides.

H. Area and energy estimation

For area and energy, we assume 32nm process technology and use CACTI [26] to estimate the area requirements of the on-chip caches and interconnect routers. To estimate the area overhead of extra hardware context in the 2-way multithreaded core for EM, we used Synopsys Design Compiler [27] to synthesize the extra logic and register-based storage. We also use CACTI to estimate the dynamic energy consumption of the caches, routers, register files, and DRAM. The area and dynamic energy numbers used in this paper are summarized in Table II. We implemented several energy counters (for example the number of DRAM reads and writes) in our simulation framework to estimate the total energy consumption of running SPLASH-2 benchmarks for both CC and EM. Note that DRAM only models the energy consumption of the RAM and the I/O pads and pins will only add to the energy cost of going off-chip.

IV. RESULTS AND ANALYSIS

Intuitively, replacing off-chip memory traffic due to (hopefully infrequent) cache misses with possibly much more frequent on-chip thread migration traffic or remote accesses may not seem like an improvement. To gain some intuition for where migration-based architectures can win on performance, we first consider the average memory latency (AML), a metric that dominates program execution times with today's fast cores and relatively slow memories. Under an EM/RA architecture, AML has three components: cache access (for cache hits and misses), off-chip memory access (for cache misses), and context migration or remote access cost (for core misses):

$$\begin{aligned} AML = {} & cost_{\$access} + rate_{\$miss} \times cost_{\$miss} \\ & + rate_{core\,miss} \times cost_{remote\,access/context\,xfer} \end{aligned}$$

% of non-memory instructions	70%
% of memory instructions accessing shared data	10%
% of memory instructions accessing private data	20%
% of read-only data in shared data	{25%, 75%, 95%, 100%}
Load:store ratio	2:1
Private data per thread	16 KB
Total shared data	1 MB
Degree of sharing	{1, 2, 4, 8, 32, 64, 128, 256}
Number of instructions per thread	100,000

TABLE III
SYNTHETIC BENCHMARK SETTINGS

While $cost_{\$access}$ mostly depends on the cache technology itself, EM/RA architectures improve performance by optimizing the other variables: in the remainder of this section, we show how our architecture improves $rate_{\$miss}$ when compared to a CC baseline, discuss several ways to keep $rate_{core\,miss}$ low, and argue that today's interconnect technologies keep $cost_{context\,xfer}$ sufficiently low to ensure good performance.

A. Memory access costs analysis

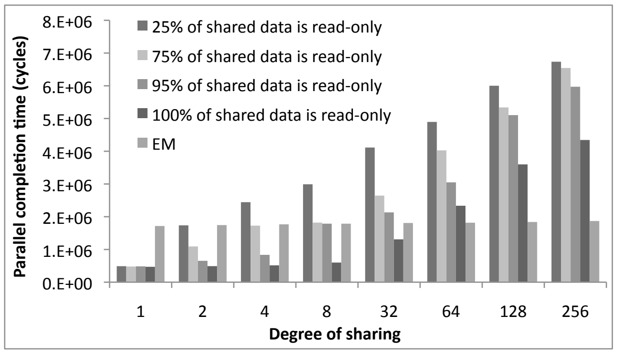
To understand memory access latency tradeoffs under cache coherence (CC) and our designs based on execution migration (EM), we broke down memory accesses into fundamental components (such as DRAM access itself or coherence traffic) and created an analytical model to estimate average memory access latency (see the Appendix). We then measured parameters like cache and core miss rates by running the OCEAN_CONTIGUOUS benchmark in our simulator under both an MSI coherence protocol and a migration architecture (we chose OCEAN_CONTIGUOUS because it had the high core miss rate, which avoids pro-EM bias). Applying these to the measured parameters (see Appendix A) shows that, on the average, EM memory accesses in the OCEAN_CONTIGUOUS benchmark take $\mathbf{1.5}\times$ less time.

Although the memory latency model does not account for some effects (for EM, the possibility of 2:1 serialization when both contexts on a given core are filled, and for CC, invalidations due to directory evictions or the extra delays associated with sending invalidations to *many* core caches and waiting for their responses), it gives a flavor for how EM might scale as the number of cores grows. Centralized effects like off-chip memory contention and network congestion around directories, which limit performance in CC, will only increase as the ratio of core count to off-chip memory bandwidth increases. Performance-limiting costs under EM, on the other hand, are either decentralized (core migrations are distributed across the chip) or much smaller (contention for off-chip memory is much lower because cache miss rates are small compared to CC), and will scale more gracefully.

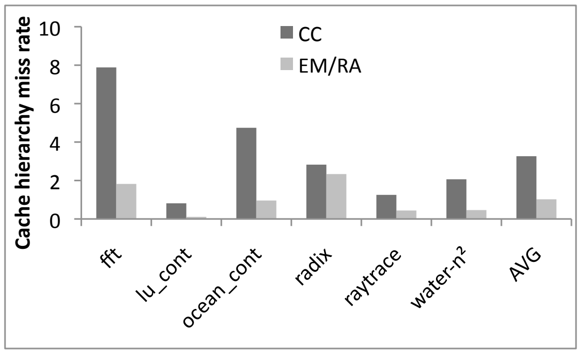
B. Advantages over directory-based cache coherence

Since the main benefit of remote-access and migration architectures over cache coherence protocols comes from improving on-chip cache utilization by not replicating writable shared data and minimizing cache capacity/conflict misses, we next investigated the impact of data sharing in cache-coherent architectures. With this in mind, we created synthetic benchmarks that randomly access addresses with varying degrees of read-only sharing and read-write sharing (see Table III). The benchmarks vary along two axes: the fraction of instructions that access read-only data, and the degree of sharing of the shared data: for example, for read-write shared data, degree d denotes that this data can be read/written by up to d sharers or threads. We then simulated the benchmarks using our cache-coherent (MOESI) baseline (Table I), and measured parallel application performance (which, unlike our memory latency model above, includes effects not directly attributable to memory accesses like serialization or cache/directory eviction costs).

Figure 4a shows that cache coherence performance worsens rapidly as the degree of sharing increases. This is for two reasons: one is that a write to shared data requires that all other copies be invalidated (this explains the near-linear growth in parallel completion time when most accesses are writes), and the other is that even read-only sharing causes one address to be stored in many core-local caches, reducing the amount of cache left for other data (this is responsible for the slower performance decay of the 100% read-only benchmarks). These results neatly illustrate the increasing challenge of data sharing with CC designs as the number of cores grows: even in the unrealistic case where *all* shared data is read-only, the higher cache miss rates of



(a) CC performance drops as the degree of sharing grows



(b) EM/RA markedly lowers miss rates

Fig. 4. (a) The performance of CC (under a MOESI protocol) degrades as the degree of sharing increases: for read-write sharing this is due to cache evictions, and for read-only sharing to reduced core-local cache effectiveness when multiple copies of the same data are cached. Under EM performance degrades much more slowly. (b) For our benchmarks, under our EM/RA designs the cache miss rates are on the average $3.2\times$ lower because storing each cache line in only one location eliminates many capacity and coherence-related evictions and effectively increases the availability of cache lines.

cache coherence cause substantial performance degradation for degrees of sharing greater than 32; when more and more of the shared data is read-write, performance starts dropping at lower and lower degrees of sharing. With an EM architecture, on the other hand, each address—even shared by multiple threads—is still assigned to only one cache, leaving more total cache capacity for other data and EM's performance degrades much more slowly.

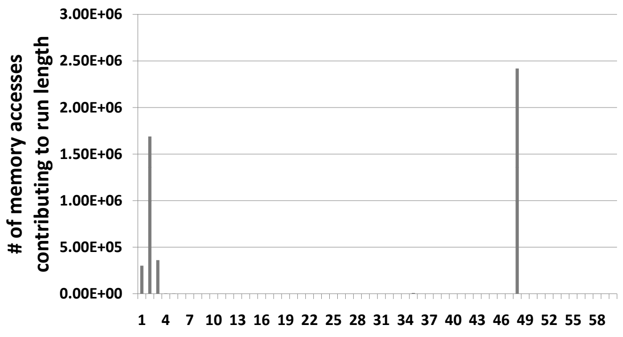
Because the additional capacity arises from not storing addresses in many locations, cache miss rates naturally depend on the memory access pattern of specific applications; we therefore measured the differences in cache miss rates for several benchmarks between our EM/RA designs and the CC baseline. (Note that the cache miss rates are virtually identical for all our EM/RA designs). The miss rate differences in realistic benchmarks, shown in Figure 4b, are attributable to two main causes. On the one extreme, the FFT benchmark does not exhibit much sharing and the high cache miss rate of 8% for MOESI is due mainly to significant directory evictions; since in the EM/RA design the caches are only subject to capacity misses, the cache miss rate falls to under 2%. At the other end of the spectrum, the OCEAN_CONTIGUOUS benchmark does not incur many directory evictions but exhibits significant read-write sharing, which, in directory-based cache coherence (CC), causes mass invalidations of cache lines actively used by the application; at the same time, replication of the same data in many per-core caches limits effective cache capacity. This combination of capacity and coherence misses results in a 5% miss rate under MOESI; the EM/RA architecture eliminates the coherence misses and increases effective cache capacity, and only incurs a 0.8% miss rate. The remaining benchmarks fall in between these two extremes, with a combination of directory evictions and read-write sharing patterns.

Cache miss rates illustrate the core potential advantage of EM/RA designs over CC: significantly lower off-chip access rates given the same cache sizes. Although miss rates in CC architectures can be reduced by increasing the per-core caches, our simulation results (not shown here) indicate that, overall, the CC design would need in excess of $2\times$ the L2 cache capacity to match the cache miss rates of EM/RA.

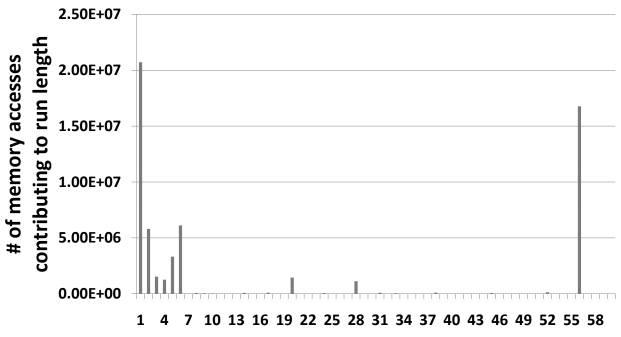
C. *Advantages over traditional directoryless NUCA (RA)*

Although RA architectures eschew automatic sharing of writable data and significantly lower cache miss rates, their main weakness lies in not being able to take advantage of shared data locality: even if many consecutive accesses are made to data on the same remote core, sequential consistency requires that each be an independent round-trip access. To examine the extent of this problem, we measured the *run length* for non-local memory access: the number of consecutive accesses to memory cached in a non-local core not interrupted by any other memory accesses.

Figure 5 shows this metric for two of our benchmarks. Predictably, the number of remote accesses with run length of one (a single access to a remote core followed by access to another remote core or the local core) is high; more significantly, however, a great portion of remote memory accesses in both benchmarks shown exhibit significant core locality and come in streaks of 40–50 accesses. Although core locality is not this dramatic in all applications, these examples show precisely where a migration-based architecture shines: the executing thread is migrated to a remote core and 40–50 now effectively "local" memory accesses are made before incurring the cost of another migration.

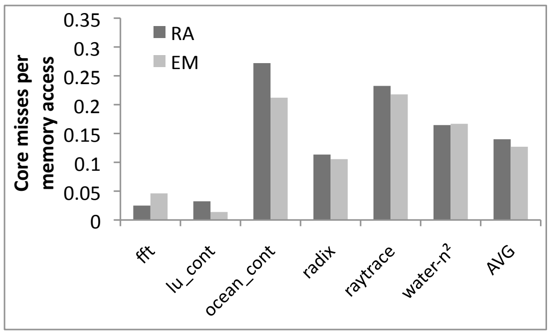


(a) LU_CONTIGUOUS

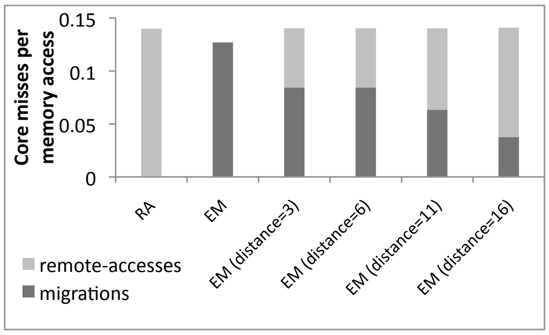


(b) OCEAN_CONTIGUOUS

Fig. 5. Non-local memory accesses in our RA baseline binned by the number of surrounding contiguous accesses to the same remote core. Although, predictably, many remote operations access just one address before accessing another core, a surprisingly large number belong to streaks of 40–50 accesses to the same remote core and indicate significant data locality.



(a) per-benchmark core miss rates



(b) core miss rates handled by remote accesses (light) and migrations (dark), averaged over all benchmarks

Fig. 6. The potential for improvement over the RA baseline. (a) When efficient core-to-core thread migrations are allowed, the number of memory accesses requiring transition to another core (core misses) significantly decreases. (b) The fraction of core miss rates handled by remote accesses and migrations in various migration/remote-access hybrids shows that the best-performing scheme, EM(distance=11), has significant migration and remote access components.

To examine the real improvement potential offered by extending RA with efficient execution migrations, we next counted the *core miss* rates—the number of times a round-trip remote-access or a migration to a remote core must be made—for the RA baseline and our EM architecture.

Figure 6a shows core misses across a range of benchmarks. As we'd expect from the discussion above (Section IV-C), OCEAN_CONTIGUOUS and LU_CONTIGUOUS show that migrations significantly lower core miss rates, and most other benchmarks also improve. The outlier here is FFT: most of the accesses it makes are to each thread's private data, and shared accesses are infrequent and brief.

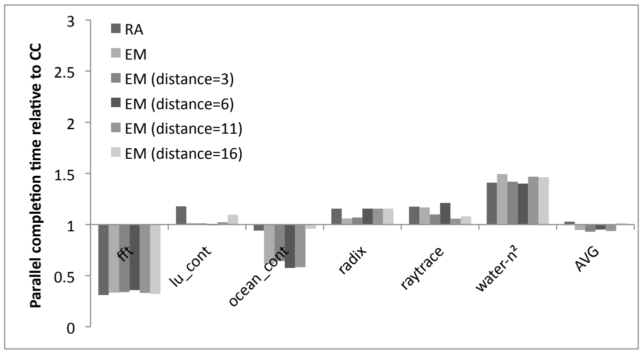
Figure 6b shows how many overall core misses were handled by remote accesses and migrations in several EM/RA variants. In the EM/RA scheme that performed best, namely, EM(distance=11), see Figure 7a, both migrations and remote access play a significant role, validating our intuition behind combining them into a hybrid architecture.

It is important to note that the cost of core misses is very different under RA and under EM: in the first, each core miss induces a *round-trip* remote access, while in the second it causes a *one-way* migration (the return migration, if any, is counted as another core miss). Adding efficient migrations to an RA design therefore offers significant performance potential, which we examine next.

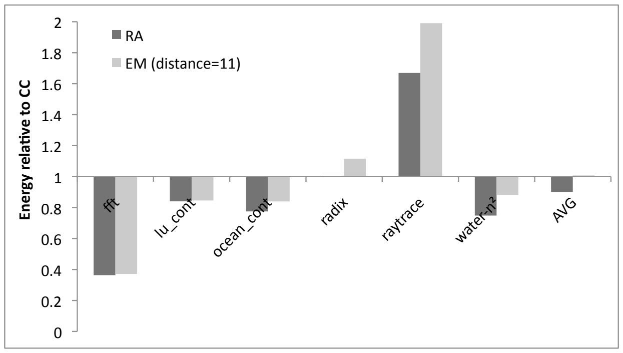
D. Overall area, performance and energy

The EM/RA architectures do not require directories and as can be seen from Table II are over 6 mm^2 smaller than the CC baseline.

Figure 7a shows the parallel completion time speedup relative to the CC baseline for various EM/RA schemes: a remote-access-only variant, a migrations-only variant, and a range of hybrid schemes where the remote-access vs. migration decision is based on on hop distance. Overall, performance is very competitive with the cache coherent

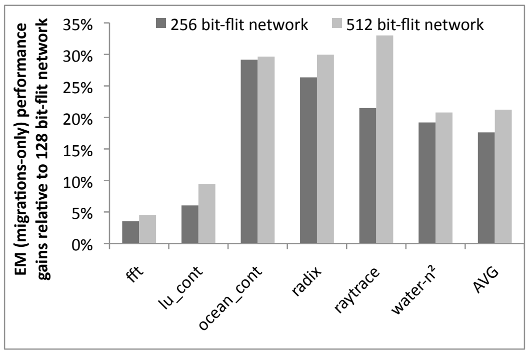


(a) parallel completion time vs. MOESI



(b) dynamic energy vs. MOESI

Fig. 7. (a) The performance of EM/RA variants relative to CC. Although results vary significantly by benchmark, the best EM/RA scheme, namely, EM(distance=11) outperforms CC by 6.8% on average. (b) Dynamic energy usage for the EM(distance=11) scheme is virtually identical to CC, and the RA variant consumes the least amount of energy. The energy numbers do not include I/O pad and pin energy which favors directory-based coherence.



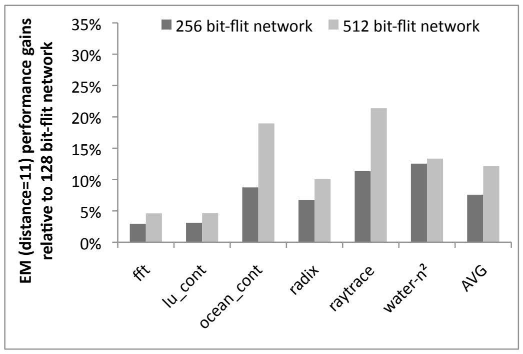


Fig. 8. EM (migrations only and distance=11) performance scales with network bandwidth. On the other hand, remote-access and directory-based coherence are primarily sensitive to latency and their performance only scales by 1%-2% when network bandwidth is increased (not shown in the figure).

baseline and the best EM/RA design EM(distance=11) shows an average 6.8% improvement over the CC baseline.

The benefits are naturally application-dependent: as might be expected from Figure 6a, the benchmarks with the largest cache miss rate reductions (FFT and OCEAN_CONTIGUOUS) offer the most performance improvements. At the other extreme, the WATER benchmark combines fairly low cache miss rates under CC with significant read-only sharing, and is very well suited for directory-based cache coherence; consequently, CC outperforms all EM/RA variants by a significant margin.

The result also shows the benefits of a combined EM/RA architecture: in some benchmarks (e.g., LU_CONTIGUOUS, OCEAN_CONTIGUOUS, RADIX), a migration-only design significantly outperforms remote accesses, while in others (e.g., WATER-N^2) the reverse is true. On average, the best distance-based EM/RA hybrid performs better than either EM or RA, and renders the EM/RA approach highly competitive with directory-based MOESI cache coherence.

Since energy dissipated per unit performance will be a critical factor in next-generation massive multicores, we employed a power model (cf. Section III) to estimate the dynamic power consumed by the various EM/RA variants and CC. On the one hand, migrations incur significant dynamic energy costs due to increased traffic in the on-chip network and the additional register file per core; on the other hand, dramatic reductions in off-chip accesses equate to very significant reductions in DRAM access energy.

As illustrated in Figure 7b, energy consumption depends on each application's access patterns. For FFT, for example, which incurs crippling rates of eviction invalidations, the energy expended by the CC protocol messages and DRAM references far outweighs the cost of energy used by remote accesses and migrations. On the other extreme, the fairly random patterns of memory accesses in RAYTRACE, combined with a mostly private-data and read-only sharing paradigm, allows CC to efficiently keep data in the core caches and consume far less energy than EM/RA. The high cost of off-chip DRAM accesses is particularly highlighted in the WATER-N^2 benchmark: although the trend in cache miss rates between CC and EM/RA is similar for WATER-N^2 and RAYTRACE, the overall cache miss rate is markedly higher in WATER-N^2; combined with the associated protocol costs, the resulting off-chip DRAM accesses make the CC baseline consume more energy than the EM/RA architecture.

We note that our energy numbers for directory-based coherence are quite optimistic, since we did not include energy consumed by I/O pads and pins; this will result in higher energy for off-chip accesses which CC makes more of.

E. Performance scaling potential for EM designs

Finally, we investigated the scaling potential of the EM architecture. We reasoned that, while directory-based cache coherence is limited by cache sizes and off-chip DRAM bandwidth and RA performance is restricted by interconnect *latencies*, EM can be improved by increasing interconnect *bandwidth*: with higher on-chip network bandwidth, the main effect is that messages carrying the execution context consume fewer cycles, and a smaller effect is that they experience less congestion.

As illustrated in Figure 8, increasing the network bandwidth significantly improves performance of a migration-only EM variant (as well as the best EM/RA hybrid with distance=11), especially on migration-limited benchmarks like RAYTRACE. Since scaling of network bandwidth is easy—although buffers and crossbars must be made wider so area increases linearly, the fundamental design of the interconnect remains constant and the clock frequencies are not appreciably affected. Moreover, since the same amount of data must be transferred, dynamic power consumption does not grow in tandem. Contrasted with the off-chip memory bandwidth wall and quadratically growing power requirements of large caches limiting cache-coherent architecture performance on the one hand, and the difficulty in reducing electrical network hop counts limiting remote-access performance on the other hand, an EM or EM/RA architecture offers a straightforward and attractive way to significantly increase performance at sublinear impact on cost and virtually no impact on verification.

V. Related work

Implicitly moving data to computation has been explored in great depth with many years of research on cache coherence protocols, and has become textbook material [28]. Meanwhile, in the past decade, the non-uniform memory architecture (NUMA) paradigm has been extended to single-die caches resulting in a non-uniform cache access (NUCA) architecture [6], [29], and applied to single-chip multicores [30], [31]. Data replication and migration, critical to the performance of NUCA designs, were originally evaluated in the context of multiprocessor NUMA architectures (e.g., [7]), 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. More recent research has explored data distribution and migration among on-chip NUCA caches with traditional and hybrid cache coherence schemes. An OS-assisted software approach is proposed in [8] 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 [2], [8]. The CoG [9] page coloring scheme moves pages to the "center of gravity" to improve data placement. The O^2 scheduler [32], 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. These data placement optimizations are generally applicable to NUCA architectures including migration-based variants. Rather than implement specific allocation schemes, our evaluations combine a first-touch data placement mechanism with manual application optimizations to reach near-optimal placement and replication of shared read-only and read-write data on par or better than that obtained with the automatic schemes.

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 [13]. 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 [14]. Computation spreading [16] splits thread code into segments and assigns cores responsible for different segments, and execution is migrated to improve code locality. Kandemir presents a data migration algorithm to address the data placement problem in the presence of non-uniform memory accesses within a traditional cache coherence protocol [15]. This work attempts to find an optimal data placement for cache lines. A compile-time program transformation based migration scheme is proposed in [33] 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; this scheme allows programmer to express when migration is desired. Dataflow machines (e.g., [34])—and, to some extent, out-of-order execution—are superficially similar as they allow an activated instruction to be claimed by any available execution unit, but cannot serve as a shared-memory abstraction. The J-machine [35] ties processors to on-chip memories, but relies on user-level messaging and does not address the challenge of off-chip memory bandwidth. Our proposed execution migration machine is unique among the previous works because we completely abandon data sharing and therefore do away with cache coherence protocols; instead, we propose to rely solely on remote accesses and execution migration to provide coherence and consistency.

Most previous works on thread migration do not explicitly consider deadlock caused by migrations. In many cases, those machines implicitly require a centralized migration scheduler or the operating system to ensure that no execution context is blocked at the migrating destination. The thread migration schemes that do consider deadlock provide deadlock recovery mechanisms instead of preventing deadlock [36]. The migration protocol and architecture we propose in this paper is deadlock-free at the protocol level without any centralized scheduler or OS aid, and, as such, enable fast, hardware-level thread migrations.

VI. Conclusion

In this paper, we have extended the family of directoryless NUCA architectures by adding efficient, hardware-level core-to-core thread migrations as a way to maintain sequential consistency and memory coherence in a large multicore with per-core caches. Taking advantage of locality in shared data accesses exhibited by many applications, migrations amortize the cost of remote accesses that limit traditional NUCA performance. At the same time, an execution migration design retains the cache utilization benefits of a shared cache distributed among many cores, and brings NUCA performance up to the level of directory-based cache-coherent designs.

We have demonstrated that appropriately designed execution migration (EM) and remote cache access (RA) hybrid designs do not cause deadlock. We have explored very straightforward hybrid EM/RA architectures in this paper; future work involves the development of better-performing migration predictors. Perhaps most promisingly, we have shown that the performance of EM designs is relatively easy to improve with low area cost, little power overhead, and virtually no verification cost, allowing chip designers to easily select the best compromise for their application space.

References

[1] S. Borkar, "Thousand core chips: a technology perspective," in *DAC*, 2007.

[2] N. Hardavellas, M. Ferdman, B. Falsafi *et al.*, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *ISCA*, 2009.

[3] I. T. R. for Semiconductors, "Assembly and packaging," 2007.

[4] S. Rusu, S. Tam, H. Muljono *et al.*, "A 45nm 8-core enterprise Xeon® processor," in *A-SSCC*, 2009.

[5] A. Gupta, W. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *International Conference on Parallel Processing*, 1990.

[6] C. Kim, D. Burger, and S. W. Keckler, "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," in *ASPLOS*, 2002.

[7] B. Verghese, S. Devine, A. Gupta *et al.*, "Operating system support for improving data locality on cc-numa compute servers," *SIGPLAN Not.*, vol. 31, no. 9, pp. 279–289, 1996.

[8] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-Level page allocation," in *MICRO*, 2006.

[9] M. Awasthi, K. Sudan, R. Balasubramonian *et al.*, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *HPCA*, 2009.

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

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

[12] K. Sudan, N. Chatterjee, D. Nellans *et al.*, "Micro-pages: increasing DRAM efficiency with locality-aware data placement," *SIGARCH Comput. Archit. News*, vol. 38, pp. 219–230, 2010.

[13] H. Garcia-Molina, R. Lipton, and J. Valdes, "A massive memory machine," *IEEE Trans. Comput.*, vol. C-33, pp. 391–399, 1984.

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

[15] M. Kandemir, F. Li, M. Irwin *et al.*, "A novel migration-based NUCA design for chip multiprocessors," in *SC*, 2008.

[16] K. Chakraborty, P. M. Wells, and G. S. Sohi, "Computation spreading: employing hardware migration to specialize CMP cores on-the-fly," in *ASPLOS*, 2006.

[17] K. K. Rangan, G. Wei, and D. Brooks, "Thread motion: fine-grained power management for multi-core systems," in *ISCA*, 2009.

[18] M. Marchetti, L. Kontothanassis, R. Bianchini *et al.*, "Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems," in *IPPS*, 1995.

[19] M. M. Bach, M. Charney, R. Cohn *et al.*, "Analyzing parallel programs with pin," *Computer*, vol. 43, pp. 34–41, 2010.

[20] J. E. Miller, H. Kasture, G. Kurian *et al.*, "Graphite: A distributed parallel simulator for multicores," in *HPCA*, 2010.

[21] S. Woo, M. Ohara, E. Torrie *et al.*, "The SPLASH-2 programs: characterization and methodological considerations," in *ISCA*, 1995.

[22] W. J. Dally and B. Towles, *Principles and practices of interconnection networks*. Morgan Kaufmann, 2003.

[23] A. Kumar, P. Kundu, A. P. Singh *et al.*, "A 4.6tbits/s 3.6ghz single-cycle noc router with a novel switch allocator," in *in 65nm CMOS, ICCD*, 2007.

[24] T. Konstantakopulos, J. Eastep, J. Psota *et al.*, "Energy scalability of on-chip interconnection networks in multicore architectures," *MIT-CSAIL-TR-2008-066*, 2008.

[25] S. C. Woo, J. P. Singh, and J. L. Hennessy, "The performance advantages of integrating block data transfer in cache-coherent multiprocessors," *SIGPLAN Not.*, vol. 29, pp. 219–229, 1994.

[26] S. Thoziyoor, J. H. Ahn, M. Monchiero *et al.*, "A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies," in *ISCA*, 2008.

[27] www.synopsys.com, "Synopsys design compiler."

[28] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. Morgan Kaufmann, September 2006.

[29] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Managing wire delay in large chip-multiprocessor caches," in *ISCA*, 2003.

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

[31] J. Huh, C. Kim, H. Shafi *et al.*, "A NUCA substrate for flexible CMP cache sharing," in *ICS*, 2005.

[32] S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek, "Reinventing scheduling for multicore systems," in *HotOS*, 2009.

[33] W. C. Hsieh, P. Wang, and W. E. Weihl, "Computation migration: enhancing locality for distributed-memory parallel systems," in *PPOPP*, 1993.

[34] G. M. Papadopoulos and D. E. Culler, "Monsoon: an explicit token-store architecture," in *ISCA*, 1990.

[35] M. D. Noakes, D. A. Wallach, and W. J. Dally, "The j-machine multicomputer: An architectural evaluation," in *ISCA*, 1993.

[36] S. Melvin, M. Nemirovsky, E. Musoll *et al.*, "A massively multithreaded packet processor," in *Workshop on Network Processors*, 2003.

Appendix

With a view of providing intuition for the real costs of cache coherence protocols, this appendix presents an analytical model comparing the MSI protocol (*CC* below) with an execution migration-only NUCA variant (*EM* below). For clarity, we analyze the relatively simple MSI protocol and show that, for the OCEAN_CONTIGUOUS benchmark, it results in memory accesses $1.5\times$ slower than the migration-only architecture; naturally, a suitably more complicated form of the same analysis applies to more complex protocols such as MOSI, but still shows them to be slower than even a basic, migration-only design.

$$AML_{CC} = cost_{\$access,CC} + rate_{\$miss,CC} \times cost_{\$miss,CC} \tag{1}$$

$$cost_{\$access} = cost_{L1} + rate_{L1\,miss} \times cost_{L2} \tag{2}$$

$$\begin{aligned} cost_{rdI,wrI,rdS} = {} & cost_{core\to dir} + cost_{dir\,lookup} + cost_{DRAM} \\ & + cost_{dir\to core} + cost_{\$insert} \end{aligned} \tag{3}$$

$$\begin{aligned} cost_{wrS} = {} & cost_{core\to dir} + cost_{dir\,lookup} + cost_{dir\to core} \\ & + cost_{\$invalidate} + cost_{core\to dir} + cost_{DRAM} \\ & + cost_{dir\to core} + cost_{\$insert} \end{aligned} \tag{4}$$

$$\begin{aligned} cost_{rdM} = {} & cost_{core\to dir} + cost_{dir\,lookup} + cost_{dir\to core} \\ & + cost_{\$flush} + cost_{core\to dir} + cost_{DRAM} \\ & + cost_{dir\to core} + cost_{\$insert} \end{aligned} \tag{5}$$

$$\begin{aligned} cost_{wrM} = {} & cost_{core\to dir} + cost_{dir\,lookup} + cost_{dir\to core} \\ & + cost_{\$flush} + cost_{core\to dir} \\ & + cost_{dir\to core} + cost_{\$insert} \end{aligned} \tag{6}$$

$$\begin{aligned} cost_{\$miss,CC} = {} & rate_{rdI,wrI,rdS} \times cost_{rdI,wrI,rdS} \\ & + rate_{wrS} \times cost_{wrS} + rate_{rdM} \times cost_{rdM} \\ & + rate_{wrM} \times cost_{wrM} \end{aligned} \tag{7}$$

$$\begin{aligned} cost_{DRAM,CC} = {} & cost_{DRAM\,latency} + cost_{DRAM\,serialization} \\ & + cost_{DRAM\,contention} \end{aligned} \tag{8}$$

$$cost_{message\,xfer} = cost_{\to,CC} + \left\lceil \frac{pkt\,size}{flit\,size} \right\rceil \tag{9}$$

$$cost_{\to,CC} = \#hops \times cost_{per\text{-}hop} + cost_{congestion,CC} \tag{10}$$

(a) MSI cache coherence protocol

$$\begin{aligned} AML_{EM} = {} & cost_{\$access,EM} + rate_{\$miss,EM} \times cost_{\$miss,EM} \\ & + rate_{core\,miss,EM} \times cost_{context\,xfer} \end{aligned} \tag{11}$$

$$cost_{\$access\,EM} = cost_{L1} + rate_{L1\,miss,EM} \times cost_{L2} \tag{12}$$

$$cost_{\$miss,EM} = cost_{core\to mem} + cost_{DRAM} + cost_{mem\to core} \tag{13}$$

$$\begin{aligned} cost_{DRAM,EM} = {} & cost_{DRAM\,latency} + cost_{DRAM\,serialization} \\ & + cost_{DRAM\,contention} \end{aligned} \tag{14}$$

$$cost_{message\,xfer} = cost_{\to,EM} + \left\lceil \frac{pkt\,size}{flit\,size} \right\rceil + cost_{Pipeline\,insertion} \tag{15}$$

$$cost_{\to,EM} = \#hops \times cost_{per\text{-}hop} + cost_{congestion,EM} \tag{16}$$

(b) EM

Fig. 9. Average memory latency (AML) costs for our MSI cache coherence protocol and for EM. The significantly less complicated description for EM suggests that EM is easier to reason about and implement. The description for a protocol such as MOSI or MOESI would be significantly bigger than for MSI.

A. Interconnect traversal costs.

Both protocols incur the cost of on-chip interconnect transmissions to retrieve data from memory, migrate thread contexts (in EM), and communicate among the caches (in CC). In the interconnect network model we assume a 16×16 mesh with two-cycle-per-hop 128-bit flit pipelined routers, an average distance of 12 hops with network congestion overheads consistent with what we observed for OCEAN_CONTIGUOUS, making the network transit cost

$$\begin{aligned} cost_{\to,CC} &= 24 + 12 = 36, \text{ and} \\ cost_{\to,EM} &= 24 + 12 = 36. \end{aligned} \tag{17}$$

Delivering a message adds a load/unload latency dependent on the packet size: for example, transmitting the 1.5 Kbit EM context requires

$$cost_{context\,xfer} = 36 + \frac{1536 \text{ bits}}{128 \text{ bits}} + 3 = 51. \tag{18}$$

By the same token, in both CC and EM, transmitting a single-flit request takes 37 cycles ($cost_{core \to dir}$ and $cost_{core \to mem}$) and transferring a 64-byte cache line needs 40 cycles ($cost_{dir \to core}$ and $cost_{mem \to core}$).

Parameter	CC	EM
$cost_{L1}$	2 cycles	2 cycles
$cost_{L2}$ *(in addition to L1)*	5 cycles	5 cycles
$cost_{\$invalidate}$, $cost_{\$flush}$	7 cycles	—
cache line size	64 bytes	64 bytes
average network distance	12 hops	12 hops
$cost_{per\text{-}hop}$	2 cycles	2 cycles
$cost_{congestion}$	12 cycles	12 cycles
$cost_{DRAM\,latency}$	235 cycles	235 cycles
$cost_{DRAM\,serialization}$ (1 cache line)	50 cycles	50 cycles
$cost_{dir\,lookup}$	10 cycles	—
flit size	128 bits	128 bits
execution context (32-bit x86)	—	1.5 Kbit [17]
$rate_{L1\,miss}$ / $rate_{\$miss}$ (both L1 and L2 miss)	5.8% / 4.8%	2.4% / 0.8%
$rate_{core\,miss,EM}$	—	21%
$rate_{rdI}$, $rate_{rdS}$, $rate_{rdM}$, $rate_{wrI}$, $rate_{wrS}$, $rate_{wrM}$	31.5%, 21.4%, 12%, 22.4%, 12.6%, 0.1%	—

TABLE IV
VARIOUS PARAMETER SETTINGS FOR THE ANALYTICAL COST MODEL FOR THE OCEAN_CONTIGUOUS BENCHMARK

B. Off-chip DRAM access costs.

In addition to the DRAM latency itself, off-chip accesses may experience a queueing delay due to contention for the DRAM itself; moreover, retrieving a 64-byte cache line must be serialized over many cycles (Equations 8 and 14). Because there were dramatically fewer cache misses under EM, we observed relatively little DRAM queue contention (11 cycle), whereas the higher off-chip access rate of CC resulted in significantly more contention on average (43 cycles):

$$\begin{aligned} cost_{DRAM,EM} &= 235 + 50 + 11 = 299 \\ cost_{DRAM,CC} &= 235 + 50 + 43 = 331. \end{aligned} \tag{19}$$

C. EM memory access latency.

Given the network and DRAM costs, it's straightforward to compute the average memory latency (AML) for EM which depends on the cache access cost and, for every cache miss, the cost of accessing off-chip RAM; under EM, we must also add the cost of migrations caused by core misses (Equation 11).

The cache access cost is incurred for every memory request and depends on how many accesses hit the L1 cache: for EM,

$$cost_{\$access\,EM} = 2 + 2.4\% \times 5 = 2.12. \tag{20}$$

Each miss under EM contacts the memory controller, retrieves a cache line from DRAM, and sends it to the requesting core:

$$cost_{\$miss,EM} = 37 + 299 + 40 = 376. \tag{21}$$

Finally, then, we arrive at the average memory latency:

$$AML_{EM} = 2.12 + 0.8\% \times 376 + 21\% \times 51 = 15.8. \tag{22}$$

D. Cache coherence memory access latency.

Since CC does not need to migrate execution contexts, memory latency depends on the cache access and miss costs (Equation 1). Because fewer accesses hit the L1 cache, even the cache access cost itself is higher than under EM:

$$cost_{\$access} = 2 + 5.8\% \times 5 = 2.29. \tag{23}$$

The cost of a cache miss is much more complex, as it depends on the kind of access (read or write, respectively *rd* and *wr* below) and whether the line is cached nowhere (*I* below) or cached at some other node in shared (*S*) or modified (*M*) state:

- Non-invalidating requests (75.3% of L2 misses for OCEAN_CONTIGUOUS)—loads and stores with no other sharers, as well as loads when there are other read-only sharers—contact the directory and retrieve the cache line from DRAM:

$$cost_{rdI,wrI,rdS} = 37 + 10 + 331 + 40 + 7 = 425. \tag{24}$$

- Stores to data cached in read-only state elsewhere (12.6%) must invalidate the remote copy before retrieving the data from DRAM: in the best case of only one remote sharer,

$$cost_{wrS} = 37 + 10 + 37 + 7 + 37 + 331 + 40 + 7 = 506. \tag{25}$$

- Loads of data cached in modified state elsewhere (11.9%) must flush the modified remote cache line and write it back to DRAM before sending the data to the requesting core via a cache-to-cache transfer:

$$cost_{rdM} = 37 + 10 + 37 + 7 + 40 + 310 + 40 + 7 = 488. \tag{26}$$

- Stores to data cached in modified state elsewhere (0.1%) must also flush the cache line but avoids a write-back to DRAM by sending the data to the requesting core via a cache-to-cache transfer:

$$cost_{wrM} = 37 + 10 + 37 + 7 + 40 + 40 + 7 = 178. \tag{27}$$

Combining the cases with their respective observed rates (cf. Table IV), we arrive at the mean cost of a cache miss for CC:

$$cost_{\$miss,CC} = (31.5\% + 22.4\% + 21.4\%) \times 425 + 12.6\% \times 506 + 11.9\% \times 488 + 0.1\% \times 178 = 442, \tag{28}$$

and, finally, the average memory cost for CC:

$$AML_{CC} = 2.29 + 4.8\% \times 442 = 23.5, \tag{29}$$

over **1.5**× greater than under EM.