Directoryless Shared Memory Architecture using Thread Migration and Remote Access

Keun Sup Shim*, Mieszko Lis*, Omer Khan‡ and Srinivas Devadas*

*Massachusetts Institute of Technology, Cambridge, MA

‡University of Connecticut, Storrs, CT

Abstract

Distributed directory cache coherence protocols for current many-core CMPs are not only difficult and error-prone to implement and verify, but also provide suboptimal performance when a thread requires access to large amounts of data distributed across the chip: the data must be brought to the core where the thread is running, incurring delays and energy costs. In this paper, we propose an approach based on the combination of partial-context thread migration and a directory-free remote access protocol: for these kinds of applications, our architecture can outperform directory-based cache coherence. In addition, unlike with distributed cache coherence protocols, the verification complexity of our architecture does not grow with the number of cores.

Keywords

Parallel Architectures, Distributed Caches, Shared Memory, Data Locality

I. Introduction

As transistor density has continued to grow, Chip Multiprocessors (CMPs) have become common; for scalability reasons, large-scale CMPs (≥ 16 cores) tend towards a tiled architecture where arrays of replicated tiles are connected over an on-chip interconnect. Each tile contains a processor with its own L1 cache and a slice of the L2 cache: to maximize effective on-chip cache capacity, physically distributed L2 cache slices form one large logically shared cache, known as Non-Uniform Cache Access (NUCA) architecture [9]. Under this *Shared-L2* organization, 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 at the L2 level. At the L1 level, data can be replicated across any requesting core since current CMPs use *Private-L1* caches. To provide a unified shared-memory abstraction, many-core systems commonly maintain *cache coherence* of such private caches using a coherence protocol and distributed directories.

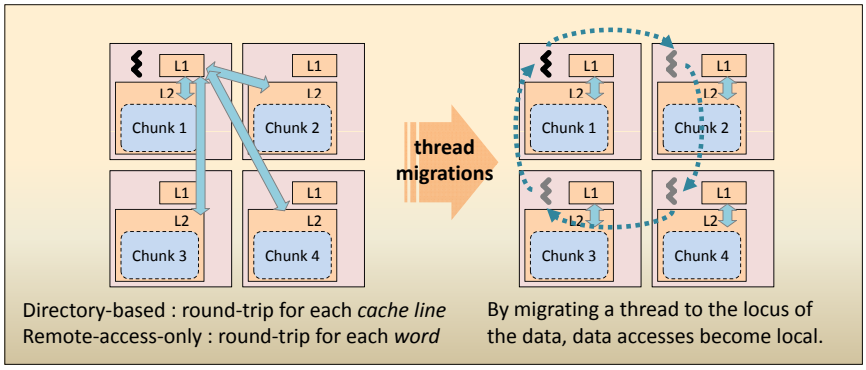

Fig. 1. When applications exhibit data access locality, efficient thread migration can turn many round-trips to retrieve data into a series of migrations followed by long stretches of accesses to locally cached data.

In such a conventional tiled architecture, data are typically distributed across multiple shared cache slices to minimize expensive off-chip accesses, especially when large data structures that do not fit in a single cache are shared by multiple threads or iteratively accessed even by a single thread. This raises the need for a thread to access data mapped to remote caches, often with high spatio-temporal locality, and results in large amounts of on-chip network traffic. For example, a database request might result in a series of phases, each consisting of many accesses to contiguous stretches of data. Each request will typically run in a separate thread, pinned to a single core throughout its execution; because this thread might access data cached in remote last-level cache slices, however, the data must be brought to the core where the thread is running. As shown in Figure 1, in a directory-based architecture, the data would be brought to the core's private cache, only to be replaced when the next phase of the request accesses a different segment of data. An alternative we leverage here is to move the thread itself to follow the data instead of transferring the data itself: if the thread context is small compared to the data that would otherwise be transferred, moving the thread can significantly reduce on-chip interconnect traffic.

Another barrier to distributed directory coherence protocols is that they are extremely difficult to implement and verify. The design of even a simple coherence protocol is not trivial; under a coherence protocol, the response to a given request is determined by the state of *all* actors in the system, transient states due to indirections (e.g., cache line invalidation), and transient states due to the nondeterminism inherent in the relative timing of events. Since the state space explodes exponentially as the distributed directories and the number of cores grow, it is virtually impossible to cover all scenarios during verification either by simulation or by formal methods [17]. Unfortunately, verifying small subsystems does not guarantee the correctness of the entire system [3]. In modern CMPs, errors in cache coherence are one of the leading bug sources in the post-silicon debugging phase [6].

A straightforward approach to removing directories while maintaining cache coherence is to disallow cache line replication across on-chip caches (even L1 caches) and use remote word-level access to load and store remotely cached data [7]: in this scheme, every access to an address cached on a remote core becomes a two-message round trip. Since only one copy is ever cached, however, coherence is trivially ensured. While such a remote-access-only architecture is still susceptible to data access patterns shown in Figure 1 in terms of performance and network traffic, data locality under a directory-free architecture can be better exploited by using *fine-grained hardware-level thread migration* to complement remote accesses [5, 10]. In this approach, accesses to data cached at a remote core can also cause the thread to migrate to that core and continue execution there. When several consecutive accesses are made to data at the same core, thread migration allows those accesses to become local, potentially improving performance over a remote-access regimen. Migration costs, however, make it crucial to migrate only when *multiple* remote accesses would be replaced to make the cost "worth it." Moreover, since only a few registers are typically used between the time the thread migrates out and returns, transfer costs can be reduced by not migrating the unused registers.

In this paper, we propose a novel migration prediction scheme that addresses these questions: it decides at instruction granularity whether to perform a remote access or a thread migration, and which part of the thread context (i.e., which registers) to migrate in each migration to further reduce the migration cost. Our results show that the hybrid architecture with our predictor improves performance and significantly reduces network traffic compared to a remote-access-only architecture. We also compare with a state-of-the-art directory-based architecture and show that our hybrid design can perform better for a certain class of applications, and is competitive overall. Since it requires no directories or coherence protocols, we argue that our architecture provides an interesting design point on the hardware coherence spectrum.

II. Architecture Overview

In what follows, we describe the remote access protocol, as well as a protocol based on *hardware-level thread migration*. We then present a framework that combines both.

A. Remote Cache Access

Under the remote-access framework of standard NUCA designs [7, 9], all non-local memory accesses cause a request to be transmitted over the interconnect, 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 first find the *home* core H for A (e.g., by consulting a mapping table or masking some address bits). If $H = C$ (a *core hit*), the request is served

locally at C. If $H \neq C$ (a *core miss*), on the other hand, a remote access request needs to be forwarded to core H, which will send a response back to C upon its completion. Note that, unlike a private cache organization where a coherence protocol takes advantage of spatial and temporal locality by making a copy of the block containing the data in the local cache, this protocol incurs round-trip costs *for every remote word access*.

B. Thread Migration

Fine-grained, hardware-level thread migration has been proposed to exploit data locality for NUCA architectures [10]. This mechanism brings the *thread* to the data instead of the other way around. When a core C running thread T executes a memory access for address A, it must first find the *home* core H for A. If $H = C$ (a *core hit*), the request is served locally at C. If $H \neq C$ (a *core miss*), the hardware interrupts the execution of the thread on C, packs the thread's execution context (microarchitectural state) into a network packet, and sends it to H via the on-chip interconnect where the packet is loaded to the context and an execution of T is resumed (cf. Figure 2.C). This provides faster migrations than other approaches that require OS intervention or memory accesses since it migrates threads directly over the interconnect. A register mask is used to allow partial context migration (i.e., selective loading/unloading of the register file).

If another thread is already executing at the destination core, it must be evicted and moved to a core where it can continue running. To reduce the need for evictions, cores duplicate the architectural context (register file, etc.) and allow a core to multiplex execution among two concurrent threads. To prevent deadlock, one context is marked as the *native context* and the other as the *guest context*: a core's native context may only hold the thread that started execution there (called the thread's *native core*), and evicted threads must return to their native cores to ensure deadlock freedom [5].

C. Performance Overhead of Thread Migration

The migration overhead is directly affected by the context size. The relevant architectural state that must be migrated in a 64-bit x86 processor amounts to about 3.1Kbits (sixteen 64-bit general-purpose registers, sixteen 128-bit floating-point registers and special purpose registers) [2], which is what we use in this paper. This introduces a *serialization* latency since the full context needs to be loaded (unloaded) into (from) the network: with 128-bit flit network and 3.1Kbits context size, the thread context consists of $\left\lceil \frac{pkt\,size}{flit\,size} \right\rceil$ = 26 flits, incurring the serialization overhead of 26 cycles. The context size will vary depending on the architecture; in the TILEPro64 [14], for example, it amounts to about 2.2Kbits (64 32-bit registers and a few special registers).

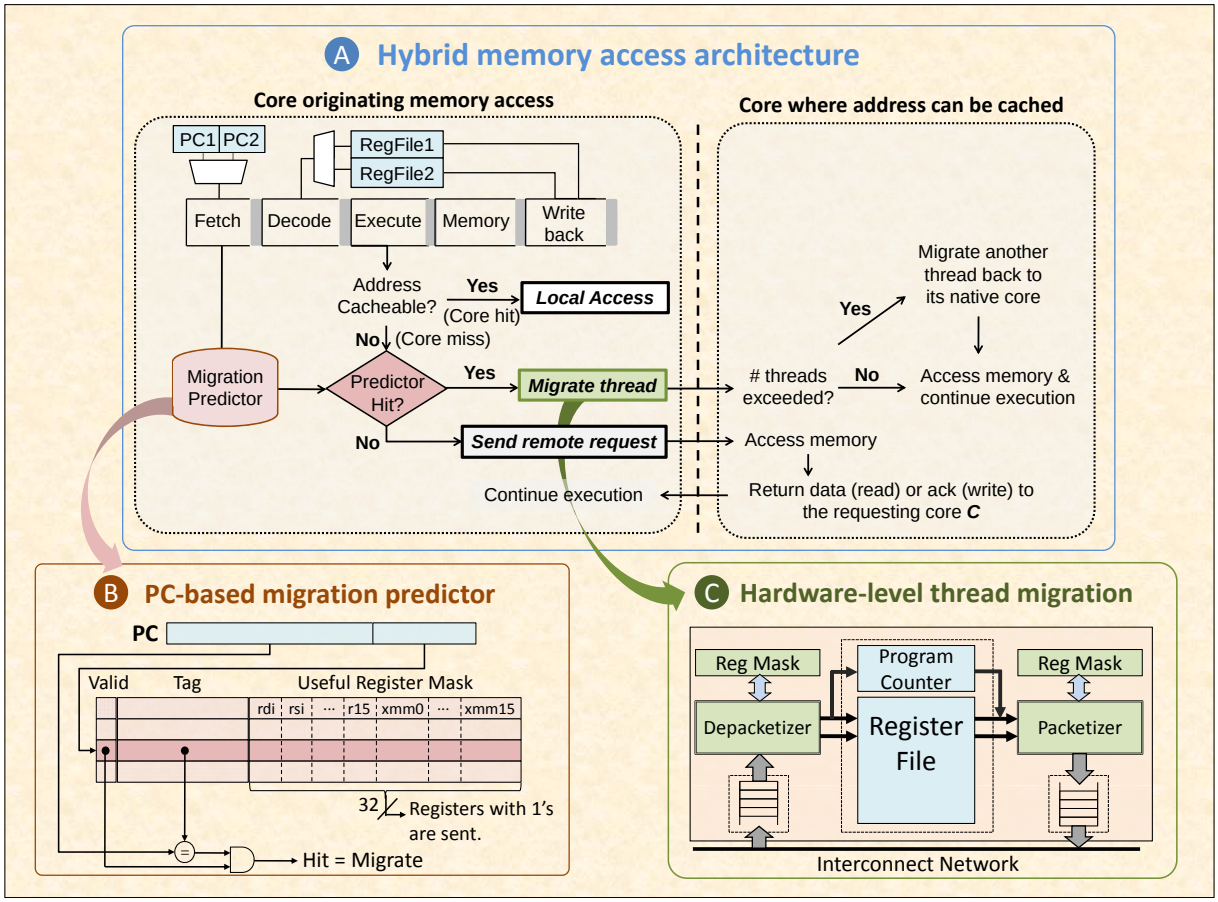

Fig. 2. Hybrid memory access architecture that supports thread migration and remote access. Each core has a PC-based migration predictor, where each entry contains a $\{PC,\ register\ mask\}$ pair.

Another overhead is the *pipeline insertion* latency. Since a memory address is computed in the middle of the pipeline, if a thread ends up migrating to another core and re-executes, it needs to *refill* the pipeline. We assume an overhead of ten cycles to refill the pipeline upon migrations.

D. Hybrid Memory Access Framework

Figure 2.A illustrates a hybrid directoryless architecture: each access to data cached on a remote core may either perform the access via a remote access or migrate the current execution thread. Our migration predictor makes this decision on a per-instruction granularity. It is worthwhile to mention that we allow replication for instructions since they are read-only; threads need not perform a remote access nor migrate to fetch instructions.

III. Thread Migration Predictor

With a large thread context size, thread migration costs exceed the cost required by remote-access-only designs on a per-access basis. On the other hand, multiple *contiguous* memory accesses to the same core

make migration beneficial because after migrating on the first access, the remaining accesses become local. Therefore, our migration predictor focuses on detecting those. Compared to the predictor presented in [15], which only supports full-context migration, we further reduce migration costs by sending only a part of the register file when a thread migrates (usually, only some of the registers are used between the time the thread migrates out of its native core and the time it returns). With the deadlock-free migration framework of [5], the native-core register file remains intact even if a thread migrates away, because its context it is not used by any other *guest threads*. This allows us to carry only the registers that will be read "during the trip" and bring back only the registers written while away.

Since the migration/remote-access decision and the partial context prediction must be made on every memory access, they must be implementable as efficient hardware. To this end, we present a per-core *migration predictor*—a PC-indexed direct-mapped data structure shown in Figure 2.B. Our predictor is based on the observation that sequences of consecutive memory accesses to the same home core and register usage patterns within those sequences are highly correlated with the program (instruction) flow, and, moreover, that these patterns are fairly consistent and repetitive across program execution. Our baseline configuration uses a 128-entry predictor, each of which consists of a 64-*bit PC* and a 32-*bit useful register mask*, about 1.5KB in total. An N-bit mask is required for an architecture with N general registers: each bit indicates whether the corresponding register is sent during migrations.

If the home core for an address is not where the thread is currently running (a *core miss*), the predictor decides between a remote access and a migration: if the PC hits in the predictor, it instructs a thread to migrate; otherwise, a remote access is performed. If the thread is migrating from its *native* core to another core, it transfers only those registers whose bits in the register mask are set.

In the next section, we first describe how instructions are detected as "migratory" and inserted into the migration predictor; then, we extend the predictor to track used-registers information for migratory instructions.

A. Detecting Migratory Instructions: WHEN to migrate

The detection of *migratory instructions* is done by tracking how many consecutive accesses to the same core have been made. If this count exceeds a threshold, we insert the PC into the predictor; otherwise, we classify the instruction as remote-access (the default state). Each thread tracks (1) *Home*, which maintains the home core ID for the most recently requested memory address, (2) *Depth*, which indicates how many times thus far a thread has contiguously accessed the recent home location (i.e., the *Home* field), and (3) *Start PC*, which tracks the PC of the very first instruction among memory sequences that accessed the

home location in the *Home* field. We separately define the depth threshold θ, which indicates the depth at which we determine the instruction as migratory.

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

1) if $Home = H$ (i.e., memory access to the same home core as that of the previous memory access),
    a) if $Depth < \theta$, increment $Depth$ by one;
2) if $Home \neq H$ (i.e., a new sequence starts with a new home core),
    a) if $Depth = \theta$, $StartPC$ is considered a migratory instruction and thus inserted into the predictor;
    b) if $Depth < \theta$, $StartPC$ is considered a remote-access instruction;
    c) reset the entry (i.e., $Home \leftarrow H$, $StartPC \leftarrow P$, $Depth \leftarrow 1$).

B. Detection of Useful Registers: WHAT to migrate

We now extend our migration predictor to support partial context migrations. In addition to (1) *Home*, (2) *Depth*, and (3) *Start PC*, each thread now also tracks (4) *Used Registers*, a 32-bit vector where each bit indicates whether the corresponding register has been used or not within a sequence of memory instructions accessing the same home core. Every instruction (both memory and non-memory) updates this *Used Registers* field by setting the bit when the corresponding register is being read or written. When the PC is detected as a migratory instruction and inserted into the predictor, the *Used Registers* field is inserted together with *Start PC* (see Figure 2.B).

Figure 3.A shows an example of the detection mechanism when $\theta = 2$. Suppose a thread executes a sequence of instructions, $I_1 \sim I_5$. I_1, I_3 and I_5 are memory instructions, I_2 and I_4 are non-memory instructions, and r_n denotes the nth register. When I_1 is first executed, the entry {Home, Depth, Start PC, Used Registers} will hold the value of {C, 1, PC_1, r1}. Then, when I_2, a non-memory instruction using r2 and r3, is executed, the *Used Registers* bit-vector is updated to set the bits for r2 and r3. When I_3 is executed, it accesses the same home core C and thus the *Depth* field is incremented by one. I_4 simply adds r4 to the register bit-vector. Lastly, when I_5 (which starts a new memory sequence for the home core A) is executed, since the *Depth* to core C has reached the threshold, PC_1 in the *Start PC* field, which represents the first memory instruction (I_1) that accessed the home core C, is classified as a migratory instruction and thus is added to the predictor with the register mask bits.

The partial context migration policy is as follows: when a thread T executes a memory instruction whose PC *hits* in the migration predictor and thus needs to migrate,

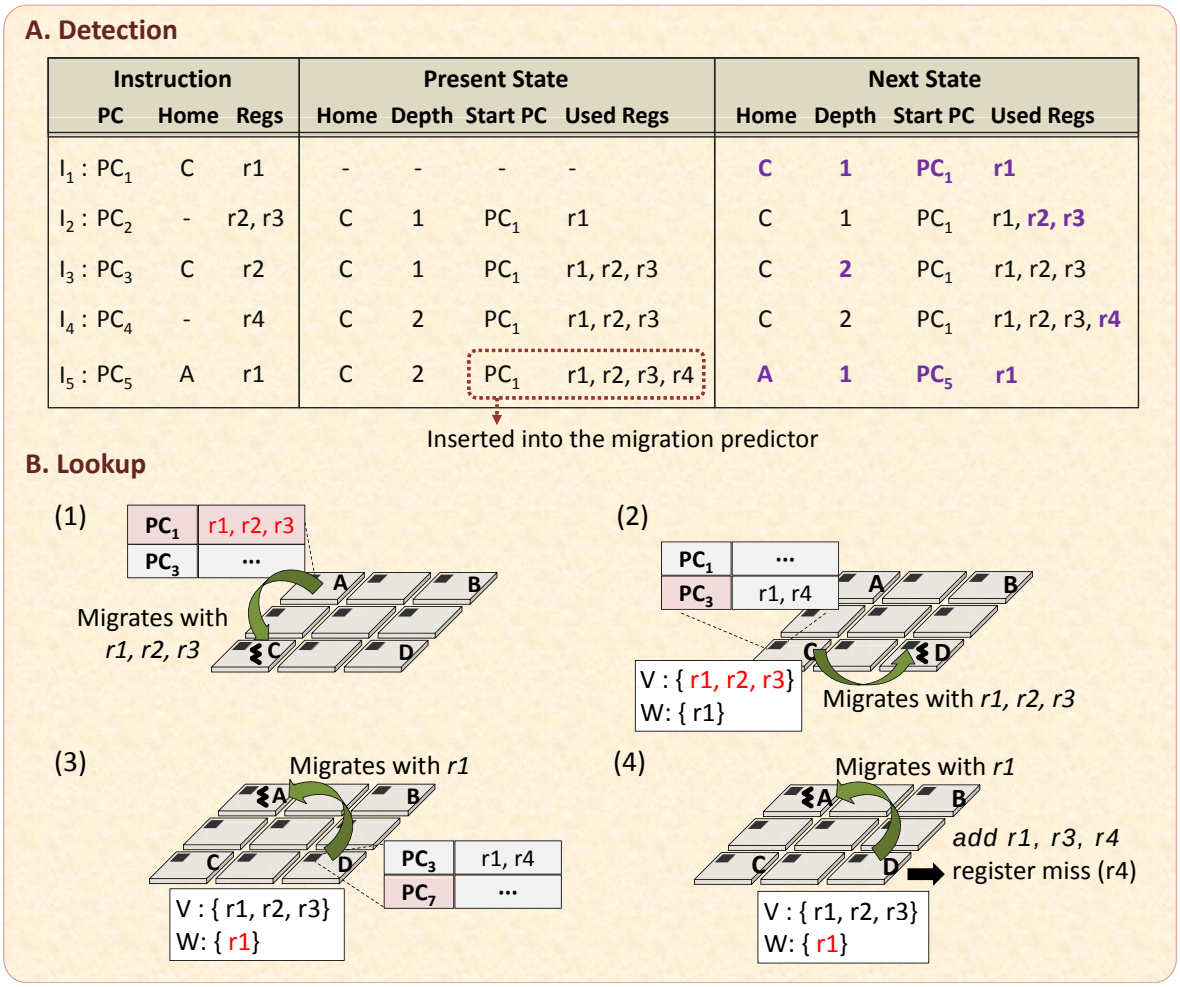

Fig. 3. (A) An example of how a specific instruction (or PC) is detected as a *migratory instruction* and is inserted into the migration predictor with the information of used-registers. (B) An example of a partial context migration.

1) if T is migrating from its *native* core to a *non-native* core, it takes the registers specified in the *Useful Register Mask* of the migration predictor (cf. Figure 3.B.1);
2) if T is migrating from a *non-native* core to another *non-native* core, it takes all the registers that T brought when T first migrated out from its native core (cf. Figure 3.B.2);
3) if T is migrating back to its *native* core from a *non-native* core, it takes only the registers that are *written* while T was outside from its native core (cf. Figure 3.B.3).
4) Special purpose registers required for the thread execution (e.g., *rip*, *rflags* and *mxcsr* for a 64-bit x86 architecture) are always transferred.

In order to implement these policies, a thread carries around two 32-bit masks: *V-mask* and *W-mask*. *V-mask* identifies the registers that the thread may access while outside of its native core (looked up in

the predictor when the thread first migrated out from its native core). *W-mask* keeps track of the registers that have been *written* while outside the native core, and is used to implement policy (3). Since a register file remains intact in the native context, a thread returning to its native core needs to carry only the registers that have been modified. During migrations, these two masks and {Home, Depth, Start PC, Used Registers} must be transferred together with the context. With 64 cores (6 bits for the home core ID), a maximum depth threshold of 8 (3 bits), a 64-bit Start PC and a 32-bit used-register mask, a total of 169 bits have to be transferred in addition to the context.

Unlike the decision on whether to perform a remote access or a thread migration, the useful register information in the predictor is only consulted by a thread when at its native core; this is because the native context is the only place where all the register values are maintained for the thread, and once it leaves the native core, the thread cannot use any registers other than the ones it initially brought from its native core (i.e., registers in *V-mask*). If a thread encounters an instruction which requires the read or write of a register r_n which has not been brought from its native core while outside its native core (i.e., $r_n \notin$ *V-mask*, called a *register miss*), the thread stops its execution and returns to its native core (cf. Figure 3.B.4).

Our migration predictor tries to minimize such *register-miss* migrations; we update the useful register mask in the predictor by adding the register that caused the register miss when the thread migrates back. With this learning mechanism, the useful register mask for a particular PC, PC_1, will eventually converge to a superset of registers that are used after the thread migrates at PC_1 until it migrates back to its native core.

IV. Methodology

A. Simulation Framework

We use Graphite [13] to model the hybrid architecture that supports both remote-access and thread migration. We simulate 64 in-order, single-issue cores with 2-way fine-grain multithreading in an 8×8 mesh (XY routing, 128-bit flits); each core has a 32 KB L1 data cache (2-way), 32 KB L1 read-only instruction cache (4-way) and a 128 KB, 4-way L2 cache (64B cache block). A 2-cycle fixed per-hop latency with extra delays due to contention is modeled. For data placement, we use the *first-touch after initialization* policy which allocates each page (4 KB) to the core that first accesses it after parallel processing has started.

B. Evaluated Systems

We compare our hybrid directoryless architecture with migration predictor (*NoDirPred*) against the remote-access-only directoryless baseline (*NoDirRA*). To see how well the predictor itself works, we also compare with a simple DISTANCE decision scheme (*NoDirDist*) previously proposed by [10]: the intuition here is that over short distances the round-trip remote-access overhead is low, so threads migrate only if the distance to the home core exceeds some threshold d. We use $d = 6$, the average hop count for an 8×8 mesh, and transfer the full context during migrations. We also present the result for a directory-based cache-coherence architecture (*DirCC*) to provide a sense of how directoryless designs perform compared to conventional designs. *DirCC* uses MSI with distributed full-map directories in a Private-L1 Shared-L2 configuration, and Reactive-NUCA [8] data placement. Since all schemes use the shared-L2 configuration and our benchmark data sets fit on chip, off-chip access rate differences are negligible across all the systems we evaluate; the main difference stems from the performance of on-chip cache accesses.

C. Application Benchmarks

Our benchmarks include a parallel perceptron cross-validation (*prcn+cv*), a distributed hash table benchmark (*dht*) and a set of *Splash-2* [16] benchmarks (cross-validation is a popular machine learning technique for optimizing model accuracy). Each application ran to completion, and we measured the parallel completion time. Migration overheads (cf. Section II-C) for our hybrid architecture are taken into account.

V. EVALUATION

A. Performance and network traffic

Overall performance of *DirCC*, *NoDirRA*, *NoDirDist*, and *NoDirPred* is compared in Figure 4.(a); for *NoDirPred*, the depth threshold θ is set to 3. When compared to *DirCC*, *NoDirRA* performs worse by 49% on average, while our hybrid architecture (*NoDirPred*) performs worse by 13% on average. *NoDirDist* performs the worst, indicating that migration decisions must be made judiciously. Since I-cache content is not transferred during migrations, *NoDirPred* shows 7% more I-cache misses than *NoDirRA* on average; I-cache miss rates, however, are still very low (mostly $< 0.1\%$) and have negligible effect on performance. We also compare on-chip network traffic in each system, measured as the number of flits sent times the number of hops traveled. Figure 4.(b) shows that *NoDirPred* reduces network traffic by 28% on average

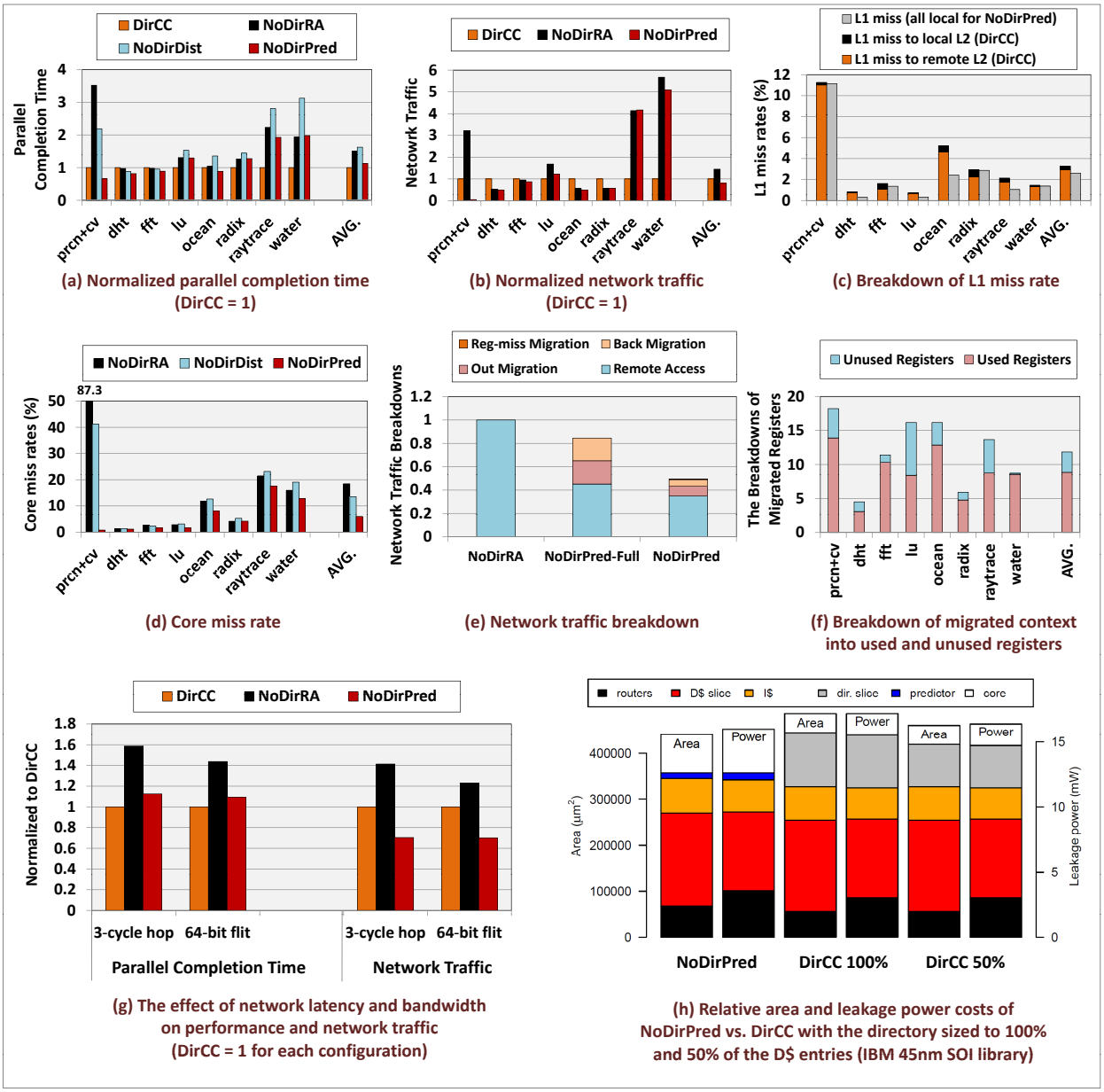

Fig. 4. Evaluation of our proposed architecture

compared to *DirCC*, and by 50% when compared to *NoDirRA*; while not shown in the figure, the network traffic for *NoDirDist* is prohibitive, 5× more traffic on average compared to *DirCC*.

Although the average performance of *NoDirPred* is less than that of *DirCC*, it is important to note that most of our benchmarks were originally developed with directory coherence in mind. Parallel cross-validation with the perceptron learning algorithm (*prcn+cv*) is an example where directory-based coherence does not work well; the computation requires each thread to traverse through a dataset spread

across the cores, resulting in many accesses to remote caches and high network overhead for *DirCC*. As a result, *NoDirPred* outperforms *DirCC* by 34% with 42× less traffic for *prcn+cv*, demonstrating that such overhead can be eliminated by migrating threads to the data.

To better understand the overall performance, we measured L1 cache miss rates for *DirCC* and *NoDirPred*; the results are shown in Figure 4.(c). Since cache lines are not replicated across L1 caches in the directoryless design (*NoDirPred*), the effective L1 cache capacity increases, always resulting in lower L1 miss rates than *DirCC*; more importantly, while all L1 misses under *NoDirPred* are forwarded to local L2 caches, a large fraction of L1 misses for *DirCC* result in memory requests to remote L2 caches, a major factor in performance degradation and network traffic for directory-based architecture.

On the other hand, directoryless designs can suffer when the core miss rate is high, i.e., when frequently accessing data cached in remote cores; the core miss rate of *DirCC* is always *zero*. Figure 4.(d) shows that on average, 18.4% of total memory accesses result in *core misses* for *NoDirRA*, which drops to only 6% for *NoDirPred*. While not shown, this improvement is achieved with the average migration rate of 1%, indicating that the predictor works well. *Raytrace* and *water* are examples where *NoDirPred* suffers in terms of both performance and network traffic due to high core miss rates.

In order to track how traffic is reduced by *partial* context migration, we compare our design with the *full* context migration variant, which always sends the full thread context during migrations (*NoDirPred-Full*). The results are shown in Figure 4.(e); *NoDirPred* reduces *out migration traffic* (migrations to non-native cores) by 58% and *back migration traffic* (migrations back to native cores) by 72% compared to *NoDirPred-Full*. The reduction in *out migration traffic* is achieved by our predictor (the useful register field) and the reduction in *back migration traffic* is achieved by the *W-mask*, which keeps track of the written registers. While using partial context migration occasionally induces unnecessary migrations due to *register misses*, we observe almost no overhead from this because our predictor learns from each miss by adding the missing register to the useful register mask for the appropriate PC. With this *union* mechanism, however, the register mask will only grow and never shrink back; this makes our context prediction conservative and thus, some of the registers that are migrated may not be actually used. Across all benchmarks, around 75% of migrated registers are actually used on average (see Figure 4.(f)), showing that our predictor is reasonably efficient.

We further demonstrate that the relative performance and network traffic of our hybrid architecture (*NoDirPred*) are maintained over different network parameters. Figure 4.(g) shows that *NoDirPred* outperforms *NoDirRA* by 29% with 3-cycle per-hop latency (originally, 24% with 2-cycle per-hop latency); this is because the round-trip nature of remote accesses suffers more from increased per-hop latency. With

a 64-bit flit network instead of 128-bit, on the other hand, the network traffic reduction rate of *NoDirPred* over *NoDirRA* decreases from 50% to 43%; this is because a large fraction of remote access messages (i.e., those that do not carry a data word) fit into 64 bits, and do not need additional flits to make up for the halved bandwidth. Performance improvements also drop slightly, but not significantly.

B. Area costs

Our hybrid architecture requires an extra architectural context (for the guest thread) and a learning migration predictor. It also requires four independent on-chip networks for migrations and evictions as well as remote access requests and responses, while a deadlock-free implementation of directory-based coherence requires three networks (for coherence requests, replies, and invalidations).

To give an idea of how these costs compare against that of a directory-based architecture, we estimated the area required for a MESI implementation with a full-map directory sized to 100% and 50% of the total L1 data cache entries, and compared the area and leakage power to that of *NoDirPred*. We assumed 8 KB L1-I cache and 32 KB L1-D cache per core and did not include the (identical) L2 caches in either *NoDirPred* or *DirCC* area results. Area and power estimates were obtained by synthesizing RTL using Synopsys Design Compiler with a 45nm ARM ASIC library and IBM SRAM blocks; synthesis targeted an 800MHz clock.

Figure 4.(h) shows how the silicon area and leakage power compare. Not surprisingly, SRAM blocks (instruction and data caches, as well as the directory for *DirCC*) were responsible for most of the area in all variants. Overall, the extra thread context and extra router present in *NoDirPred* were outweighed by the area required for the directory in both the 50% and 100% versions of MESI, indicating that *NoDirPred* can reduce area costs.

C. Verification Complexity

Distributed cache coherence protocols are notoriously complex and difficult to design, as well as hard to verify due to the state space explosion as the number of cores grows. Even relatively simple protocols (e.g., MSI, MESI) introduce many transient states that are not explicit in the higher-level protocol [3], and writing testbenches that exercise all the reachable transient states is an arduous task. Significant modeling simplifications must be made to make exploring the state space tractable [1], and even formally verifying a given protocol on a few cores gives no confidence that it will work on 100.

While design and verification complexity is difficult to quantify and compare, directory-free designs (*NoDirRA* and *NoDirPred*) have a significant advantage over directory-based designs. Since a given memory address may only be cached in a single place, a memory request will depend only on the

validity of a given line in a single cache, and no indirections or transient states are required. Unlike caches and directory entries in DirCC, directoryless caches keep no information about more than the local core; the VALID and DIRTY flags that together determine the state of a given cache line are local to the tile. Further, the thread migration framework does not introduce additional complications, since the data cache does not care whether local memory requests come from native or migrated threads: the same local data cache access interface is used. All of the logic required for the migration framework—deciding whether to migrate, computing the destination core, serializing and deserializing network packets from/to the execution context, evicting a running thread if necessary, etc.—is also local to the tile.

As a result, the overall correctness can be cleanly separated into (a) the remote access framework, (b) the thread migration framework, (c) the cache that serves the memory request, and (d) the underlying on-chip interconnect, all of which can be reasoned about separately. Furthermore, the entire state space can be exercised in the 4-tile system, meaning that the system could be scaled to an arbitrary number of cores without incurring an additional verification burden.

VI. Related Work

Recent many-core CMPs have organized physically distributed L2 cache slices to form one logically shared L2 cache, leading to a Non-Uniform Cache Access (NUCA) architecture [8,9]. Our baseline also uses shared L2 cache; the main difference between previously-proposed NUCA designs and our baseline architecture is that while they still rely on private L1 caches and allow replication at the L1, which requires directories and a coherence protocol to maintain coherence, our baseline allows no replication across L1 caches, completely obviating the need for directories and a coherence protocol. Our work focuses on proposing a *directoryless* shared-memory architecture that outperforms the remote-access-only baseline by complementing remote accesses with judicious thread migrations.

Migrating computation to the locus of the data is not itself a novel idea. Michaud showed that execution migration can improve the overall on-chip cache capacity and selectively migrated sequential programs to improve cache performance [12]. Computation spreading [4] splits thread code into segments and assigns cores to different segments, migrating execution to improve code locality. Thread migration has also been used to provide memory coherence among per-core caches [10] using a deadlock-free fine-grained thread migration protocol [5]; we adopt the same protocol for our hybrid framework. Although a migration predictor that decides between migrations and remote accesses is introduced in [15], it does not address the overhead of high network traffic for thread migration. This paper proposes a novel migration predictor that supports partial context migration, improving both performance and network traffic. As a

silicon prototype, we have built a 110-core CMP in a 45nm ASIC which supports hardware-level thread migration on a stack-based core architecture; its physical implementation details and evaluation results are presented in [11].

VII. Conclusions

We have demonstrated that, for certain applications, a directoryless architecture with fine-grained partial-context thread migration can outperform or match directory-based coherence with less on-chip traffic and reduced verification complexity.

The lack of data replication, however, can limit its performance benefits; we believe more ways to avoid this limitation, such as implementing thread migration on top of simplified hardware coherence or software coherence, can be explored.

References

[1] D. Abts, S. Scott, and D. J. Lilja, "So many states, so little time: Verifying memory coherence in the cray x1," in *PDP*, 2003.

[2] AMD, "AMD64 Architecture Programmer's Manual," in *AMD64 Technology*, May 2013. [Online]. Available: http://support.amd.com/us/Processor_TechDocs/24592_APM_v1.pdf

[3] Arvind, N. Dave, and M. Katelman, "Getting formal verification into design flow," in *FM2008*, 2008.

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

[5] M. H. Cho, K. S. Shim, M. Lis, O. Khan, and S. Devadas, "Deadlock-free fine-grained thread migration," in *NOCS*, 2011.

[6] A. DeOrio, A. Bauserman, and V. Bertacco, "Post-silicon verification for cache coherence," in *ICCD*, 2008.

[7] C. Fensch and M. Cintra, "An OS-based alternative to full hardware coherence on tiled CMPs," in *HPCA*, 2008.

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

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

[10] M. Lis, K. S. Shim, M. H. Cho, O. Khan, and S. Devadas, "Directoryless shared memory coherence using execution migration," in *PDCS*, 2011.

[11] M. Lis, K. S. Shim, M. H. Cho, I. Lebedev, and S. Devadas, "Hardware-level Thread Migration in a 110-core Shared-memory Multiprocessor," in *MIT CSAIL CSG Technical Memo 512*, November 2013. [Online]. Available: http://csg.csail.mit.edu/pubs/memos/Memo-512/memo512.pdf

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

[13] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *HPCA*, 2010.

[14] S. Bell et al, "TILE64 - processor: A 64-Core SoC with mesh interconnect," in *ISSCC*, 2008.

[15] K. S. Shim, M. Lis, O. Khan, and S. Devadas, "Thread migration prediction for distributed shared caches," *Computer Architecture Letters*, Sep 2012.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 programs: characterization and methodological considerations,” in *ISCA*, 1995.

[17] M. Zhang, A. R. Lebeck, and D. J. Sorin, “Fractal coherence: Scalably verifiable cache coherence,” in *MICRO*, 2010.