Hardware-level Thread Migration in a 110-core Shared-memory Multiprocessor

Mieszko Lis* Keun Sup Shim* Myong Hyon Cho Ilia Lebedev Srinivas Devadas

Abstract

Chip multiprocessors (CMPs) with hundreds of cores are becoming a reality as technology nodes continue shrinking. Scaling directories to maintain cache coherence for these large-scale CMPs, however, remains an active area of research; at the same time, skyrocketing verification costs limit the design complexity of practical schemes.

In this paper, we describe the silicon implementation of a 110-core chip multiprocessor with unified shared memory in a 45nm ASIC. Unlike traditional designs, the architecture implements a directory-free remote cache access protocol supplemented with partial-context fine-granularity thread migration to take advantage of spatio-temporal locality. Through RTL-level simulation, we demonstrate that, with the support of partial context thread migration a directoryless architecture can outperform or match conventional directory-based designs for several applications while reducing silicon area.

1. Introduction

While process technology scaling has continued to allow for more and more transistors on a die, threshold voltage constraints have put an end to automatic power benefits due to scaling. Largely because of this power wall, advanced high-frequency designs have in practice been replaced with designs that contain several lower-frequency cores, and forward-looking pundits predict large-scale chip multiprocessors (CMPs) with thousands of cores. To support a unified shared memory abstraction—the most widely used parallel programming model—designers have turned to directory-based cache coherence, in which complex protocols ensure agreement among per-core private caches. For large-scale CMPs, however, the scalability of directories is arguably a critical challenge since the area required for directories and coherence traffic overhead keeps increasing along with the core count. Although some recent works propose more scalable directories in terms of area and performance (e.g., [13, 28]), the design and verification complexity of directories and complex coherence protocols still remain significant and not easily scalable to a large number of cores [1, 12, 35].

Various alternative designs have therefore been explored by the community. For example, the burden of cache coherence is transferred from hardware to the operating system (OS) and software in [21, 33], while DeNovo [11] offers a simplified hardware coherence protocol at the cost of relying on more disciplined shared-memory programming models. Intel's 48-core Single-Chip Cloud Computer (SCC) entirely forgoes a hardware coherence mechanism and requires the programmer to explicitly manage data coherence [24].

A straightforward approach to support shared memory without using directories and a coherence protocol is to disallow data replication among caches and use the hardware support of remote cache access for remotely cached data, as proposed by Fensch and Cintra [14]. Since only one copy is ever cached on-chip, coherence is trivially ensured without the need for directories. However, such an architecture cannot take advantage of data locality for remote data. [14] addresses this through the aid of the OS and a relaxed memory model. [23], meanwhile, have proposed fine-grained thread migration to complement remote cache access, while [29] refined this design to predict beneficial thread migration automatically.

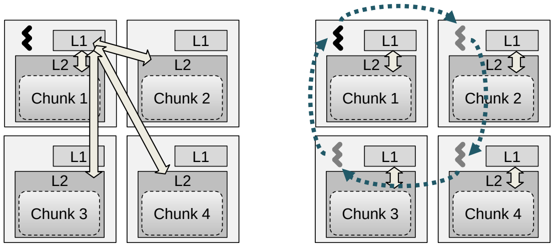
In this paper, we describe the 110-core Execution Migration Machine (EM²)—the first-ever silicon implementation to support hardware-level thread migration (extended from [23]) in a 45nm ASIC, and demonstrate that it can outperform or match traditional directory-based cache-coherence on several workloads. The novel contributions of this paper are:

1. a detailed description of the first implementation of hardware-level thread migration in a 110-core CMP;
2. the extension of [23] and [29] to include *partial context migration*, which significantly reduces thread migration costs by transferring only an automatically predicted useful part of the context;
3. an exploration of what computation patterns benefit from thread migration;
4. a detailed performance comparison vs. an "ideal" cache coherence baseline implemented in RTL.

2. Fine-grained thread migration

2.1. Motivation

When large data structures that do not fit in a single cache are shared by multiple threads or iteratively accessed even by a



(a) Directory-based / RA-only (b) Thread migration

Figure 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.

[*] equal contribution

single thread, the data are typically distributed across multiple shared cache slices to minimize expensive off-chip accesses. This raises the need for a thread to access data mapped at remote caches often with high spatio-temporal locality, which is prevalent in many applications; for example, a database request might result in a series of phases, each consisting of many accesses to contiguous stretches of data.

In a large multicore architecture without efficient thread migration, this pattern results in large amounts of on-chip network traffic. 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 last-level cache slices located in different tiles, the data must be brought to the core where the thread is running. For example, 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 (see Figure 1a); in an architecture based on remote cache access, each request to non-local data would result in a request-response pair sent across the on-chip interconnect. Because much of the dynamic power in large multicores is consumed in the interconnect, these data movement patterns incur a significant power cost.

If threads can be efficiently migrated across the chip, however, the on-chip data movement—and with it, energy use—can be significantly reduced. Instead of transferring data to feed the computing thread, the thread itself can migrate to follow the data (see Figure 1b); if the thread context is small compared to the data that would otherwise be transferred, moving the thread can be a huge win. Next, we argue that these requirements call for a simple, efficient hardware-level implementation of thread migration at the architecture level, and outline a memory access model which makes thread migration automatic and transparent to the programmer.

2.2. The need for efficient thread migration

Moving thread execution from one processor to another has long been a common feature in operating systems. This OS-mediated form of migration, however, is far too slow to make migrating threads for more efficient cache access viable: just moving the thread takes many hundreds of cycles at best (indeed, OSs generally avoid rebalancing processor core queues when possible). In addition, commodity processors are simply not designed to support migration efficiently: while context switch time is a design consideration, the very coarse granularity of OS-driven thread movement means that optimizing for fast migration is not.

Similarly, existing descriptions of hardware-level thread migration do not focus primarily on fast, efficient migrations. Thread Motion [27], for example, uses special microinstructions to write the thread context to the cache and leverages the underlying MESI coherence protocol to move threads via the last-level cache. The considerable on-chip traffic and delays that result when the coherence protocol contacts the directory, invalidates sharers, and moves the cache line, is acceptable for the 1000-cycle granularity of the centralized thread balancing logic, but not for the reduction in on-chip interconnect traffic that is the focus of our paper. Similarly, hardware-level migration among cores via a single, centrally scheduled pool of inactive threads has been described in a four-core CMP [5]; designed to hide off-chip DRAM access latency, this design did not focus on migration efficiency, and, together with the round-trips required for thread-swap requests, the indirections via a per-core spill/fill buffer and the central inactive pool make it inadequate for the fine-grained migration needed to access remote caches.

We see fine-grained thread migration as an enabling technology; since the range of applications of this technique is necessarily limited by migration performance, we focus primarily on minimizing migration latency and the incurred on-chip interconnect traffic. Our all-hardware implementation can complete an inter-tile thread migration in as few as 4 cycles and, in our 110-core ASIC design, migrations will not exceed 33 cycles provided there is no network congestion.

2.3. The elements of efficient thread migration

The implementation we describe here achieves very low migration latency by migrating the thread context directly from the core onto the interconnect network. Threads migrate autonomously and directly into their target cores: there are no centralized thread storage facilities, and there is no central migration arbiter. This means that a single migration only incurs the delay of one core-to-core message, without the round-trip messages that would arise in a centrally coordinated system.

Each of our chip's cores (described in more detail in Section 3) contains two separate thread contexts: a *native* context and a *guest* context. A core's native context may only execute the thread that originated on the core; the guest contexts serve all other threads, and evict threads back to their native cores if too many threads contend for the guest context. Together with a separate on-chip network for threads returning to their native cores, this avoids protocol-level deadlock because the native core can always accept a returning thread [9].

To further improve migration performance, our implementation can reduce migration message sizes (and therefore the bandwidth required) by migrating just enough of the thread context to perform its task on the destination core. To simplify hardware support for this *partial context* migration, our chip follows a custom stack-machine ISA (see Section 3).

In this scheme, a thread migrating out of its native core can bring along only a few entries from the top of the stack; the minimum useful migration size on our chip fits into two 64-bit flits. Our implementation of partial migration is robust: if the migrating thread brought along too few or too many entries, it is automatically transferred to its native core to access them.

Although our ASIC implementation employs a somewhat simpler stack-based variant, it is certainly possible to implement partial context migration in a register-based architecture, with memory access patterns remaining similar. While in this paper we only include cycle-accurate RTL-level simulations using the stack-based variant, additional studies using higher-level simulators on a register-based architecture show same performance trends as we report here.

The final component of efficient migration is deciding *when* the thread should migrate. Our design uses a learning migration predictor to migrate only when the reduction in on-chip network traffic is likely to outweigh the migration costs.

2.4. Shared memory

The EM² shared memory abstraction is based on a remote cache access (RA) paradigm. In this scheme, each load or store access to an address cached in a different core incurs a word-granularity round-trip message to the tile allowed to cache the address, and the retrieved data is never cached locally (the combination of word-level access and no local caching ensures correct memory semantics). 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. This makes it easy to compute which tile can cache the data.

To accelerate this base scheme, our design can automatically turn contiguous sequences of remote cache accesses into migration to the core where the data is cached followed by a sequence of local accesses. For each access to memory cached on a remote core, an instruction-address-based decision algorithm (see Section 3.5) determines whether the thread should migrate or execute a remote access (see Figure 2).

The protocol for accessing address A by thread T executing on core C is as follows:

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*), and the predictor indicates remote access,
   (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.
4. if $H \neq C$ (a *core miss*), and the predictor indicates migration,
   (a) interrupt the execution of the thread on C (as for a precise exception),
   (b) migrate the microarchitectural state to H via the on-chip interconnect:
      i. if H is the native core for T, place it in the native context slot;
      ii. otherwise, if the guest slot on H contains another thread T', evict T' to its native core N';[1] next, 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).

Described in detail in Section 3, our implementation of the combined architecture (EM²) is significantly less complex than a directory-based cache coherence protocol. Furthermore, correctness arguments do not depend on the number of cores, and, without the many transient states endemic in coherence protocols, EM² is far easier to reason about.

2.5. Virtual memory and OS implications

Although our test chip follows the accelerator model and does not support virtual memory and does not require a full operating system, fine-grained migration can be equally well implemented in a full-fledged CPU architecture. Virtual addressing at first sight potentially delays the local-vs-remote decision by one cycle (since the physical address must be resolved via a TLB lookup), but in a distributed shared cache architecture this lookup is already required to resolve which tile caches the data (if the L1 cache is virtually addressed, this lookup can proceed in parallel with the L1 access as usual). Program-initiated OS system calls and device access occasionally require that the thread remain pinned to a core for some number of instructions; these can be accomplished by migrating the thread to its native context on the relevant instruction.[2] OS-initiated tasks such as process scheduling and load rebalancing typically take place at a granularity of many milliseconds, and can be supported by requiring each thread to return to its native core every so often.

3. The EM² silicon implementation

In this section, we describe in detail the implementation of a test chip that exploits the techniques we propose here, and which we use for evaluation in Section 5. The physical chip comprises approximately 357,000,000 transistors on a 10 mm × 10 mm die in 45nm ASIC technology, using a 476-pin wirebond package.

3.1. System architecture

The chip we evaluate in this paper consists of 110 homogeneous tiles placed on a 10×11 grid, connected via an on-chip network. Figure 3 shows the actual chip layout. In lieu of a DRAM interface, our test chip exposes the two networks that carry off-chip memory traffic via a programmable rate-matching interface; this, in turn, connects to a maximum of 16GB of DRAM via a controller implemented in an FPGA.

Tiles are connected by six independent on-chip networks: two networks carry migration/eviction traffic, another two carry remote-access requests/responses, and a further two ex-

[1] Evictions must wait for any outstanding remote accesses to complete in addition to waiting for DRAM → cache responses.

[2] In fact, our ASIC implementation uses this approach to allow the program to access various statistics tables.

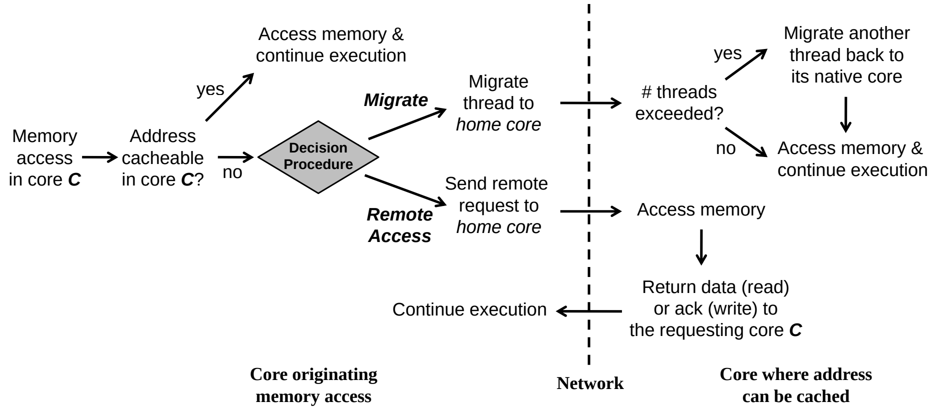

Memory access in core *C*
Address cacheable in core *C*?
yes
Access memory & continue execution
no
Decision Procedure
Migrate
Migrate thread to *home core*
Remote Access
Send remote request to *home core*
# threads exceeded?
yes
Migrate another thread back to its native core
no
Access memory & continue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

Figure 2: The EM² implementation of shared memory leverages a combination of remote access and thread migration. Memory accesses to addresses not assigned to the local core can either result in a remote cache access or cause the execution context to be migrated to the relevant core.

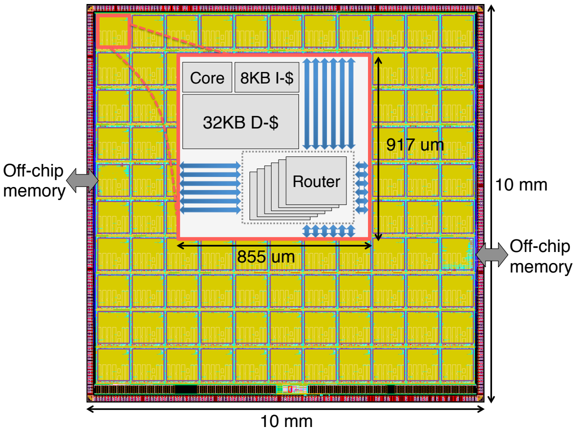


Figure 3: The 110-core EM² chip layout

ternal DRAM requests/responses; in each case, two networks are required to ensure deadlock-free operation [9].

The networks are arranged in a 2D mesh geometry: each tile contains six Network-on-Chip (NoC) routers which link to the corresponding routers in the neighboring tiles. Each network carries 64-bit flits using wormhole flow control and dimension order routing. The routers are ingress-buffered, and are capable of single-cycle forwarding under congestion-free conditions, a technique feasible even in multi-GHz designs [22].

3.2. Memory hierarchy

The memory subsystem consists of a single level (L1) of instruction and data caches, and a backing store implemented in external DRAM. Each tile contains an 8KB read-only instruction cache and a 32KB data cache, for a total of 4.4MB on-chip cache capacity; the caches are capable of single-cycle read hits and two-cycle write hits. The entire memory address space of 16GB is divided into 110 non-overlapping regions as required by the EM² shared memory semantics (see Section 2), and each tile's data cache may only cache the address range assigned to it. In addition to serving local and remote requests for the address range assigned to it, the data cache block also provides an interface to remote caches via the remote-access protocol. Memory is word-addressable and there is no virtual address translation; cache lines are 32 bytes.

3.3. Stack-based core architecture

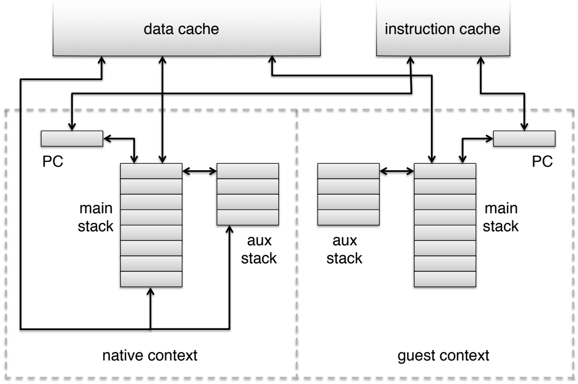


Figure 4: The processor core consists of two contexts that share the same fetch/execute pipeline. Each context comprises two stacks and a program counter, while the cache ports, migration network ports (not shown), and the migration predictor (not shown) are shared between the contexts. Stacks of the native context are backed by the data cache in the event of overflow or underflow.

To simplify the implementation of partial context migration and maximally reduce on-chip bit movement, EM² cores implement a custom 32-bit stack-based architecture (cf. Figure 4). Since the likelihood of the context being necessary increases toward the top of the stack from the nature of a stack-based ISA, a migrating thread can take along only as much of its context as is required by only migrating the top part of the stack. Furthermore, the amount of the context to transfer can be easily controlled with a single parameter, which is the depth of the stack to migrate (i.e., the number of stack entries from the top of the stack).

To reduce CPU area, the EM² core contains neither a floating point unit nor an integer divider circuit. The core is a two-stage pipeline with a top-of-stack bypass that allows an instruction's

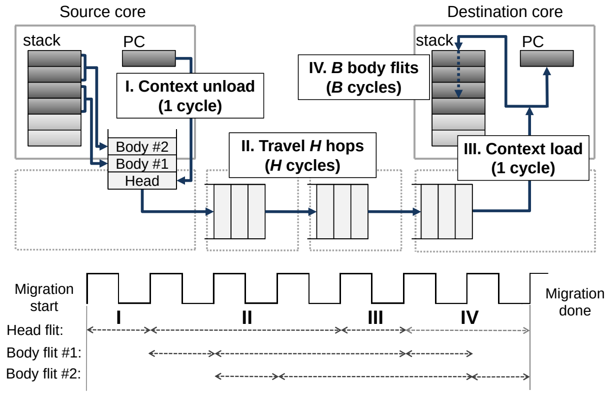


Figure 5: Hardware-level thread migration via the on-chip interconnect. Only the main stack is shown for simplicity.

arguments to be sourced from the previous instruction's ALU outputs. Each context has two stacks, *main* and *auxiliary*: most instructions take their arguments from the top entries of the main stack and leave their result on the top of the main stack, while the auxiliary stack can only be used to copy or move data from/to the top of the main stack; special instructions rearrange the top four elements of the main stack. The sizes of the main stack and the auxiliary stack are 16 and 8 entries. On stack overflow or underflow, the core automatically spills or refills the stack from the data cache; in a sense, the main and auxiliary stacks serve as caches for conceptually infinite stacks stored in memory.

To ensure deadlock-free thread migration in all cases, the core contains two thread contexts, called a *native* context and a *guest* context (both contexts share the same I$ port, which means that they do not execute concurrently). Each thread has a unique *native context* where no other thread can execute; when a thread wishes to execute in another core, it must execute in that core's guest context [9]. Functionally, the two contexts are nearly identical; the differences consist of the data cache interface in the native context that supports stack spills and refills (in a guest context stacks are not backed by memory, and stack underflow/overflow causes the thread to migrate back to its native context where the stacks can be spilled or refilled), and the thread eviction logic and associated link to the on-chip eviction network in the guest context.

3.4. Thread migration implementation

Whenever the thread migrates out of its native core, it has the option of transmitting only the part of its thread context that it expects to use at the destination core. In each packet, the first (head) flit encodes the destination packet length as well as the thread's ID and the program counter, as well as the number of main stack and auxiliary stack elements in body flits that follow. The smallest useful migration packet consists of one head flit and one body flit which contains two 32-bit stack entries. Migrations from a guest context must transmit all of the occupied stack entries, since guest context stacks are not backed by memory.

Figure 5 illustrates how the processor cores and the on-chip network efficiently support fast instruction-granularity thread migration. When the core fetches an instruction that triggers a migration (for example, because of a memory access to data cached in a remote tile), the migration destination is computed and, if there is no network congestion, the migration packet's head flit is serialized into the on-chip router buffers in the same clock cycle. While the head flit transits the on-chip network, the remaining flits are serialized into the router buffer in a pipelined fashion. Once the packet has arrived at the destination NoC router and the destination core context is free, it is directly deserialized; the next instruction is fetched as soon as the program counter is available and the instruction cache access proceeds in parallel with the deserialization of the migrated stack entries. In our implementation, assuming a thread migrates H hops with B body flits, the overall thread migration latency amounts to $1+H+1+B$ cycles from the time a migrating instruction is fetched at the source core to when the thread begins execution at the destination core. In the EM² chip, H varies from 1 (nearest neighbor core) to 19 (the maximum number of hops for 10×11 mesh), and B varies from 1 (two main stack entries and no auxiliary stack entries) to 12 (sixteen main stack entries and eight auxiliary stack entries, two entries per flit); this results in the very low migration latency, ranging from the minimum of 4 cycles to the maximum of 33 cycles (assuming no network congestion).[3]

While a native context is reserved for its native thread and therefore is always free when this thread arrives, a guest context might be executing another thread when a migration packet arrives. In this case, the newly arrived thread is buffered until the currently executing thread has had a chance to complete some (configurable) number of instructions; then, the active guest thread is evicted to make room for the newly arrived one. During the eviction process the entire active context is serialized just as in the case of a migration (the eviction network is used to avoid deadlock), and once the last flit of the eviction packet has entered the network the newly arrived thread is unloaded from the network and begins execution.

3.5. Migration prediction

EM² can improve performance and reduce on-chip traffic by turning sequences of memory accesses to the same remote cache into migrations followed by local cache accesses (see Section 2.4). To detect sequences suitable for migration, each EM² core includes a learning *migration predictor* [29]—a program counter (PC)-indexed, direct-mapped data structure shown in Figure 6. In addition to detecting migration-friendly memory references and making a remote-access vs migration decision for every non-local load and store as in [29], our predictor further reduces on-chip network traffic by learning

[3] Although it is possible to migrate with no main stack entries, this is unusual, because most instructions require one or two words on the stack to perform computations. The minimum latency in this case is still 4 cycles, because execution must wait for the I$ fetch to complete anyway.

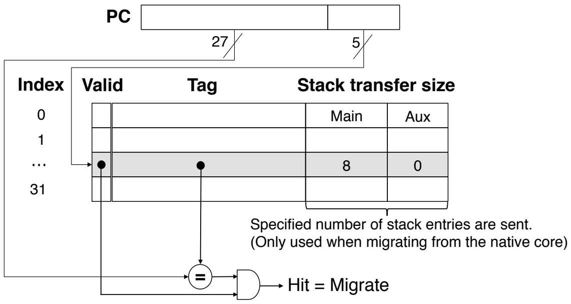

PC
27
5
Index
Valid
Tag
Stack transfer size
0
1
...
31
Main
Aux
8
0
Specified number of stack entries are sent.
(Only used when migrating from the native core)
=
Hit = Migrate

Figure 6: Each core has a PC-based migration predictor.

and deciding how much of the stack should be transferred for every migrating instruction.

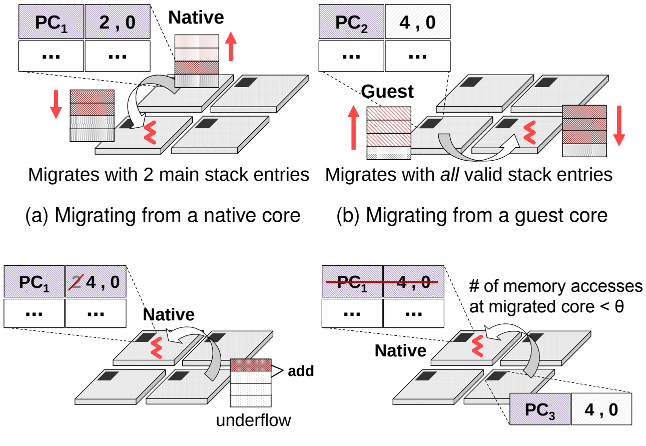
The predictor bases these decisions on the instruction's PC. In most programs, sequences of consecutive memory accesses to the same home core and context usage patterns within those sequences are highly correlated with the instructions being executed, and those patterns are fairly consistent and repetitive across program execution. Each predictor has 32 entries, each of which consists of a tag for the PC and the transfer sizes for the main and auxiliary stacks.

Detecting contiguous access sequences. Initially, the predictor table is empty, and all instructions are predicted to be remote-access. To detect memory access sequences suitable for migration, the predictor tracks how many consecutive accesses to the same remote core have been made, and, if this count exceeds a (configurable) threshold θ, inserts the PC of the instruction at the *start* of the sequence into the predictor. To accomplish this, each thread tracks (1) *home*, which maintains the home location (core ID) for the memory address being requested, (2) *depth*, which counts the number of contiguous times made to an address cached at the core identified by the *home* field, and (3) *start PC*, which tracks the PC of the first instruction that accessed memory at the *home* core.

When a thread T executes a memory instruction for address A whose PC is P, it must

1. find the home core H for A (e.g., by masking the appropriate bits);
2. if $home = H$ (i.e., memory access to the same home core as that of the previous memory access),
   (a) if $depth < \theta$, increment $depth$ by one;
   (b) otherwise, if $depth = \theta$, insert *start PC* into the predictor table;
3. if $home \neq H$ (i.e., a new sequence starts with a new home core),
   (a) if $depth < \theta$, invalidate any existing entry for *start PC* in the predictor table (thus making *start PC* non-migratory);
   (b) reset the current sequence counter (i.e., $home \leftarrow H$, $start\ PC \leftarrow P$, $depth \leftarrow 1$).

When an instruction is first inserted into the predictor, the stack transfer sizes for the main and auxiliary stack are set to the default values of 8 (half of the main stack) and 0, respectively.



(a) Migrating from a native core (b) Migrating from a guest core

(c) Learning the best context size (d) Learning from misprediction

Figure 7: Decision/Learning mechanism of the migration predictor

Migration prediction for memory accesses. When a load or store instruction attempts to access an address that cannot be cached at the core where the thread is currently running (a *core miss*), the predictor uses the instruction's address (i.e., the PC) to look up the table of migrating sequences. If the PC is in the table, the predictor instructs the thread to migrate; otherwise, to perform a remote access.

When the predictor instructs a thread to migrate from its *native* core to another core, it also provides the number of main and auxiliary stack entries that should be migrated (cf. Figure 7a). Because the stacks in the guest context are not backed by memory, however, all valid stack entries must be transferred (cf. Figure 7b).

Feedback and learning. To learn how many stack entries to send when migrating from a native context at runtime, the native context keeps track of the *start PC* that caused the last migration. When the thread arrives *back* at its native core, it reports the reason for its return: when the thread migrated back because of stack overflow (or underflow), the stack transfer size of the corresponding *start PC* is decremented (or incremented) accordingly (cf. Figure 7c). In this case, less (or more) of the stack will be brought along the next time around, eventually reducing the number of unnecessary migrations due to stack overflow and underflow.

The returning thread also reports the number of local memory instructions it executed at the core it originally migrated to. If the thread returns without having made θ accesses, the corresponding *start PC* is removed from the predictor table and the access sequence reverts to remote access (cf. Figure 7d).[4] This allows the predictor to respond to runtime changes in program behavior.

[4] Returns caused by *evictions* from the remote core do not trigger removal, since the thread might have completed θ accesses had it not been evicted.

3.6. The instruction set

Stacks. Each core context contains a main stack (16 entries) and an auxiliary stack (8 entries), and instructions operate the top of those stacks much like RISC instructions operate on registers. On stack overflow or underflow, the core automatically accesses the data cache to spill or refill the core stacks. Stacks naturally and elegantly support partial context migration, since the topmost entries which are migrated as a partial context are exactly the ones that the next few instructions will use.

Computation and stack manipulation. The core implements the usual arithmetic, logical, and comparison instructions on 32-bit integers, with the exception of hardware divide. Those instructions consume one or two elements from the main stack and push their results back there. Instructions in the `push` class place immediates on the stack, and variants that place the thread ID, core ID, or the PC on top of the stack help effect inter-thread synchronization.

To make stack management easier, the top four entries of the main stack can be rearranged using a set of stack manipulation instructions. Access to deeper stack entries can be achieved via instructions that move or copy the top of the main stack onto the auxiliary stack and back.

Control flow and explicit migration. Flow control is effected via the usual conditional branches (which are relative) and unconditional jumps and calls (relative or absolute). Threads can be manually migrated using the `migrate` instruction, and efficiently spawned on remote cores via the `newthread` instruction.

Memory instructions. Word-granularity loads and stores come in EM (migrating) and RA (remote access) versions, as well as in a generic version which defers the decision to the migration predictor. The EM and generic versions encode the stack depths that should be migrated, which can be used instead of the predictor-determined depths. Providing manual and automatic versions gives the user both convenience and maximum control.

Similarly, stores come in acked as well as fire-and-forget variants. Together with per-instruction memory fences, the ack variant provides sequential consistency while the fire-and-forget version may be used if a higher-level protocol obviates the need for per-word guarantees. Load-reserve and store-conditional instructions provide atomic read-modify-write access, and come in EM and RA flavors.

4. Methods

4.1. RTL simulation

To evaluate the EM² implementation, we chose an idealized cache-coherent baseline architecture with a two-level cache hierarchy (a private L1 data cache and a shared L2 cache). In this scheme, the L2 is distributed evenly among the 110 tiles and the size of each L2 slice is 512KB. An L1 miss results in a cache line being fetched from the L2 slice that corresponds to the requested address (which may be on the same tile as the L1 cache or on a different tile). While this cache fetch request must still traverse the network to the correct L2 slice and bring the cache line back, our cache-coherent baseline is idealized in the sense that rather than focusing on the details of a specific coherence protocol implementation, it does not include a directory and never generates any coherence traffic (such as invalidates and acknowledgements); coherence among caches is ensured "magically" by the simulation infrastructure. While such an idealized implementation is impossible to implement in hardware, it represents an upper bound on the performance of any implementable directory coherence protocol, and serves as the ultimate baseline for performance comparisons.

To obtain the on-chip traffic levels and completion times for our architecture, we began with the post-tapeout RTL of the EM² chip, removed such ASIC-specific features as scan chains and modules used to collect various statistics at runtime, and added the same shared-L2 cache hierarchy as the cache-coherent baseline. Since our focus is on comparing on-chip performance, the working set for our benchmarks is sized to fit in the entire shared-L2 aggregate capacity. All of the simulations used the entire 110-core chip RTL; for each benchmark, we report the completion times as well as the total amount of on-chip network traffic (i.e., the number of times any flit traveled across any router crossbar).

The ideal CC simulations only run one thread in each core, and therefore only use the native context. Although the EM² simulations can use the *storage space* of both contexts in a given core, this does not increase the parallelism available to EM²: because the two contexts share the same I$ port, only one context can be executing an instruction at any given time.

Both simulations use the same 8 KB L1 instruction cache as the EM² chip. Unlike the PC, instruction cache entries are not migrated as part of the thread context; while this might at first appear to be a disadvantage when a thread first migrates to a new core, we have observed that in practice at steady state the I$ has usually already been filled (either by other threads or by previous iterations that execute the same instruction sequence), and the I$ hit rate remains high.

4.2. Area and power estimates

Area and power estimates were obtained by synthesizing RTL using Synopsys Design Compiler (DC). For the EM² version, we used the post-tapeout RTL with the scan-chains and statistics modules deleted; we reused the same IBM 45nm SOI process with the ARM sc12 low-power ASIC cell library and SRAM blocks generated by IBM Memory Compiler. Synthesis targeted a clock frequency of 800MHz, and leveraged DC's automatic clock-gating feature.

To give an idea of how these costs compare against that of a well-understood, realistic architecture, we also estimated the area and leakage power of an equivalent design where the data caches are kept coherent via a directory-based MESI protocol (CC). We chose an exact sharer representation (one bit for each of the 110 sharers) and either the same number of entries as in the data cache (CC 100%) or half the entries (CC 50%);[5] in both versions the directory was 4-way set-associative. To estimate the area and leakage power of the directory, we synthesized a 4-way version of the data cache controller from EM² chip with SRAMs sized for each directory configuration, using the same synthesis constraints (since a directory controller is somewhat more complex than a cache controller, this approach likely results in a slight underestimate).

For area and leakage power, we report the synthesis estimates computed by DC, i.e., the total cell area in μm^2 and the total leakage power. While all of these quantities typically change somewhat post-layout (because of factors like routing congestion or buffers inserted to avoid hold-time violations), we believe that synthesis results are sufficient to make architectural comparisons.

5. Evaluation

5.1. Performance tradeoff factors

To precisely understand the conditions under which fast thread migration results in improved performance, we created a simple parameterized benchmark that executes a sequence of loads to memory assigned to a remote L2 slice. There are two parameters: the *run length* is the number of contiguous accesses made to the given address range, and *cache misses* is the number of L1 misses these accesses induce (in other words, this determines the stride of the access sequence); we also varied the on-chip distance between the tile where the thread originates and the tile whose L2 caches the requested addresses.

Figure 8 shows how a program that only makes remote cache accesses (RA-only) compares with a program that migrates to the destination core 4 hops away, makes the memory accesses, and returns to the core where it originated (EM²), where the migrated context size is 4, 8, and 12 stack entries (EM²-4, EM²-8, and EM²-12). Since the same L1 cache is always accessed—locally or remotely—both versions result in exactly the same L1 cache misses, and the only relevant parameter is the run length. For a singleton access (*run length* = 1), RA is slightly faster than any of the migration variants because the two migration packets involved are longer than the RA request/response pair, and, for the same reason, induce much more network traffic. For multiple accesses, however, the single migration round-trip followed by local cache accesses performs better than the multiple remote cache access round trips, and the advantage of the migration-based solution grows as the run length increases.

The tradeoff against our "ideal cache coherence" private-cache baseline (CC) is less straightforward than against RA: while CC will still make a separate request to load every cache *line*, subsequent accesses to the same cache line will result in L1 cache hits and no network traffic. Figure 9 illustrates how the performance of CC and EM² depends on how many times the same cache line is reused in 8 accesses. When all 8 accesses are to the same cache line (*cache misses* = 1), CC requires one round-trip to fetch the entire cache line, and is slightly faster than EM², which needs to unload the thread context, transfer it, and load it in the destination core. Once the number of misses grows, however, the multiple round-trips required in CC become more costly than the context load/unload penalty of the one round-trip migration, and EM² performs better. And in all cases, EM² can induce less on-chip network traffic: even in the one-miss case where CC is faster, the thread context that EM² has to migrate is often smaller than the CC request and the cache line that is fetched.

Finally, Figure 10 examines how the three schemes are affected by the on-chip distance between the core where the thread originates and the core that caches the requested data (with *run length* = 8 and *cache misses* = 2). RA, which requires a round-trip access for every word, grows the fastest (i.e., eight round-trips), while CC, which only needs a round-trip cache line fetch for every L1 miss (i.e., two round-trips), grows much more slowly. Because EM² only requires one round-trip for all accesses, the distance traveled is not a significant factor in performance.

5.2. Benchmark performance

Figure 11 shows how the performance of EM² compares to the ideal CC baseline for several benchmarks. These include: (1) single-threaded *memcpy* in next-neighbor (*near*) and cross-chip (*far*) variants, (2) parallel *k*-fold cross-validation (*par-cv*), a machine learning technique that uses stochastic gradient learning to improve model accuracy, (3) 2D Jacobi iteration (*jacobi*), a widely used algorithm to solve partial differential equations, and (4) partial table scan (*tbscan*), which executes queries that scan through a part of a globally shared data table distributed among the cache shards. We first note some overall trends and then discuss each benchmark in detail below.

Overall remarks. First, Figure 11a illustrates the overall performance (i.e., completion time) and on-chip network traffic of the ideal directory-based baseline (CC), the remote-access-only variant (RA), and the EM² architecture. Overall, EM² always outperforms RA, offering up to 3.9× reduction in run time, and as well or better than CC in all cases except one. Throughout, EM² also offers significant reductions in on-chip network traffic, up to 42× less traffic than CC for *par-cv*.

Migration rates, shown in Figure 11c, range from 0.2 to 20.9 migrations per 1,000 instructions depending on the benchmark. These quantities justify our focus on *efficient* thread movement: if migrations occur at the rate of nearly one in every hundred to thousand instructions, taking 1000+ cycles to move a thread to a different core would indeed incur a prohibitive performance impact. Most migrations are caused by data accesses, with stack under/overflow migrations at a negligible level, and evictions significant only in the *tbscan* benchmarks.

Even with many threads, effective migration latencies are

[5] Note that because multiple data caches can "gang up" on the same directory slice, the 100% version does not guarantee freedom from directory-capacity invalidations.

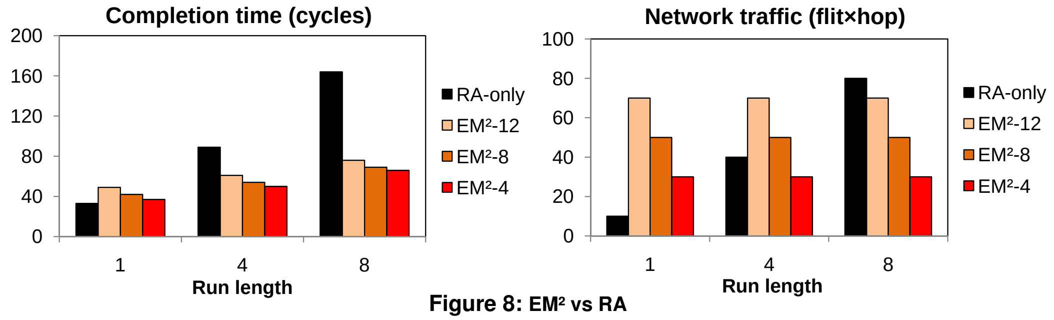


Figure 8: EM² vs RA

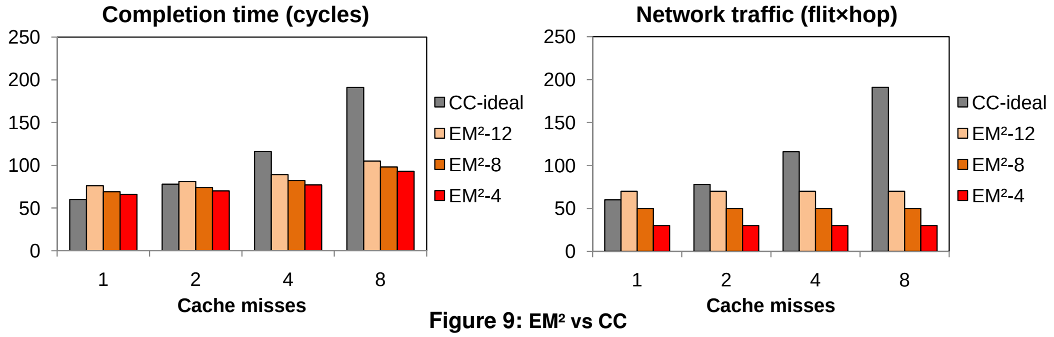


Figure 9: EM² vs CC

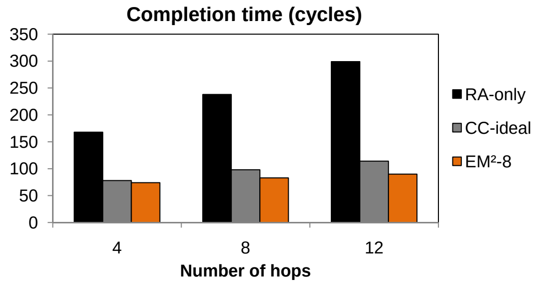


Figure 10: The effect of distance on RA, CC and EM²

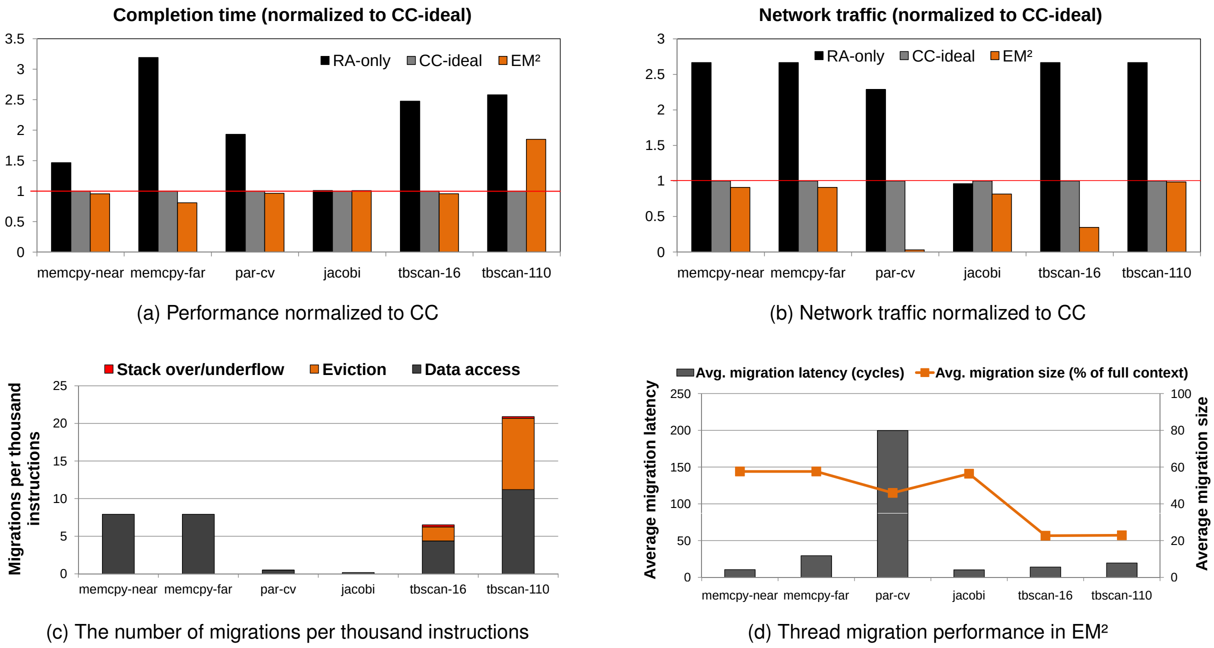
low (Figure 11d, bars), with the effect of distance clearly seen for the *near* and *far* variants of *memcpy*; the only exception here is *par-cv*, in which the migration latency is a direct consequence of delays due to inter-thread synchronization (as we explain below). At the same time, migration sizes (Figure 11d, line) vary significantly, and stay well below the 60% mark (44% on average): since most of the on-chip traffic in the EM² case is due to migrations, forgoing partial-context migration support would have significantly increased the on-chip traffic (cf. Figure 11b).

Memory copy. The *memcpy-near* and *memcpy-far* benchmarks copy 32 KB (the size of an L1 data cache) from a memory address range allocated to a next-neighbor tile (*memcpy-near*) or a tile at the maximum distance across the 110-core chip (*mempcy-far*). In both cases, EM² is able to repeatedly migrate to the source tile, load up a full thread context's worth of data, and migrate back to store the data at the destination addresses; because the maximum context size exceeds the cache line size that ideal CC fetches, EM² has to make fewer trips and performs better both in terms of completion time and network traffic. Distance is a significant factor in performance—the fewer round-trips of EM² make a bigger difference when the source and destination cores are far apart—but does not change the % improvement in network traffic, since that is determined by the the total amount of data transferred in EM² and CC.

Partial table scan. In this benchmark, random SQL-like queries are assigned to separate threads, and the table that is searched is distributed in equal chunks among the per-tile L2 caches. We show two variants: a light-load version where only 16 threads are active at a time (*tbscan-16*) and a full-load version where all of the 110 available threads execute concurrently (*tbscan-110*); under light load, EM² finishes slightly faster than CC-ideal and significantly reduces network traffic (2.9×), while under full load EM² is 1.8× slower than CC-ideal and has the same level of network traffic.

Why such a large difference? Under light load, EM² takes full advantage of data locality, which allows it to significantly reduce on-chip network traffic, but performs only slightly better than CC-ideal because queries that access the same data chunks compete for access to the same core and effectively serialize some of the computation. Because the queries are random, this effect grows as the total number of threads increases (Figure 12), resulting in very high thread eviction rates under full load (Figure 11c); this introduces additional delays and network traffic as threads ping-pong between their home core and the core that caches the data they need.

This ping-pong effect, and the associated on-chip traffic, can be reduced by guaranteeing that each thread can perform N (configurable in hardware) memory accesses before being evicted from a guest context. Figure 12 illustrates how *tbscan* performs when $N = 10$ and $N = 100$: a longer guaranteed guest-context occupation time results in up to 2× reductions in network traffic at the cost of a small penalty in completion time due to the increased level of serialization. This highlights an effective tradeoff between performance and power: with more serialization, EM² can use far less dynamic power due to



(a) Performance normalized to CC

(b) Network traffic normalized to CC

(c) The number of migrations per thousand instructions

(d) Thread migration performance in EM²

Figure 11: The evaluation of EM²

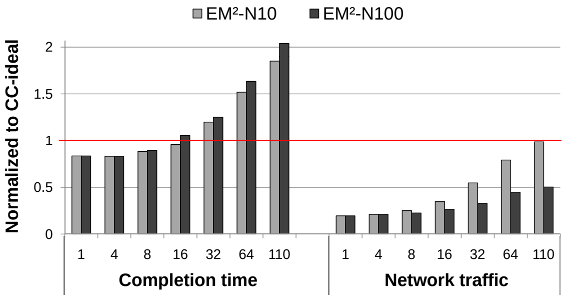


Figure 12: Performance and network traffic with different number of threads for *tbscan* under EM²

on-chip network traffic (and because fewer cores are actively computing) if the application can tolerate lower performance.

Parallel K-fold cross validation. In the *k-fold cross-validation* technique common in machine learning, data samples are split into k disjoint chunks and used to run k independent leave-one-out experiments. For each experiment, $k-1$ chunks constitute the training set and the remaining chunk is used for testing; the results are then averaged to estimate the final prediction accuracy of the algorithm being trained. Since the experiments are computationally independent, naturally map to multiple threads (indeed, for sequential machine learning algorithms, such as stochastic gradient descent, this is the *only* practical form of parallelization because the model used in each experiment is necessarily sequential). The chunks are typically spread across the shared cache shards, and each experiment repeatedly accesses a given chunk before moving on to the next one.

With overall completion time slightly better under EM² than under CC-ideal and much better than under RA-only, *par-cv* stands out for its 42× reduction in on-chip network traffic vs. CC-ideal (96× vs. RA). This is because the cost of every migration is amortized by a large amount of local cache accesses on the destination core (as the algorithm learns from the given data chunk), while CC-ideal continuously fetches more data to feed the computation.

Completion time for *par-cv*, however, is only slightly better because of the nearly 200-cycle average migration times at full 110-thread utilization (Figure 11d). This is because of a serialization effect similar to that in *tbscan*: a thread that has finished learning on a given chunk and migrates to proceed onto the next chunk must sometimes wait en route while the previous thread finishes processing that chunk. Unlike *tbscan*, however, where the contention results from random queries, the threads in *par-cv* process the chunks in order, and avoid the penalties of eviction. As a result, at the same full utilization rate of 110 threads, *par-cv* has a better completion time under EM² but *tbscan* performs better under CC. (At a lower utilization, the average migration latency of *par-cv* falls: e.g., at 50 threads it becomes 9 cycles, making the EM² version 11% faster than CC).

2D Jacobi iteration. In its essence, the *jacobi* benchmark propagates a computation through a matrix, and so the communication it incurs is between the boundary of the 2D matrix region stored in the current core and its immediate neighbors stored in the adjacent cores. Since the data accesses are largely to a thread's own private region, intercore data transfers are a negligible factor in the overall completion time, and the runtime for all three architectures is approximately the same. Still, EM² is able to reduce the overall network traffic because it can amortize the costs of migrating by consecutively accessing many matrix elements in the boundary region, while CC-ideal has to access this data with several L2 fetches.

5.3. Area and power costs

Since the CC-ideal baseline we use for the performance evaluation above does not have directories, it does not make a good baseline for area and power comparison. Instead, we estimated the area required for MESI implementations with the directory sized to 100% and 50% of the total L1 data cache entries, and compared the area and leakage power to that of EM². The L2 cache hierarchy, which was added for more realistic performance evaluation and not a part of the actual chip, is not included here for both EM² and CC.

Table 1 summarizes the architectural components that differ. EM² requires an extra architectural context (for the guest thread) and on-chip networks for migrations and evictions as well as RA requests and responses. Our EM² implementation also includes a learning migration predictor; while this is not strictly necessary in a purely instruction-based migration design, it offers runtime performance advantages similar to those of a hardware branch predictor. In comparison, a deadlock-free implementation of MESI would replace the four migration and remote-access on-chip networks with three (for coherence requests, replies, and invalidations), implement D$ controller logic required to support the coherence protocol, and add the directory controller and associated SRAM storage.

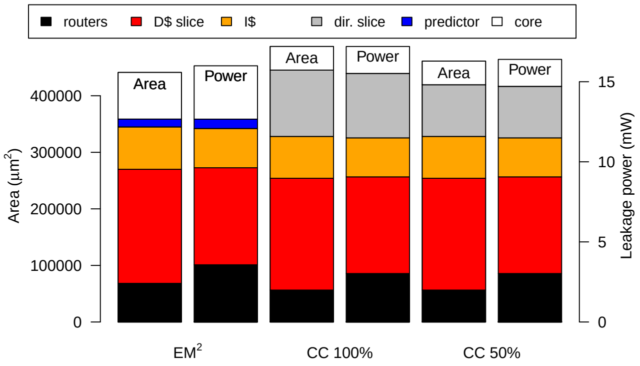


Figure 13: Relative area and leakage power costs of EM² vs. estimates for exact-sharer CC with the directory sized to 100% and 50% of the D$ entries (DC Ultra, IBM 45nm SOI hvt library, 800MHz).

Figure 13 shows how the silicon area and leakage power compare. Not surprisingly, blocks with significant SRAM storage (the instruction and data caches, as well as the directory in the CC version) were responsible for most of the area in all variants. Overall, the extra thread context and extra router present in EM² were outweighed by the area required for the directory in both the 50% and 100% versions of MESI, which suggests that EM² may be an interesting option for area-limited CMPs.

	EM²	CC
extra execution context in the core	yes	no
migration predictor logic & storage	yes	no
remote cache access support in the D$	yes	no
coherence protocol logic in the D$	no	yes
coherence directory logic & storage	no	yes
number of independent on-chip networks	6	5

Table 1: A summary of architectural costs that differ in the EM² and CC implementations.

6. Related Work

Migrating computation to accelerate data access is not itself a novel idea. Hector Garcia-Molina in 1984 introduced the idea of moving processing to data in memory bound architectures [15], and improving memory access latency via migration has been proposed using coarse-grained compiler transformations [17]. In recent years migrating execution context has re-emerged in the context of single-chip multicores. Thread migration has been used to take advantage of the overall on-chip cache capacity and improving performance of sequential programs [25]. Computation spreading [6] splits thread code into segments and migrates threads among cores assigned to the segments to improve code locality. Thread migration to a central inactive pool has been used to amortize DRAM access latency [5]. Core salvaging [26] allows programs to run on cores with permanent hardware faults provided they can migrate to access the locally damaged module at a remote core, while Thread motion [27] migrates less demanding threads to cores in a lower voltage/frequency domain to improve the overall power/performance ratios. More recently, thread migration among heterogeneous cores has been proposed to improve program bottlenecks (e.g., locks) [19]. The very fine-grained nature of the migrations contemplated in many of these proposals—for example, a thread must be able to migrate immediately because of hardware faults [26]—requires support for fast, fine-grained migration with decentralized control, with the migration decision made autonomously by each thread; an implementation has not, however, been described and evaluated in detail, and is among our contributions.

The remote-access architecture that underlies EM² has its roots in the non-uniform memory architecture (NUMA) paradigm as extended to single-die caches (NUCA [8,20]). Migration and replication (of data rather than threads) was used to speed up NUMA systems (e.g., [32]), but the differences in both interconnect delays and memory latencies make the general OS-level approaches studied inappropriate for today's fast on-chip interconnects.

NUCA architectures were applied to CMPs [3,18] and more recent research has explored the distribution and movement of data among on-chip NUCA caches with traditional and hybrid cache coherence schemes to improve data locality. Page-granularity approaches that leverage the virtual addressing system and the TLBs have been proposed to improve locality [10, 16]; CoG [2] moves pages to the "center of gravity" to improve data placement, while the O^2 scheduler [4] improves memory performance in distributed-memory multicores by trying to keep threads near their data during OS scheduling. Victim replication [34] improves performance while keeping total cache capacity high, but requires a directory to keep track of sharers. Reactive NUCA (R-NUCA) replicates read-only data based on the premise that shared read-write data do not benefit from replication [16]. Other schemes add hardware support for page migration support [7, 31], and taking advantage of the programmer's application-level knowledge to replicate not only read-only data but also read-write shared data during periods when it is not being written [30]. Our design differs from these by focusing on efficiently moving computation to data, rather than replicating and moving data to computation.

On the other end of the spectrum, moving computation to data has been proposed to provide memory coherence among per-core caches [23]. We adopt the same shared memory paradigm as a proof of concept for our fine-grained thread migration implementation, and use the same deadlock-free thread migration protocol [9]. The present work focuses on the thread migration itself, and contributes a detailed microarchitecture of a CMP with deeply integrated hardware-level thread migration; rather than the coarse-grained simulations employed in those works, we use exact RTL-level simulations and post-synthesis power estimates to report a precise comparison of EM² and an idealized upper-bound cache-coherent architecture. The migration predictor we present here is based on that of [29]; unlike the full-context predictor described there, our predictor supports stack-based partial context migration by learning how much of the thread context to migrate.

7. Conclusion

In this paper, we have demonstrated that, on many applications, a directoryless architecture supplemented with fine-grained hardware-level thread migration can outperform or match directory-based cache coherence while cutting on-chip traffic.

We have also explored some limitations of this architecture: specifically, the lack of data replication can limit the hardware's ability to take advantage of available parallelism, limiting performance benefits. In our future work, we plan to explore ways to avoid this limitation, such as implementing fine-grained thread migration on top of substrates with a hardware-based coherence protocol or software coherence.

References

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

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

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

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

[5] J. A. Brown and D. M. Tullsen, "The shared-thread multiprocessor," in *ICS*, 2008.

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

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

[8] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Distance associativity for high-performance energy-efficient non-uniform cache architectures," in *ISCA*, 2003.

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

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

[11] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *PACT*, 2011.

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

[13] M. Feldman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: a scalable directory for many-core systems," in *HPCA*, 2011.

[14] C. Fensch and M. Cintra, "An OS-Based Alternative to Full Hardware Coherence on Tiled CMPs," in *HPCA*, 2008.

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

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

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

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

[19] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," in *ASPLOS*, 2012.

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

[21] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, J. W. Meira, S. Dwarkadas, and M. Scott, "VM-based shared memory on low-latency, remote-memory-access networks," in *ISCA*, 1997.

[22] A. Kumar, P. Kundu, A. Singh, L.-S. Peh, and N. K. Jha, "A 4.6Tbits/s 3.6GHz Single-cycle NoC Router with a Novel Switch Allocator in 65nm CMOS," in *ICCD*, 2008.

[23] M. Lis, K. S. Shim, M. H. Cho, O. Khan, and S. Devadas, "Directoryless Shared Memory Coherence using Execution Migration," in *PDCS*, 2011.

[24] T. Mattson, R. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core SCC Processor: the Programmer's View," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, 2010.

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

[26] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee, "Architectural core salvaging in a multi-core processor for hard-error tolerance," in *ISCA*, 2009.

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

[28] D. Sanchez and C. Kozyrakis, "SCD: A scalable coherence directory with flexible sharer encoding," in *HPCA*, 2012.

[29] K. Shim, M. Lis, O. Khan, and S. Devadas, "Thread migration prediction for distributed shared caches," *Computer Architecture Letters*, vol. PP, no. 99, 2013.

[30] K. S. Shim, M. Lis, M. H. Cho, O. Khan, and S. Devadas, "System-level Optimizations for Memory Access in the Execution Migration Machine (EM²)," in *CAOS*, 2011.

[31] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-pages: increasing DRAM efficiency with locality-aware data placement," *SIGARCH Comput. Archit. News*, vol. 38, pp. 219–230, 2010.

[32] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating system support for improving data locality on cc-numa compute servers," *SIGPLAN Not.*, vol. 31, pp. 279–289, 1996.

[33] H. Zeffer, Z. Radović., M. Karlsson, and E. Hagersten, "TMA: A Trap-Based Memory Architecture," in *ICS*, 2006.

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

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