The Execution Migration Machine

Mieszko Lis*, Keun Sup Shim*, Myong Hyon Cho, Ilia Lebedev, Srinivas Devadas
MIT CSAIL, Cambridge, MA, USA
{mieszko, ksshim, mhcho, ilebedev, devadas}@mit.edu

Abstract

On-chip interconnect power already forms a significant portion of the power consumed by chip multiprocessors (CMPs), and with continued transistor scaling leading to higher and higher core counts, chip power will be increasingly dominated by the on-chip network. For massive multicores to be feasible, therefore, it will be necessary to significantly reduce total on-chip data movement. Since most of the traffic is related to bringing data to the locus of computation, one solution is to enable threads to efficiently migrate across the chip and execute near the data they access.

In this paper, we present the detailed implementation of hardware-level instruction-granularity thread migration in a 110-core CMP. Implemented in 45nm ASIC technology, the chip occupies 100mm^2 and is currently in the fabrication stage. With a custom stack-based ISA to enable partial context migration, when there is no network congestion, our implementation provides end-to-end migration latency of 4 cycles between neighboring cores with a minimum thread context, and 33 cycles between the farthest cores with a maximum context. To supplement a remote-cache-access-based shared memory paradigm, our cores learn a thread's data access patterns and migrate threads automatically. Through RTL-level simulation, we demonstrate that migration can reduce on-chip data movement by up to 14× at a relatively small area cost of 23%.

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.

A natural consequence is that data must cross relatively longer distances across the chip. With wire delays and power scaling much more slowly than logic gates and memories, however, global and semi-global wires are becoming impractical, and local wires driven by per-tile network-on-chip routers have become the norm in large-scale CMPs. In such devices, interconnect power is already a significant part of the power requirements: for example, in the 16-tile MIT RAW chip, implemented in 0.15µm CMOS, the interconnect consumed up to 39% of each tile's power in the interconnect [16], while in Intel's 80-tile TeraFLOPS, which was implemented in 65nm and includes two floating point MAC units, communication accounts for 28% of a tile's power [25]. As feature sizes shrink further, large-scale CMPs will be increasingly dominated by intercore communication requirements.

Reducing on-chip traffic, therefore, is becoming critical to keeping CMPs within a manageable power envelope. With threads effectively pinned to specific cores for most of their execution, a lion's share of the on-chip traffic consists of data being brought to the core that uses it. An alternative, which we explore in depth in this paper, is to allow threads to move around the chip as necessary to take advantage of a computation's spatiotemporal locality. When the migration is sufficiently fine-grained and the latency low enough, we argue, judicious thread migration can significantly reduce on-chip bit movement. Based on a recently taped-out 110-core ASIC, we describe an efficient hardware-only implementation of thread migration, and, using RTL simulation and synthesis to a 45nm ASIC library, explore architectural and performance tradeoffs.

The novel contributions of this paper are as follows:

Fine-grained hardware thread migration. Although thread (or process) movement has long been a common OS feature, the millisecond granularity makes this technique unsuitable for taking advantage of shorter-lived phenomena like fine-grained memory access locality. Our pure hardware implementation can accomplish a single thread migration in as few as 4 cycles between the closest cores when the minimum context is migrated, and 33 cycles between the farthest on our 110-core grid with the maximum possible thread context.

Instruction-granularity migration prediction. Fast thread movement requires an equally speedy method for detecting and responding to changes in memory access patterns, which precludes software-based mechanisms. For statically discoverable patterns, our compiler can annotate memory instructions as either remote-access or migratory. For access patterns that are difficult to discover statically (or that change at runtime), our implementation employs a hardware-level learning migration predictor to detect and respond to locality patterns.

Stack architecture for partial context migration. Always moving the entire thread context can be wasteful if only a portion of it will be used at the destination core. To further reduce communication, our cores implement a stack-based architecture where a migrating thread can take along only as much of its context as is required by only migrating the top of the stack: the minimum migration size is just 128 bits. Our migration predictor implementation takes advantage of this by learning the best context size for every migration. To ensure

* Mieszko Lis and Keun Sup Shim contributed equally to this work.

a simple programming model, the in-core stack registers are backed by memory and automatically spilled or refilled as necessary without user intervention.

Lightweight shared memory model. As a proof of concept for the fine-grained migration infrastructure, our chip leverages thread migration to accelerate a shared-memory mechanism based on remote cache access. We show that adding migration can result in up to 14× reduction in on-chip traffic, and, depending on computational load, significant improvements in runtime. To our knowledge, our design is the largest shared-memory CMP in terms of core count.

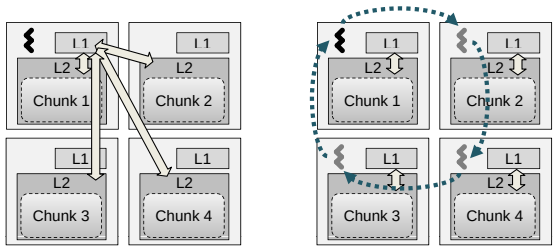
A benefit-vs-cost analysis of fine-grained thread migration. Using RTL simulation of several benchmarks on a full 110-core chip, we demonstrate that adding fine-grained thread migration to a remote-cache-access CMP significantly reduces on-chip bit movement. We report the area and leakage power of both variants in a 45nm ASIC node, and estimate those costs for an equivalent directory-based coherence design.

2. Architectural-level 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 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



(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.

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. In the remainder of this section 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 [21], 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 [4]; 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 [8].

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).[1] 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.

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: an application of thread migration

As a proof-of-concept of the feasibility and performance of our hardware-level migration architecture, we used it to implement shared memory in our 110-core CMP. Memory traffic, which in shared-memory designs constitutes a lion's share of on-chip interconnect traffic, is very sensitive to migration costs, and therefore provides a good target for optimization.

For simplicity and scalability, we implemented a remote cache access (RA) shared memory paradigm as our baseline. 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.

In addition to remote cache access, 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:
         A. if the guest slot on H contains another thread T', evict T' to its native core N'
         B. move T into the guest slot for H;
   (c) resume execution of T on H, requesting A from its cache hierarchy (and potentially accessing DRAM).

To avoid interconnect deadlock,[2] the system must ensure that all remote requests must always eventually be served. This is accomplished by a total of six on-chip subnetworks (a request-response pair for remote accesses, a migrate-evict pair for migrations, and a request-response pair for memory traffic); all of the networks must deliver packets in order. At the protocol level, evictions must also wait for any outstanding remote accesses to complete in addition to waiting for DRAM → cache responses.

Described in detail in the next section, 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

[1] Although it is certainly possible to implement partial context migration in a register-based architecture, in our ASIC we have chosen the somewhat simpler stack-based variant.

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

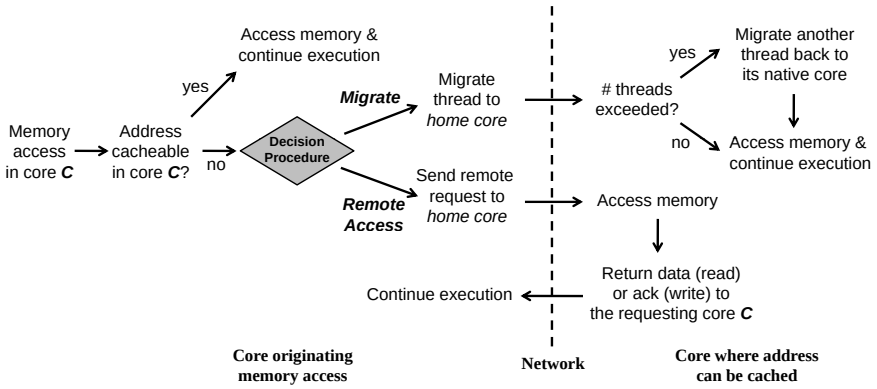


Figure 2: In our proof-of-concept implementation, shared memory leverages a combination of remote access and thread migration. Memory accesses to addresses not assigned to the local core can either result in a remote cache access or cause the execution context to be migrated to the relevant core.

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.[3] 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 Architecture

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; we are currently awaiting silicon samples.

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

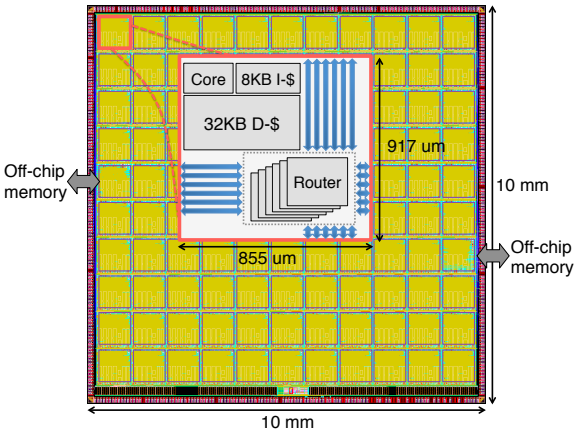


Figure 3: The 110-core EM² chip layout

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 external DRAM requests/responses; in each case, two networks are required to ensure deadlock-free operation [8].

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 [17].

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

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

read hits and two-cycle write hits. The first 86% $\left(=\frac{110}{128}\right)$ of 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; a further 14% of the address range is cacheable by any tile but without any hardware-level coherence guarantees. 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

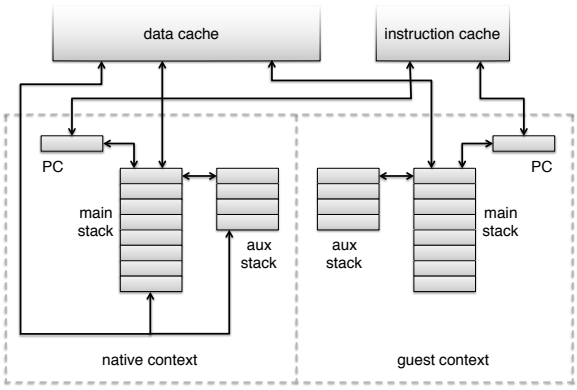


Figure 4: The processor core consists of two contexts in an SMT configuration, each of which comprises two stacks and a program counter, while the cache ports, migration network ports (not shown), and the migration predictor (not shown) are shared between the contexts. Stacks of the native context are 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, therefore, 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 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

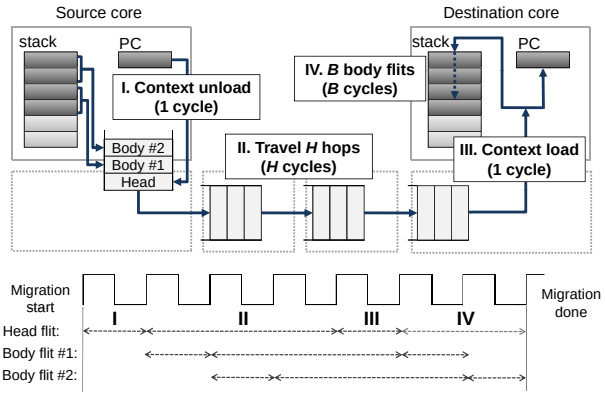


Figure 5: Hardware-level thread migration via the on-chip interconnect. Only the main stack is shown for simplicity.

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 in the native context, the core automatically spills or refills the stack from the data cache; in a sense, the main and auxiliary stacks serve as caches for larger stacks stored in memory. In a guest context, on the other hand, stacks are not backed by memory (cf. Figure 4); stack overflow or underflow at a guest context, therefore, cause the thread to migrate back to its native context where the stacks can be spilled or refilled.

To ensure deadlock-free thread migration in all cases, the core contains two thread contexts that execute in SMT fashion, called a *native* context and a *guest* context. 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 [8]. 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, 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

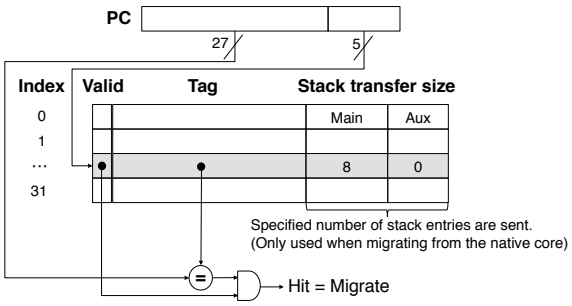


Figure 6: Each core has a PC-based migration predictor.

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).[4]

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* [23]—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 [23], our predictor further reduces on-chip network traffic by learning 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 core identified by *home*.

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 a specific 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.

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

[4] 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.

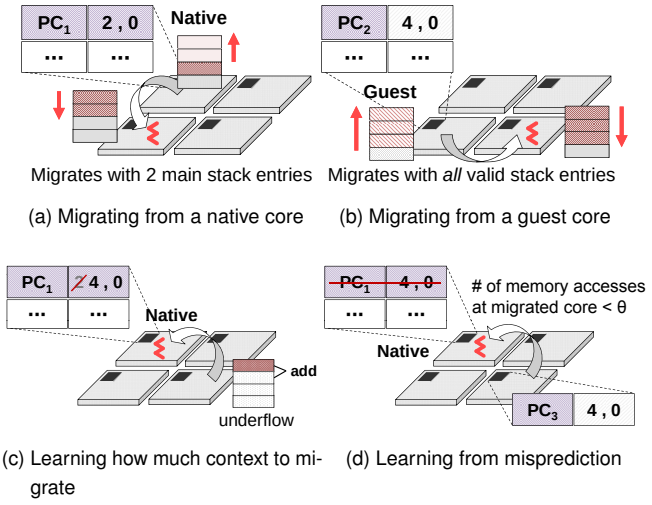


Figure 7: Decision/Learning mechanism of the migration predictor

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).[5] This allows the predictor to respond to runtime changes in program behavior.

3.6. The instruction set

This section outlines the custom stack ISA of the EM² cores.

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. Conceptually, the stacks are infinite and the entries that are not stored in the core are kept in memory; 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. The copying versions of the instructions make it easy to keep values like base addresses on the auxiliary stack.

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, which can be effectively leveraged to increase performance (see Section 5).

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 obtain the on-chip traffic levels and completion times for our architecture, we simulated the post-tapeout RTL of our chip, slightly altered to remove such ASIC-specific features as scan chains and modules used to collect various statistics at runtime. For the EM² results, the hardware migration predictor was used to automatically migrate the threads as dictated by their memory access patterns; for the baseline RA version, we replaced all memory instructions with the corresponding RA-only versions (such as `ld_ra` and `st_ra`) to prevent any migrations.

[5] 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.

For an apples-to-apples comparison, the guest and native contexts in the EM² core were not permitted to execute simultaneously: in each cycle, the instruction cache supplied either the native context or the guest context but not both. To focus on on-chip communication, we pre-initialized all caches.[6] All of the simulations used the entire 110-core chip; for each microbenchmark, 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).

4.2. Area and power estimates

Area and power estimates for the baseline and EM² versions were obtained by synthesizing the RTL using Synopsys Design Compiler (DC). For the EM² version, we used the post-tapeout RTL with the scan-chains and statistics modules deleted; for the RA-only version, we further deleted migration-specific features (the core's guest context, the migration predictor, and the migration and eviction networks). Both RTL versions passed the same tests as the tapeout RTL (save for any tests requiring migration for the RA-only version). 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.

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.

Dynamic power dominates the power signature, but is highly dependent on the specific benchmark, and obtaining accurate estimates for all of our benchmark is not practical. Instead, we observe that for the purposes of comparing EM² to the baseline architecture, it suffices to focus on the differences, which consist of (a) the additional core context, (b) the migration predictor, and (c) differences in cache and network accesses. The first two are insignificant: our implementation allowed only one of the EM² core contexts to be active in any given cycle, so even though the extra contexts adds leakage, dynamic power remains constant. The migration predictor is a small part of the tile and does not add much dynamic power (for reference, DC estimated that the predictor accounts for $< 4.5\%$ of the tile's dynamic power). Since we ran the same programs and pre-initialized caches, the cache accesses were the same, meaning equal contribution to dynamic power. The only significant difference is in the dynamic network power, which is directly proportional to the on-chip network traffic (i.e., the number of network flits sent times the distance traveled by each flit); we therefore report this for all benchmarks as a proxy for dynamic power.

[6] Although in general I$ contents can differ between RA and EM², this is not the case for our evaluation because all of our threads run the same code (as in most multithreaded applications), so pre-initializing the instruction caches treats RA and EM² equally.

To give an idea of how these costs compare against that of a well-understood architecture, we also estimated the area and leakage power of a 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%);[7] 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 the RA-only variant 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).

5. Evaluation

5.1. Microbenchmarks

Our benchmarks include: (1) partial table scan (*t-scan*), which executes queries that scan through a part of a globally shared data table distributed among the cache shards; (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; (4) merge sort (*merge-sort*), a classical sorting algorithm; and (5) parallel prefix sum (*pp-sum*), which computes partial sums of a vector.

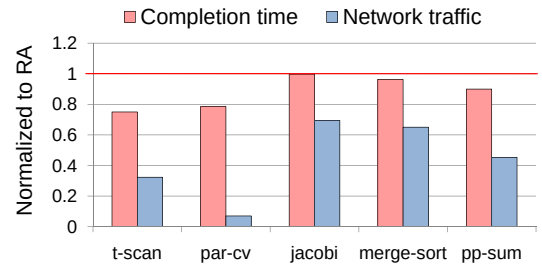
T-scan, *par-cv*, and *pp-sum* employ 110 threads (i.e., one thread for each core in the EM² chip). For convenience, *jacobi* and *merge-sort* use 72 and 64 threads, respectively; because those benchmarks do not exhibit any core contention, adding more threads has no negative impact on performance. Since the focus of this paper is on-chip memory performance and data movement, the inputs for all benchmarks are sized to fit in a shared aggregate on-chip cache. The migration predictor depth threshold θ is set to 3, which turns contiguous sequences of three or more memory accesses to the same core into migrations followed by local accesses.

5.2. Performance and migration rates

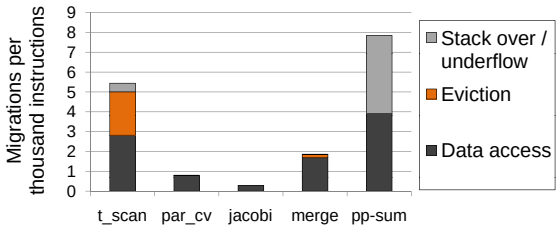
Figure 8a illustrates the improvements in the overall performance (i.e., completion time) and on-chip network traffic that the EM² architecture allows over the RA baseline. For our set of benchmarks, EM² never does worse than the baseline, and offers up to 25% reduction in run time. Throughout our benchmark suite, EM² offers significant reductions in on-chip network traffic, up to 14× less traffic for *par-cv*.

Migration rates, shown in Figure 8b, vary from 0.3 to 7.8 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 for every hundred instructions, taking 100+ cycles to move a thread to a different core would incur a prohibitive performance impact.

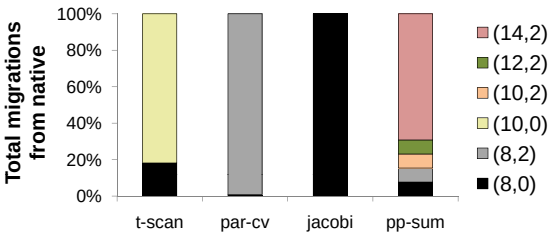
[7] Note that because multiple data caches can "gang up" on the same directory slice, the 100% version does not guarantee freedom from directory-capacity invalidations.



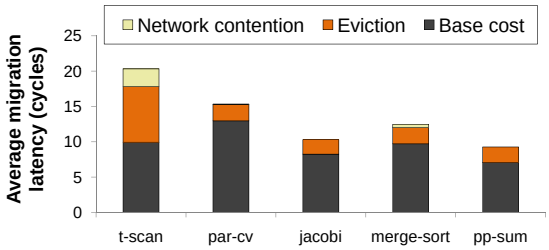
(a) EM² performance and network traffic normalized to RA



(b) The number of migrations per thousand instructions

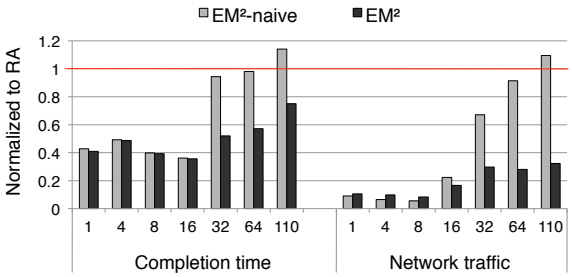


(c) The number of stack entries (main, aux) sent by the predictor for migrations from the native core. The result for *merge-sort* is not shown here because it uses explicit EM instructions instead of the predictor (details described in Section 5.3).

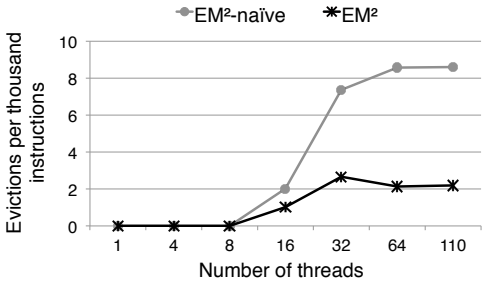


(d) Average migration latency and its breakdown in cycles

Figure 8: The evaluation of EM²



(a) Performance and network traffic with different number of threads



(b) The number of thread evictions per thousand instructions

Figure 9: The effect of core contention for *t-scan* under EM²

The number of migrations that occur because of stack underflows or overflows in a guest context are negligible for all benchmarks except *pp-sum*. This is because the migration predictor quickly learns the number of stack entries that should be transferred with each migration (see Figure 8c). In the case of *pp-sum*, the benchmark consumes more of the main stack than its 16-entry capacity, and therefore each thread must often return to its native core to refill the stack (even though, as Figure 8c shows, the predictor correctly learns to make its maximum prediction of 14 entries). In this case, a larger hardware stack would be needed to reduce the migration rate.

Figure 8d shows the actual average latency of all migrations for our benchmarks; in addition to the base cost which is statically determined by the distance and the migrating context size, migrations can take longer if a thread needs to evict a running guest thread, or if the network is congested. Considering all these factors, we observe that end-to-end migration takes about 10 to 20 cycles on average, demonstrating the efficient thread migration mechanism of EM².

5.3. Benchmark performance details

Partial table scan. In this benchmark, queries are assigned to threads and the table that is searched is distributed in equal chunks among the per-tile D$ shards. Because each query can scan through any data chunk at random, multiple threads may contend for the same core. This *core contention* causes threads to evict each other, resulting in ping-pong effect where a thread repeatedly migrates to access data only to be evicted and return to its native core. In a naïve implementation, this effect grows with the number of concurrent threads, eventually canceling out the performance gained by exploiting data locality (see Figure 9, EM²-naïve).

In order to mitigate this ping-pong effect, EM² can be configured to allow a guest thread to execute a set number of

instructions N before being evicted (see Figure 10a). This can allow more memory instructions to complete before an eviction occurs, greatly reducing the eviction rate: for the *t-scan* benchmark, setting $N = 10$ (a sequence containing four memory instructions) improved both performance and network traffic dramatically (Figure 9, EM²).

Parallel K-fold cross validation. In the common *k-fold cross-validation* technique, the data samples are split into k disjoint chunks and used to run k independent experiments: for each experiment $k-1$ chunks are used for training and the remaining chunk for testing, and the accuracy results are averaged. Since the experiments are independent, they can be naturally mapped 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 depends on learning from previous data. 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.

These characteristics make *par-cv* one of the benchmarks that benefits most from introducing thread migration (a 21% improvement in completion time and a 14× reduction in on-chip traffic). Efficient inter-thread synchronization was a key optimization in this benchmark: without it, when a thread has finished processing a data chunk and moves on to the next one before its predecessor has finished, the two threads will keep evicting each other while their current working sets overlap (see Figure 10b). To combat this, we let a thread to spin locally at its native core until the preceding thread informs it that it has finished its work; since a thread never gets evicted from its native context, this spin-at-native-core synchronization effectively eliminates the ping-pong evictions. Since the RA-only baseline does not require this optimization (indeed, it only degrades RA performance by 11%), we used the faster-performing unsynchronized version for the RA runs; nevertheless, the synchronized EM² version still outperforms RA by a significant margin.

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. In the naïve form, the local elements are computed one by one, and all of the memory accesses to remote cores become one-off accesses; in this case, the predictor never instructs threads to migrate and EM² performs the same as the RA-only baseline. The key optimization here is loop unrolling. After this transformation, multiple remote loads are now performed contiguously, meaning that a thread migrates with addresses for several loads, and migrates back with its stack filled with multiple load results (see Figure 10c). While unrolling does not change the performance under the RA regime, EM² uses migration to take advantage of this locality, and incurs 31% less network traffic than RA.

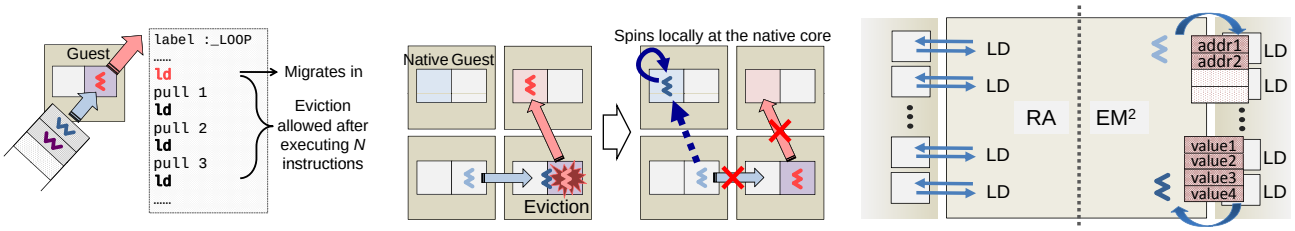
Merge sort. A typical implementation of parallel merge sort merges two sorted subarrays iteratively in a bottom-up manner. During the merge step, a thread reads two elements from two different subarrays, compares the two, and stores the smaller in the appropriate place in the output array; this output array then becomes one of the inputs for the next-level merge. While merging results in remote accesses as a thread traverses through the subarrays, these tend to be scattered one-off accesses rather than contiguous bursts; in this paradigm, the migration predictor never learns to migrate, and the EM² will show no improvement over RA.

A closer examination, however, reveals that significant improvement is nevertheless possible. Memory accesses (two reads and one store for a single element merge) are interleaved, but the home core of the output array changes only after making multiple writes on the same array chunk (i.e., same core), so running the thread on the core where the current output array chunk is located will reduce overall traffic. Although this pattern is too subtle for the predictor to detect, it is easily expressed at program level by changing all stores to their migration-only versions (cf. Section 3.6): the store will be local while the current region of the output array is being accessed but will automatically migrate as soon as the write pointer moves into the region cached in a different core. With this optimization, EM² has a 4% shorter runtime and 35% less on-chip traffic than the RA version.

Parallel prefix sum. This benchmark computes an additive reduction of an integer array: given an array $A = \{a_0, a_1, \ldots, a_n\}$, the result is a new array $B = \{\sum_{i=0}^{n} A_i, \sum_{i=1}^{n} A_i, \ldots, \sum_{i=n}^{n} A_i\}$. On machines with multiple concurrent threads of execution, parallel prefix operations are efficiently implemented using a series of strided vector operations: $B_i = \sum_{i}^{n} A_i = a_i + a_{i+1} + \cdots + a_n$, where a single additive term of each B_i is computed in parallel using a vectorized addition. Although each element of the result vector combines terms mapped to different cores, these terms are grouped so that $a_i, \ldots, a_j$ are mapped to the same core for some range $[i, j]$. EM² automatically detects and leverages this locality, in effect implementing vector loads and stores by migrating and executing batches of memory accesses locally; this reduces runtime by 10% and the network traffic by 55%.

	RA	EM²	CC
extra execution context in the core	no	yes	no
migration predictor logic & storage	no	yes	no
remote cache access support in the D$	yes	yes	no
coherence protocol logic in the D$	no	no	yes
coherence directory logic & storage	no	no	yes
number of independent on-chip networks	4	6	5

Table 1: A summary of architectural differences in the baseline RA-only version, EM², and a CC implementation.



(a) *T-scan*: EM² guarantees that a thread can execute *N* instructions before being evicted (*N* is configurable).

(b) *Par-cv*: thread evictions can be efficiently reduced with spin-at-native-core synchronization.

(c) *Jacobi*: EM² allows efficient bulk loads from a remote core.

Figure 10: Hardware/software collaboration to improve performance and energy under EM²

5.4. Area and power costs

Table 1 summarizes the architectural components required to add hardware-level thread migration capability to a remote-cache-access design. EM² requires an extra architectural context (for the guest thread) and two extra on-chip networks (for migrations and evictions) to avoid migration protocol deadlock. 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 contrast, a deadlock-free implementation of a directory-based coherence protocol such as MESI would replace the two on-chip networks of RA (for remote-cache-access requests and responses) 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.

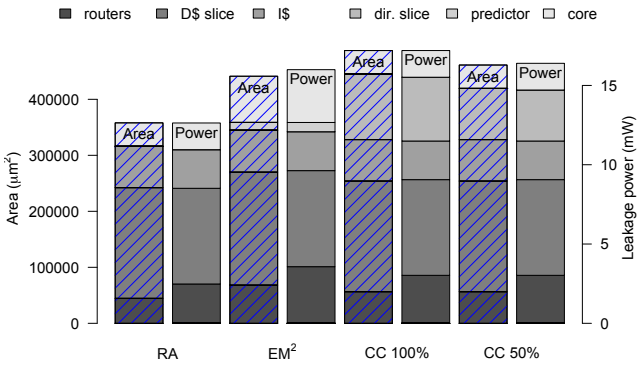


Figure 11: Relative area and leakage power costs of execution migration (EM²) versus a remote-access-only baseline (RA) and estimates for an exact-sharer cache coherent design (CC) with the directory sized to 100% and 50% of the data cache entries (Synopsys DC Ultra, IBM 45nm SOI high-voltage-threshold library, 800MHz).

Figure 11 shows the area and leakage power overheads of EM² and a MESI implementation (CC) with two directory sizes over a RA-only baseline. 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 two extra routers of EM² combined to add 23% more area and 27% more leakage to the cost of the RA baseline, with most of the overhead accounted for by the extra thread context. This compares favorably to our estimate of the directory-coherence alternatives—the CC 100% configuration added 36% area and 36% leakage power, and the CC 50% variant 29% area and 30% leakage over RA—and leads us to believe that the area and power overheads of implementing fine-grained thread migration in hardware are well within the acceptable range.

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 [10], and improving memory access latency via migration has been proposed using coarse-grained compiler transformations [12]. 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 [19]. Computation spreading [5] 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 [4]. Core salvaging [20] 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 [21] 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., lock acquisition) [14]. The very fine-grained nature of the migrations contemplated in many of these proposals—for example, a thread must be able to migrate immediately if its next instruction cannot be executed on the current core because of hardware faults [20]—requires support for fast, fine-grained migration with decentralized control, where the decision to migrate can be made autonomously by each thread; the design of such a system has not, however, been described and evaluated in detail, and is among the

contributions of this paper.

Our baseline remote-access-only (RA) architecture has its roots in the non-uniform memory architecture (NUMA) paradigm as extended to single-die caches (NUCA [15,7]). Migration and replication (of data rather than threads) was used to speed up NUMA systems (e.g., [26]), 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 [2,13] 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 [9,11]; CoG [1] moves pages to the "center of gravity" to improve data placement, while the O^2 scheduler [3] improves memory performance in distributed-memory multicores by trying to keep threads near their data during OS scheduling. Victim replication [27] 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 [11]. Other schemes add hardware support for page migration support [6,24], 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 [22]. 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 [18]. 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 [8]. 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 RA architectures. The migration predictor we present here is based on that of [23]; unlike the full-context predictor described there, however, our predictor supports stack-based partial context migration by learning how much of the thread context to migrate.

7. Limitations and Ongoing work

Read-only sharing. While the test-chip implementation we present in this paper allows read-only sharing in the 14% of memory region that is not assigned to any specific core, there is no hardware-level coherence guarantee for this region. Although this approach is acceptable for many applications, it places the burden of ensuring consistency on the software and makes programming more difficult. We expect to address alternatives, such as a design that combines cache coherence at the L1 level with fast migration to access the upper levels of cache, in future work.

Other uses of migration. In this paper we have focused on using fine-grained migration to accelerate data access; while this places strict requirements on the migration substrate and therefore makes an excellent "stress test," we view fine-grained migration as an enabling technology suitable for many applications. An architecture that supports fast migration makes it easy to build heterogeneous multicores: for example, if some of them have an FPU or a vector unit and others don't, threads can transparently migrate to the "heavy" cores to perform the required computations without the need for special-purpose protocols (in such a design, the architectural overhead of a guest context could be limited to the "heavy" cores). Investigating the possible applications of fine-grained migration is a focus of our ongoing work.

References

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

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

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

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

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

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

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

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

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

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

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

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

[13] 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.

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

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

[16] J. S. Kim, M. B. Taylor, J. Miller, and D. Wentzlaff, "Energy Characterization of a Tiled Architecture Processor with On-Chip Networks," in *ISLPED*, 2003.

[17] 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.

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

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

[20] 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.

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

[22] 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.

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

[24] 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.

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

[26] 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.

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