Deadlock-Free Fine-Grained Thread Migration

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

***Abstract*—Several recent studies have proposed fine-grained, hardware-level thread migration in multicores as a solution to power, reliability, and memory coherence problems. The need for fast thread migration has been well documented, however, a fast, deadlock-free migration protocol is sorely lacking: existing solutions either deadlock or are too slow and cumbersome to ensure performance with frequent, fine-grained thread migrations.**

In this study, we introduce the Exclusive Native Context (ENC) protocol, a general, provably deadlock-free migration protocol for instruction-level thread migration architectures. Simple to implement, ENC does not require additional hardware beyond common migration-based architectures. Our evaluation using synthetic migrations and the SPLASH-2 application suite shows that ENC offers performance within 11.7% of an idealized deadlock-free migration protocol with infinite resources.

I. Introduction

In SMP multiprocessor systems and multicore processors, process and thread migration has long been employed to provide load and thermal balancing among the processor cores. Typically, migration is a direct consequence of thread scheduling and is performed by the operating system (OS) at timeslice granularity; although this approach works well for achieving long-term goals like load balancing, the relatively long periods, expensive OS overheads, and high communication costs have generally rendered fast thread migration impractical [16].

Recently, however, several proposals with various aims have centered on thread migration too fine-grained to be effectively handled via the OS. In the design-for-power domain, rapid thread migration among cores in different voltage/frequency domains has allowed less demanding computation phases to execute on slower cores to improve overall power/performance ratios [12]; in the area of reliability, migrating threads among cores has allowed salvaging of cores which cannot execute some instructions because of manufacturing faults [11]; finally, fast instruction-level thread migration has been used in lieu of coherence protocols or remote accesses to provide memory coherence among per-core caches [4] [5]. The very fine-grained nature of the migrations contemplated in these proposals—a thread must be able to migrate immediately if its next instruction cannot be executed on the current core because of hardware faults [11] or to access data cached in another core [4]—demands fast, hardware-level migration systems with decentralized control, where the decision to migrate can be made *autonomously* by each thread.

The design of an efficient fine-grained thread migration protocol has not, however, been addressed in detail. The foremost concern is avoiding deadlock: if a thread context can be blocked by other contexts during migration, there is an additional resource dependency in the system which may cause the system to deadlock. But most studies do not even discuss this possibility: they implicitly rely on expensive, centralized migration protocols to provide deadlock freedom, with overheads that preclude frequent migrations [3], [10], or limit migrations to a core's local neighborhood [14]. Some fine-grain thread migration architectures simply give up on deadlock avoidance and rely on expensive recovery mechanisms (e.g., [8]).

With this in mind, we introduce a novel thread migration protocol called Exclusive Native Context (ENC). To the best of our knowledge, ENC is the first on-chip network solution to guarantee freedom from deadlock for general fine-grain thread migration without requiring handshaking. Our scheme is simple to implement and does not require any hardware beyond that required for hardware-level migrations; at the same time, it decouples the performance considerations of on-chip network designs from deadlock analysis, freeing architects to consider a wide range of on-chip network designs.

In the remainder of this paper,

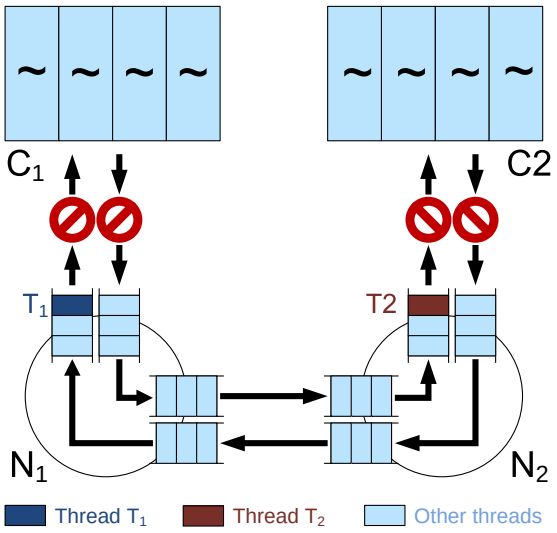
- we present ENC, a novel deadlock-free fine-grained thread migration protocol;
- we show how deadlock arises in other migration schemes, and argue that ENC is deadlock-free;
- we show that ENC performance on SPLASH-2 application benchmarks [17] running under a thread-migration architecture [4] is on par with an idealized deadlock-free migration scheme that relies on infinite resources.

II. Deadlock in Thread Migration

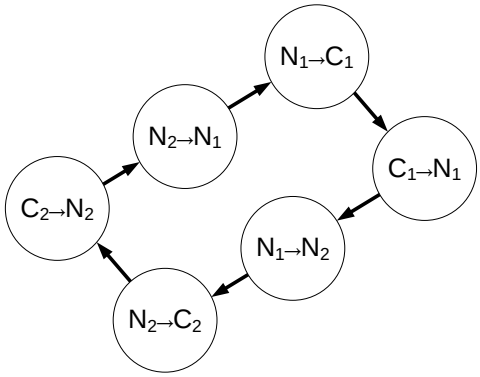
A. Protocol-level deadlock

Most studies on on-chip networks focus on the network itself and assume that a network packet *dies* soon after it reaches its destination core—for example, the result of a memory load request might simply be written to its destination register. This assumption simplifies deadlock analysis because the dead packet no longer holds any resources that might be needed by other packets, and only *live* packets are involved in deadlock scenarios.

With thread migration, however, the packet carries an execution context, which moves to an execution unit in the core and *occupies* it until it migrates again to a different core. Thus, unless migrations are centrally scheduled so that the migrating context always finds available space at its destination, execution contexts occupying a core can block contexts arriving over the network, creating additional deadlock conditions that conventional on-chip network deadlock analysis does not consider.



(a) Network Buffers in a Protocol-level Deadlock



(b) Channel Dependency Graph

Fig. 1. Protocol-level Deadlock of Fine-grain, Autonomous Thread Migration

For example, suppose a migrating thread T_1 in Figure 1(a) is heading to core C_1. Although T_1 arrives at routing node N_1 directly attached to C_1, all the execution units of C_1 are occupied by other threads ($\sim$), and one of them must migrate to another core for T_1 to make progress. But at the same time, thread T_2 has the same problem at core C_2, so the contexts queued behind T_2 are backed up all the way to C_1 and prevent a C_1 thread from leaving. So T_1 cannot make progress, and the contexts queued behind it have backed up all the way to C_2, preventing any of C_2's threads from leaving, and completing the deadlock cycle. Figure 1(b) illustrates this deadlock using a channel dependency graph (CDG) [2] where nodes correspond to channels of the on-chip network and edges to dependencies associated with making progress on the network.

We call this type of deadlock a *protocol-level deadlock*, because it is caused by the migration protocol itself rather than the network routing scheme. Previous studies involving rapid thread migration typically either do not discuss protocol-level deadlock, implicitly relying on a centralized deadlock-free migration scheduler [3], [10], [14], using deadlock detection and recovery [8], employing a cache coherence protocol to migrate contexts via the cache and memory hierarchy, effectively providing a very large buffer to store contexts [12], or employing slow handshake-based context swaps [11]. All of these approaches have substantial overheads, motivating the development of an efficient network-level deadlock-free migration protocol.

B. Evaluation with synthetic migration benchmarks

As a non-deadlock-free migration protocol, we consider the naturally arising SWAP scheme, implicitly assumed by several works: whenever a migrating thread T_1 arrives at a core, it evicts the thread T_2 currently executing there and sends it back to the core where T_1 originated. Although intuitively one might expect that this scheme should not deadlock because T_2 can be evicted into the slot that T_1 came from, this slot is not reserved for T_2 and another thread might migrate there faster, preempting T_2; it is therefore not guaranteed that T_2 will exit the network and deadlock may arise. (Although adding a handshake protocol with extra buffering can make SWAP deadlock-free [11], the resulting scheme is too slow for systems which require frequent migrations).

In order to examine how often the migration system might deadlock in practice, we used a synthetic migration benchmark where each thread keeps migrating between the initial core where it was spawned and a *hotspot* core. (Since migration typically occurs to access some resource at a core, be it a functional unit or a set of memory locations, such hotspots naturally arise in multithreaded applications). We used varying numbers (one to four) of randomly assigned hotspots, and 64 randomly located threads that made a thousand migrations to destinations randomly chosen among their originating core and the various hotspots every 100 cycles. To stress the migartion framework as in a fine-grain migration system, we chose the migration interval of 100 cycles. We used the cycle-level network-on-chip simulator DARSIM [6], suitably modified with a migration system, to model a 64-core system connected by a 2D mesh interconnect. Each on-chip network router had enough network buffers to hold 4 thread contexts on each link with either 2 or 4 virtual channels; we also examined the case where each core has a context queue to hold arriving thread contexts when there are no available execution units. We assumed Intel Atom-like x86 cores with execution contexts of 2 Kbits [12] and enough network bandwidth to fit each context in four or eight flits. Table I summarizes the simulation setup.

Figure 2 shows the percentage of runs (out of the 100) that end with deadlock under the SWAP scheme. Without an additional context queue, nearly all experiments end in deadlock. Further, even though context buffering can *reduce* deadlock, deadlock still occurs at a significant rate for the tested configurations.

The synthetic benchmark results also illustrate that susceptibility to deadlock depends on migration patterns: when there is only one hotspot, the migration patterns across threads are usually not cyclic because each thread just moves back and forth between its own private core and only one shared core;

Core and Migration	
Core architecture	single-issue, two-way multithreading
The size of a thread context (relative to the size of network flit)	4 flits
Number of threads	64
Number of hotspots	1, 2, 3 and 4
Migration interval	100 cycles
On-chip Network	
Network topology	8-by-8 mesh
Routing algorithms	Dimension-order wormhole routing
Number of virtual channels	2 and 4
The size of network buffer (relative to the size of context)	4 per link or 20 per node
The size of context queue (relative to the size of context)	0, 4 and 8 per core

TABLE I
THE SIMULATION SETUP FOR SYNTHETIC MIGRATION BENCHMARKS.

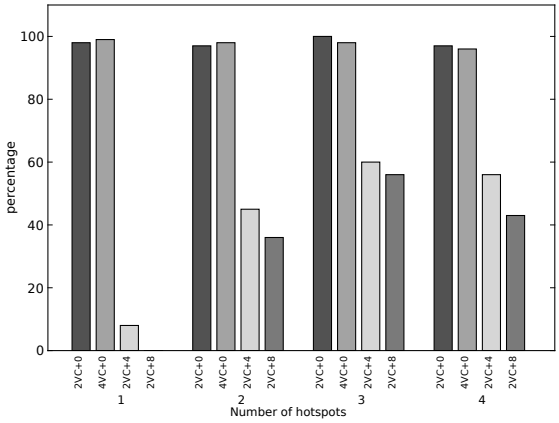


Fig. 2. Deadlock scenarios with synthetic sequences of migrations. 2VC+4 for example corresponds to 2 virtual channels and a context queue of 4 contexts.

when there are two or more hotspots and threads have more destinations, on the other hand, their paths intersect in more complex ways, making the system more prone to deadlock. Although small context buffers prevent deadlock with some migration patterns, they do not ensure deadlock avoidance because there are still a few deadlock cases.

III. Exclusive Native Context Protocol

ENC takes a network-based approach to provide deadlock freedom. Unlike coarse-grain migration protocols, ENC allows autonomous thread migrations. To enable this, the new thread context may evict one of the thread contexts executing in the destination core, and ENC provides the evicted thread context a *safe path* to another core on which it will never be blocked by other threads that are also in transit.

To provide the all-important safe path for evicted threads, ENC uses a set of policies in core scheduling, routing, and virtual channel allocation.

Each thread is set as a *native context* of one particular core, which reserves a register file (and other associated context

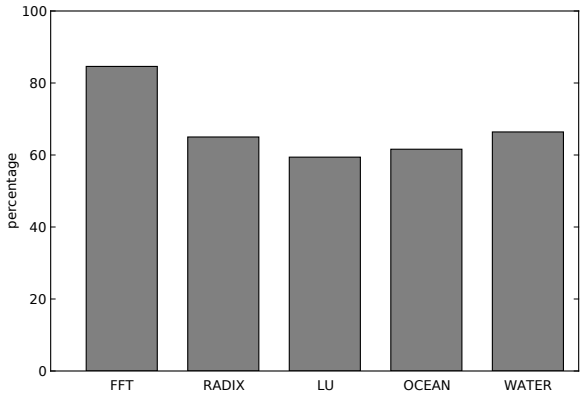


Fig. 4. The percentage of accesses to a threads *native* core (i.e., the core where it started and that holds its stack) in various SPLASH-2 benchmarks.

state) for the thread. Other threads cannot use the reserved resource even if it is not being used by the native context. Therefore, a thread will always find an available resource every time it arrives at the core where the thread is a native context. We will refer to this core as the thread's *native core*.

Dedicating resources to native contexts requires some rudimentary multithreading support in the cores. If a thread may migrate to an arbitrary core which may have a different thread as its native context, the core needs to have an additional register file (i.e., a *guest context*) to accept a non-native thread because the first register file is only available to the *native* context. Additionally, if a core has multiple native contexts, there must be enough resources to hold all of its native contexts simultaneously so no native thread is blocked by other native threads. It is a reasonable assumption that an efficient fine-grain, migration-based architecture will require some level of multithreading, in order to prevent performance degradation when multiple threads compete for the resources of the same core.

If an arriving thread is not a native context of the core, it may be temporarily blocked by other non-native threads currently on the same core. The new thread evicts one of the executing non-native threads and takes the released resource. We repeat that a thread never evicts a native context of the destination core because the resource is usable only by the native context. To prevent livelock, however, a thread is not evicted unless it has executed at least one instruction since it arrived at the current core. That is, an existing thread will be evicted by a new thread only if it has made some progress in its current visit on the core. This is why we say the arriving thread may be temporarily blocked by other non-native threads.

Where should the native core be? In the first-touch data placement policy [7] we assume here, each thread's stack and private data are assigned to be cached in the core where the thread originates. We reasoned, therefore, that most accesses made by a thread will be to its originating core (indeed,

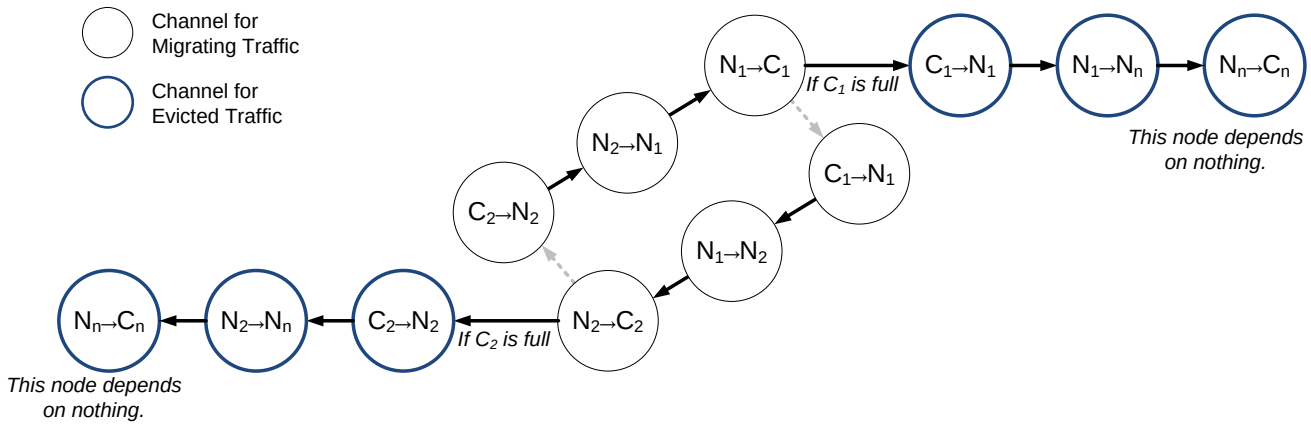


Fig. 3. Acyclic Channel Dependency Graph of ENC

Figure 4 shows that in the SPLASH-2 benchmarks we used, about 60%–85% of a thread's accesses are to its native core). We therefore select each thread's originating core as its native core.

In what follows, we first describe a basic, straightforward version of ENC, which we term ENC0, and then describe a better-performing optimized version.

A. The basic ENC algorithm (ENC0)

Whenever a thread needs to move from a non-native core to a destination core, ENC0 first sends the thread to its native core which has a dedicated resource for the thread. If the destination core is not the native core, the thread will then move from its native core to the destination core. Therefore, from a network standpoint, a thread movement either ends at its native core or begins from its native core. Since a thread arriving at its native core is guaranteed to be unloaded from the network, any migration is fully unloaded (and therefore momentarily occupies no network resources) somewhere along its path.

To keep the migrations deadlock-free, however, we must also ensure that movements destined for a native core actually get there without being blocked by any other movements; otherwise the native-core movements might never arrive and be unloaded from the network. The most straightforward way of ensuring this is to use two sets of virtual channels, one for to-native-core traffic and the other for from-native-core traffic. If the baseline routing algorithm requires only one virtual channel to prevent network-level deadlock like dimension-order routing, ENC0 requires a minimum of two virtual channels per link to provide protocol-level deadlock avoidance. Note that ENC0 may work with any baseline routing algorithm for a given source-destination pair, such as Valiant [15] or O1TURN [13], both of which require two virtual channels to avoid deadlock. In this case, ENC0 will require four virtual channels.

B. The full ENC algorithm

Although ENC0 is simple and straightforward, it suffers the potential overhead of introducing an intermediate destination for each thread migration: if thread T wishes to move from core A to B, it must first go to N, the native core for T. In some cases, this overhead might be significant: if A and B are close to each other, and N is far away, the move may take much longer than if it had been a direct move.

To reduce this overhead, we can augment the ENC0 algorithm by distinguishing *migrating* traffic and *evicted* traffic: the former consists of threads that wish to migrate on their own because, for example, they wish to access resources in a remote core, while the latter corresponds to the threads that are evicted from a core by another arriving thread.

Whenever a thread is *evicted*, ENC, like ENC0, sends the thread to its native core, which is guaranteed to accept the thread. We will not therefore have a chain of evictions: even if the evicted thread wishes to go to a different core to make progress (e.g., return to the core it was evicted from), it must first visit its native core, get unloaded from the network, and then move again to its desired destination. Unlike ENC0, however, whenever a thread migrates on its own accord, it may go directly to its destination without visiting the home core. (Like ENC0, ENC must guarantee that evicted traffic is never blocked by migrating traffic; as before, this requires two sets of virtual channels).

Based on these policies, the ENC migration algorithm can be described as follows. Note that network packets always travel within the same set of virtual channels.

1) If a native context has arrived and is waiting on the network, move it to a reserved register file and proceed to Step 3.
2) a) If a non-native context is waiting on the network and there is an available register file for non-native contexts, move the context to the register file and proceed to Step 3.
   b) If a non-native context is waiting on the network and all the register files for non-native contexts are full, choose one among the threads that have fin-

ished executing an instruction on the core[1] *and* the threads that want to migrate to other cores. Send the chosen thread to its *native* core on the virtual channel set for evicted traffic. Then, advance to the next cycle. (No need for Step 3).

3) Among the threads that want to migrate to other cores, choose one and send it to the desired destination on the virtual channel set for migrating traffic. Then, advance to the next cycle.

This algorithm effectively breaks the cycle of dependency of migrating traffic and evicted traffic. Figure 3 illustrates how ENC breaks the cyclic dependency shown in Figure 1(b), where C_n denotes the native core of the evicted thread, and N_n its attached router node.

There is a subtlety when a migrating context consists of multiple flits and the core cannot send out an entire context all at once. For example, the core may find no incoming contexts at cycle 0 and start sending out an executing context T_1 to its desired destination, but before T_1 completely leaves the core, a new migrating context, T_2, arrives at the core and is blocked by the remaining flits of T_1. Because T_1 and T_2 are on the same set of virtual channels for migration traffic, a cycle of dependencies may cause a deadlock. To avoid this case, the core must inject migration traffic only if the whole context can be moved out from the execution unit so arriving contexts will not be blocked by incomplete migrations; this can easily be implemented by monitoring the available size of the first buffer on the network for migration traffic or by adding an additional outgoing buffer whose size is one context size.

Although both ENC0 and ENC are provably deadlock-free under deadlock-free routing because they eliminate all additional dependencies due to limited context space in cores, we confirmed that they are deadlock-free with the same synthetic benchmarks used in Section II-B. We also simulated an incomplete version of ENC that does *not* consider the aforementioned subtlety and sends out a migrating context if it is possible to push out its first flit. While ENC0 and ENC had no deadlocks, deadlocks occurred with the incomplete version because it does not provide a safe path for evicted traffic in the case when a migrating context is being sequentially injected to the network; this illustrates that fine-grained migration is very susceptible to deadlock and migration protocols need to be carefully designed.

IV. Performance Evaluation

A. Baseline protocols and simulated migration patterns

We compared the performance overhead of ENC0 and ENC to the baseline SWAP algorithm described in Section II-B. However, as SWAP can deadlock, in some cases the execution might not finish. Therefore, we also tested SWAPinf, a version of SWAP with an infinite context queue to store migrating thread contexts that arrive at the core; since an arriving context can always be stored in the context queue, SWAPinf never deadlocks. Although impractical to implement, SWAPinf provides a useful baseline for performance comparison. We compared SWAP and SWAPinf to ENC0 and ENC with two virtual channels. The handshake version of SWAP was deemed too slow to be a good baseline for performance comparison.

In order to see how ENC would perform with arbitrary migration patterns, we first used a random sequence of migrations in which each thread may migrate to any core at a fixed interval of 100 cycles. In addition, we also wished to evaluate real applications running under a fine-grained thread-migration architecture. Of the three such architectures described in Section I, we rejected core salvaging [11] and ThreadMotion [12] because the thread's migration patterns do not depend on the application itself but rather on external sources (core restrictions due to hard faults and the chip's thermal environment, respectively), and could conceivably be addressed with synthetic benchmarks. We therefore selected the EM^2 architecture [4], which migrates threads to a given core to access memory exclusively cached in that core; migrations in EM^2 depend intimately on the application's access patterns and are difficult to model using synthetic migration patterns.

We used the same simulation framework as described in Section II-B to examine how many cycles are spent on migrating thread contexts.

B. Network-Independent Traces (NITs)

While software simulation provides the most flexibility in the development of many-core architectures, it is severely constrained by simulation time. For this reason, common simulation methods do not faithfully simulate every detail of target systems, to achieve reasonably accurate results in an affordable time. For example, Graphite [9] provides very efficient simulation of a many-core system based on the x86 architecture. However, it has yet to provide faithful simulation of network buffers. Therefore, Graphite simulation does not model the performance degradation due to head-of-line blocking, and moreover, deadlock cannot be observed even if the application being simulated may actually end up in deadlock.

On the other hand, most on-chip network studies use a detailed simulator that accurately emulates the effect of network buffers. However, they use simple traffic generators rather than simulating actual cores in detail. The traffic generator often replays *network traces* captured from application profiling, in order to mimic the traffic pattern of real-world applications. It, however, fails to mimic complex dependency between operations, because most communication in many-core systems depends on the previous communication. For example, a core may need to first receive data from a producer, before it processes the data and sends it to a consumer. Obviously, if the data from the producer arrives later than in profiling due to network congestion, sending processed data to the consumer is also delayed. However, network traces typically only give the absolute time when packets are sent, so the core may send processed data to the consumer prior to it even receiving the data from its producer! In other words, the network-trace approach

[1] No instructions should be in flight.

fails to realistically evaluate application performance, because the timing of packet generation, which depends on on-chip network conditions, is assumed *before* the actual simulation of the network.

It is very important to reflect the behavior of network conditions, because it is critical not only for performance, but also to verify that network conditions don't cause deadlock. Therefore, we use DARSIM [6], a highly configurable, cycle-accurate on-chip network simulator. Instead of using network traces, however, we generate *network-independent traces* (NITs) from application profiling. Unlike standard application traces, NITs keep inter-thread dependency information and relative timings instead of absolute packet injection times; the dependencies and relative timings are replayed by an interpreter module added to the network simulator. By replacing absolute timestamps with dependencies and relative timings, NITs allow cores to "respond" to messages from other cores once they have arrived, and solve the consumer-before-producer problem that occurs with network traces.

The NITs we use for EM^2 migration traffic record memory instruction traces of all threads, which indicate the home core of each memory instruction and the number of cycles it takes to execute all non-memory instructions between two successive memory instructions. With these traces and the current location of threads, a simple interpreter can determine whether each memory instruction is accessing memory cached on the current core or on a remote core; on an access to memory cached in a remote core, the interpreter initiates a migration of the corresponding thread. After the thread arrives at the home core and spends the number of cycles specified in the traces for non-memory operations, the interpreter does the same check for the next memory instruction.

The interpreter does not, of course, behave exactly the same as a real core does. For one, it does not consider lock/barrier synchronization among threads; secondly, it ignores possible dependencies of the actual *memory addresses* accessed on network performance (consider, for example, a multithreaded work-queue implemented via message passing: the memory access patterns of the program will clearly depend on the order in which the various tasks arrive in the work queue, which in turn depends on network performance). Nevertheless, NITs allow the system to be simulated in a much more realistic way by using memory traces rather than network traces.

C. Simulation details

For the evaluation under arbitrary migration patterns, we used a synthetic sequence of migrations for each number of hotspots as in Section II-B. We also chose five applications from the SPLASH-2 benchmark suite to examine application-specific migration patterns, namely FFT, RADIX, LU (contiguous), WATER (nsquared), and OCEAN (contiguous), which we configured to spawn 64 threads in parallel. Then we ran those applications using Pin [1] and Graphite [9], to generate memory instruction traces. Using the traces and the interpreter as described in the previous section, we executed the sequences of memory instructions on DARSIM.

Protocols and Migration Patterns	
Migration Protocols	SWAP, SWAPinf ENC0 and ENC
Migration Patterns	a random migration sequence & 5 applications from SPLASH-2: FFT, RADIX, LU (contiguous) WATER (nsquared) OCEAN (contiguous)
Core	
Core architecture	single-issue, two-way multithreading EM^2
The size of a thread context (relative to the size of network flit)	4, 8 flits
Number of threads	64
On-chip Network	
Network topology	8-by-8 mesh
Routing algorithms	Dimension-order wormhole routing
Number of virtual channels	2
The size of network buffer (relative to the size of context)	4 per link (20 per node)
The size of context queue	∞ for SWAPinf, 0 otherwise

TABLE II
THE SIMULATION SETUP FOR SYNTHETIC SEQUENCES OF MIGRATIONS.

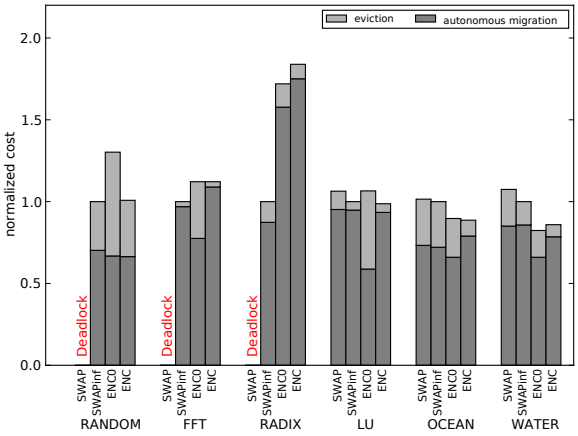


Fig. 5. Total migration cost (4 flits per context)

As in Section II-B, we first assumed the context size is 4 flits. However, we also used the context size of 8 flits, to examine how ENC's performance overhead would change if used with an on-chip network with less bandwidth, or a baseline architecture which has very large thread context size. The remaining simulation setup is similar to Section II-B. Table II summarizes the simulation setup used for the performance evaluation.

D. Simulation results

Figure 5 shows the total migration cost in each migration pattern normalized to the cost in SWAPinf when the context size is equivalent to four network flits. Total migration cost is the sum of the number of cycles that each thread spends between when it moves out of a core and when it enters another. First of all, the SWAP algorithm causes deadlock in FFT and RADIX, as well as in RANDOM, when each thread context migrates in 4 network flits. As we will see in Figure 8,

RANDOM	FFT	RADIX	LU	OCEAN	WATER
8	61	60	61	61	61

TABLE III
THE MAXIMUM SIZE OF CONTEXT QUEUES IN SWAPINF RELATIVE TO THE SIZE OF A THREAD CONTEXT

LU and OCEAN also end up with deadlock with the context size of 8 flits. Our results illustrate that real applications are also prone to deadlock if they are not supported by a deadlock-free migration protocol, as mentioned in Section II-B.

Deadlock does not occur when SWAPinf is used due to the infinite context queue. The maximum number of contexts at any moment in a context queue is smaller in RANDOM than in the application benchmarks because the random migration evenly distributes threads across the cores so there is no heavily congested core (cf. Table III). However, the maximum number of contexts is over 60 for all application benchmarks, which is more than 95% of all threads on the system. This discourages the use of context buffers to avoid deadlock.[2]

Despite the potential overhead of ENC described earlier in this section, both ENC and ENC0 have comparable performance, and are overall 11.7% and 15.5% worse than SWAPinf, respectively. Although ENC0 has relatively large overhead of 30% in total migration cost under the random migration pattern, ENC reduces the overhead to only 0.8%. Under application-specific migration patterns, the performance largely depends on the characteristics of the patterns; while ENC and ENC0 have significantly greater migration costs than SWAPinf under RADIX, they perform much more competitively in most applications, sometimes better as in applications such as WATER and OCEAN. This is because each thread in these applications mostly works on its private data; provided a thread's private data is assigned to its native core, the thread will mostly migrate to the native core (cf. Figure 4). Therefore, the native core is not only a safe place to move a context, but also the place where the context most likely makes progress. This is why ENC0 usually has less cost for autonomous migration, but higher eviction costs. Whenever a thread migrates, it needs to be "evicted" to its native core. After eviction, however, the thread need not migrate again if its native core was its migration destination.

The effect of the portion of native cores in total migration destinations can be seen in Figure 6, showing total migration distances in hop counts normalized to the SWAPinf case. When the destinations of most migrations are native cores, such as in FFT, ENC has not much different total migration distance from SWAPinf. When the ratio is lower, such as in LU, the migration distance for ENC is longer because it is more likely for a thread to migrate to non-native cores after it is evicted to its native core. This also explains why ENC has the most overhead in total migration distance under random migrations because the least number of migrations are going to native cores.

[2]Note that, however, the maximum size of context buffers from the simulation results is not a necessary condition, but a sufficient condition to prevent deadlock.

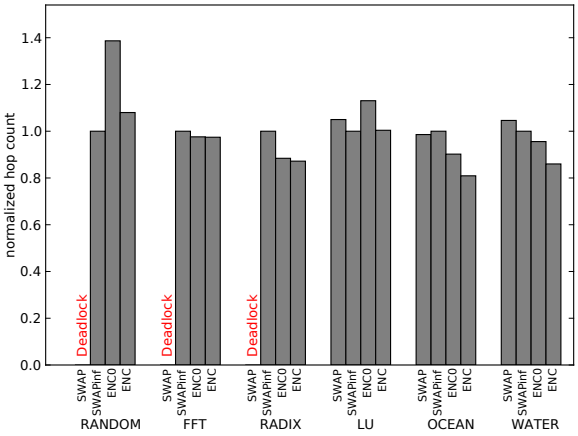


Fig. 6. Total migration distance in hop counts for various SPLASH-2 benchmarks.

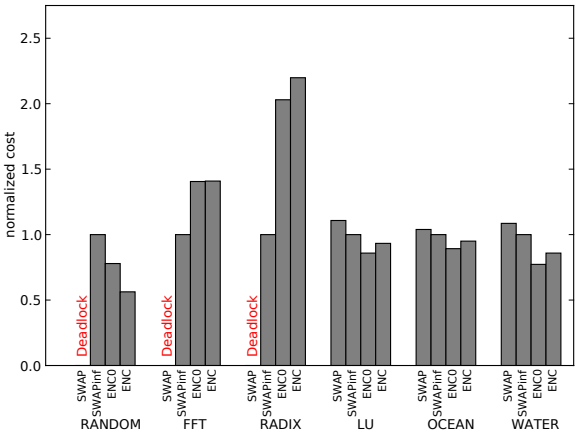


Fig. 7. Part of migration cost due to congestion

Even in the case where the destination of the migration is often not the native core of a migrating thread, ENC may have an overall migration cost similar to SWAPinf as shown in LU, because it is less affected by network congestion than SWAPinf. This is because ENC effectively distributes network traffic over the entire network, by sending out threads to their native cores. Figure 7 shows how many cycles are spent on migration due to congestion, normalized to the SWAPinf case. ENC and ENC0 have less congestion costs under RANDOM, LU, OCEAN, and WATER. This is analogous to the motivation behind the Valiant algorithm [15]. One very distinguishable exception is RADIX; while the migration distances of ENC/ENC0 are similar to SWAPinf because the native-core ratio is relatively high in RADIX, they are penalized to a greater degree by congestion than SWAPinf. This is because other applications either do not cause migrations as frequently as RADIX, or their migration traffic is well distributed because threads usually migrate to nearby destinations only.

If the baseline architecture has a large thread context or an

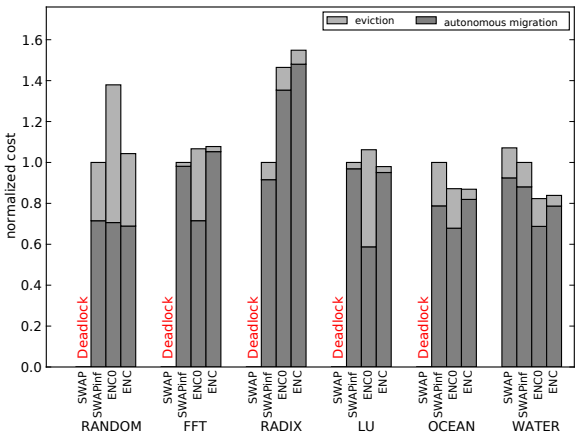


Fig. 8. Total migration cost (8 flits per context)

on-chip network has limited bandwidth to support thread migration, each context migrates in more network flits which may affect the network behavior. Figure 8 shows the total migration costs when a thread context is the size of eight network flits. As the number of flits for a single migration increases, the system sees more congestion. As a result, the migration costs increase by 39.2% across the migration patterns and migration protocols. While the relative performance of ENC/ENC0 to SWAPinf does not change much for most migration patterns, the increase in the total migration cost under RADIX is greater with SWAPinf than with ENC/ENC0 as the network becomes saturated with SWAPinf too. Consequently, the overall overhead of ENC and ENC0 with the context size of 8 flits is 6% and 11.1%, respectively. The trends shown in Figure 6 and Figure 7 also hold with the increased size of thread context.

V. Conclusions

ENC is a deadlock-free migration protocol for general fine-grain thread migration. Using ENC, threads can make autonomous decisions on when and where to migrate; a thread may just start traveling when it needs to migrate, without being scheduled by any global or local arbiter. Therefore, the migration cost is only due to the network latencies in moving thread contexts to destination cores, possibly via native cores.

Compared to a baseline SWAPinf protocol which assumes infinite queues, ENC has an average of 11.7% overhead for overall migration costs under various types of migration patterns. The performance overhead depends on migration patterns, and under most of the synthetic and application-specific migration patterns used in our evaluation ENC shows negligible overhead, or even performs better; although ENC may potentially increase the total distance that threads migrate by evicting threads to their native cores, it did not result in higher migration cost in many cases because evicted threads often need to go to the native core anyway, and intermediate destinations can reduce network congestion.

While the performance overhead of ENC remains low in most migration patterns, a baseline SWAP protocol actually ends up with deadlock, not only for synthetic migration sequences but also for real applications. Considering this, ENC is a very compelling mechanism for any architecture that exploits very fine-grain thread migrations and which cannot afford conventional, expensive migration protocols.

Finally, ENC is a flexible protocol that can work with various on-chip networks with different routing algorithms and virtual channel allocation schemes. One can imagine developing various ENC-based on-chip networks optimized for performance under a specific thread migration architecture.

References

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

[2] William J. Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2003.

[3] Wei Hu, Xingsheng Tang, Bin Xie, Tianzhou Chen, and Dazhou Wang. An efficient power-aware optimization for task scheduling on noc-based many-core system. In *Proceedings of CIT 2010*, pages 172–179, 2010.

[4] Omer Khan, Mieszko Lis, and Srinivas Devadas. EM^2: A Scalable Shared Memory Multi-Core Achitecture. In *CSAIL Technical Report MIT-CSAIL-TR-2010-030*, 2010.

[5] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Omer Khan, and Srinivas Devadas. Scalable directoryless shared memory coherence using execution migration. In *CSAIL Technical Report MIT-CSAIL-TR-2010-053*, 2010.

[6] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Pengju Ren, Omer Khan, and Srinivas Devadas. DARSIM: a parallel cycle-level NoC simulator. In *Proceedings of MoBS-6*, 2010.

[7] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. Scott. Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems. In *IPPS*, 1995.

[8] Steve Melvin, Mario Nemirovsky, Enric Musoll, and Jeff Huynh. A massively multithreaded packet processor. In *Proceedings of NP2: Workshop on Network Processors*, 2003.

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

[10] Matthew Misler and Natalie Enright Jerger. Moths: Mobile threads for on-chip networks. In *Proceedings of PACT 2010*, pages 541–542, 2010.

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

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

[13] Daeho Seo, Akif Ali, Won-Taek Lim, Nauman Rafique, and Mithuna Thottethodi. Near-Optimal Worst-Case Throughput Routing for Two-Dimensional Mesh Networks. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA 2005)*, pages 432–443, 2005.

[14] Kelly A. Shaw and William J. Dally. Migration in single chip multiprocessor. In *Computer Architecture Letters*, pages 12–12, 2002.

[15] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 263–277, 1981.

[16] Boris Weissman, Benedict Gomes, Jürgen W. Quittek, and Michael Holtkamp. Efficient fine-grain thread migration with active threads. In *Proceedings of IPPS/SPDP 1998*, pages 410–414, 1998.

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