Hardware-Level Thread Migration to Reduce On-Chip Data Movement via Reinforcement Learning

Quintin Fettes, *Student Member, IEEE*, Avinash Karanth, *Senior Member, IEEE*, Razvan Bunescu, *Member, IEEE*, Ahmed Louri, *Fellow, IEEE*, and Kyle Shiflett, *Student Member, IEEE*

***Abstract*—As the number of processing cores and associated threads in chip multiprocessors (CMPs) continues to scale out, on-chip memory access latency dominates application execution time due to increased data movement. Although tiled CMP architectures with distributed shared caches provide a scalable design, increased physical distance between requesting and responding cores has led to both increased on-chip memory access latency and excess energy consumption. Near data processing is a promising approach that can migrate threads closer to data, however prior hand-engineered rules for fine-grained hardware-level thread migration are either too slow to react to changes in data access patterns, or unable to exploit the large variety of data access patterns. In this article, we propose to use reinforcement learning (RL) to learn relatively complex data access patterns to improve on hardware-level thread migration techniques. By utilizing the recent history of memory access locations as input, each thread learns to recognize the relationship between prior access patterns and future memory access locations. This leads to the unique ability of the proposed technique to make fewer, more effective migrations to intermediate cores that minimize the distance to multiple distinct memory access locations. By allowing a low-overhead RL agent to learn a policy from real interaction with parallel programming benchmarks in a parallel simulator, we show that a migration policy which recognizes more complex data access patterns can be learned. The proposed approach reduces on-chip data movement and energy consumption by an average of 41%, while reducing execution time by 43% when compared to a simple baseline with no thread migration; furthermore, energy consumption and execution time are reduced by an additional 10% when compared to a hand-engineered fine-grained migration policy.**

***Index Terms*—Chip multiprocessors (CMPs), data movement, reinforcement learning (RL), thread migration.**

I. Introduction

MANYCORE architectures suffer from both energy and performance penalties that exacerbate with the increase in the number of processing cores and associated threads [1]. As on-chip and off-chip memory access points become increasingly scattered around the on-chip interconnect fabric, memory access latency and energy consumption will increase as a result of longer round-trip request and response distances for data traversal. As technology scales into the subnanometer regime and provides opportunities to integrate hundreds of processing cores on a single chip, the cost of moving data in terms of energy and performance is currently dominating overall chip costs [1].

Near data processing is a powerful technique for reducing on-chip data movement during task execution [2]. Rather than repeatedly moving data from its on-chip storage location to the locus of computation, threads are moved closer to the locus of the data they require; if the cost (latency and energy) of moving the thread is lesser than the cost of moving the data, then the overall on-chip data movement is significantly reduced [2]–[11]. By reducing the distance data travels on the chip, both energy consumed by the on-chip network and memory access latency can be reduced.

Prior work advanced a deadlock free, fine-grain thread migration scheme that can take advantage of local memory access patterns to migrate threads at the granularity of a few cycles [6], [8], [9], [11]–[13]. Such a model used a distributed nonuniform memory architecture (NUMA) where each core has a slice of shared cache, and every cache line corresponds to a unique core where the data is located on chip. By enforcing a single copy of the data to be present on-chip, prior work avoids the complexity and performance drawbacks of cache coherence protocols. However, the performance benefits come at a cost of increased memory access latency since the data must be repeatedly requested from the remote core every time it is required because the data cannot be cached locally. To mitigate the loss in performance, prior work used simple hand-engineered rules which recognize one type of data access pattern to migrate the threads. However, prior thread migration techniques are either too slow to react to changes in data access patterns or too simple to exploit the large variety of data access patterns.

In this article, we propose to use reinforcement learning (RL) to learn data access patterns to improve hardware-level, deadlock-free thread migration techniques. In the proposed scheme, we use a history of recent memory accesses and their on-chip location to train a low-overhead RL policy to make

Manuscript received April 17, 2020; revised June 12, 2020; accepted July 6, 2020. Date of publication October 2, 2020; date of current version October 27, 2020. This work was supported in part by NSF under Award CCF-1513606, Award CCF-1513923, Award CCF-1702980, Award CCF-1703013, Award CCF-1812495, Award CCF-1901192, and Award CCF-1936794. This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis 2020 and appears as part of the ESWEEK-TCAD special issue. *(Corresponding author: Avinash Karanth.)*

Quintin Fettes, Avinash Karanth, Razvan Bunescu, and Kyle Shiflett are with EECS Department, Ohio University, Athens, OH 45701 USA.

Ahmed Louri is with the ECE Department, George Washington University, Washington, DC 20052 USA.

Digital Object Identifier 10.1109/TCAD.2020.3012650

migration decisions. By using the recent history of memory access locations as input, each thread learns to recognize the relationship between prior access patterns and future memory access locations. This enables the proposed technique to make fewer, more effective migrations to intermediate cores that minimize the distance to multiple remote memory locations simultaneously. By training RL agents on real interactions within parallel programming benchmarks in a parallel simulator, we show that a migration policy based on more complex data access patterns can be learned. The main contributions of this article are as follows.

Recognition of Complex Access Patterns: In [11] and [13], simple manually engineered rules trigger thread migration whenever a fixed number of consecutive accesses to remote data is observed. In this article, we show that a more intelligent, RL-trained policy can further reduce on-chip data movement. The proposed RL formulation relaxes constraints on where the threads can migrate and this allows the threads to move to intermediate cores that place the thread closer to multiple remote data locations. In Section III-A, we illustrate how the proposed RL policy learns to combine many local memory access statistics in a more sophisticated way to improve both energy and execution time.

Low-Overhead RL to Make Migration Decisions: As each thread can theoretically make a decision on whether or not to migrate on every memory access, it was crucial to design an RL solution that had minimal overhead. While tabular RL algorithms would be the quickest to determine the optimal core to migrate, the size of the state–action table can be prohibitively large to store in memory. Therefore, we use the six most recent memory accesses and the current location of the thread on chip as features for approximate Q-learning [14], [15], as well as other approximations to reduce the cost of computing migrations.

RL as an Improvement Operator for Existing Solutions: In the proposed formulation, there are approximately 1.76×10^{16} possible states. We use the simple policy from [9] and [11] to collect experience for pretraining the RL agent. Effectively, this forces the RL agent to first learn the value of a known, good policy, and then fine tune it to form a better, more complex policy. In this sense, the proposed algorithm can be thought of as a policy improvement operator on the previous policy. This step proved crucial to good performance.

Less On-Chip Traffic Than Existing Methods: We compared the performance of our proposed algorithm to a baseline that never migrates, as well as against our implementation of the NUMA architecture described in [9], [11], and [13] using the Snipersim simulator [16]. Relative to the baseline, the RL approach was able to save 41.1% energy and reduce execution time by 43.1% on a set of Splash2 [17] and Parsec [18] benchmarks. Likewise, when compared to the prior work from [9], the proposed RL algorithm saved an additional 10.2% energy and reduced execution time by an additional 9.6% on the same set of benchmarks.

II. Background

In this section, we provide background on the key aspects of the prior thread-migration-based framework described in [9]; this prior work is referred to as TM throughout this article. We also provide background on RL and the RL algorithm used to learn the migration policy.

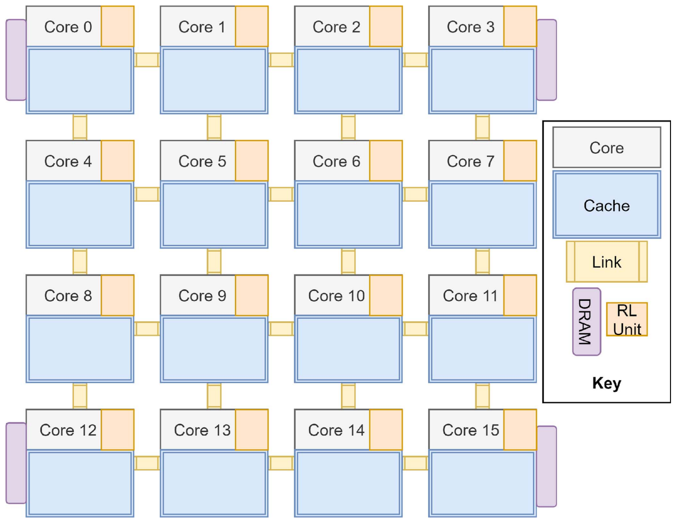


Fig. 1. Example 4 × 4 mesh architecture using NUMA design. Each core is connected to its immediate neighbors by links. Each core views all cache slices as a single, logical cache; however, accessing data on nonlocal cache slice will require the core to go on the network.

A. Prior Hardware-Level Thread Migration

1) Remote Access Cache: In the NUMA architecture used in [6], [8], [9], [11], and [13], the address space is divided among the cache slices such that each cache line is assigned a unique home core. Suppose a thread T running on core C is reading or writing data with address A whose home core is H. If $H = C$, T behaves normally such that the value is read/written without interacting with the on-chip interconnection network. On the other hand, if $H \neq C$, T must issue a request on the on-chip interconnection network and wait for a response for the memory operation to be completed. When compared to a directory-based cache coherence protocol, an important difference is that the remote cache block is *never cached locally at C*. Thus, accessing the cache line at H incurs round trip costs every time it is accessed. An example of a 4 × 4 mesh-based NUMA architecture is shown in Fig. 1.

2) Fine-Grained Deadlock-Free Thread Migration: Prior work in [9] seeks to exploit data locality by moving threads to the locus of data. Instead of remotely requesting memory operations, T could choose to migrate within the on-chip interconnection network to core H and execute the memory operation locally. Fig. 2 shows the migration decision process for a 5-stage pipeline architecture. If T is able to execute a sufficient number of memory operations on the remote core, the migration will have been "worth it." In [9], a thread T chooses to migrate if it executes $\zeta = 3$ consecutive memory accesses to a remote core. After ζ consecutive accesses to the remote core, the hardware interrupts the execution of the thread and sends it on the on-chip interconnection network to the remote core.

If a Thread T_1 is migrating to core H and a thread T' is already executing there, T' must be evicted. As described thus far, there are situations in which repeated evictions could lead

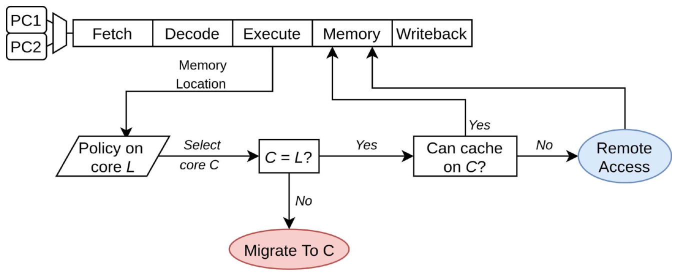


Fig. 2. This shows the migration decision process in a 5-stage pipeline. At the execution stage, the migration policy (hand-engineered or RL) identifies the home core of a memory address for a memory access instruction (load/store). This outputs a core, which can indicate either a migration or a decision to stay.

Algorithm 1 Fine-Grained Thread Migration [9]

```
 1: Input: a predetermined migration threshold $\zeta$
 2: Input: thread T executing on core C
 3: Input: address A whose home core is H
 4: procedure OnMemoryAccess($T, C, H, \zeta$)
 5:     if $T.last == H$ then
 6:         $T.d = T.d + 1$
 7:     else
 8:         $T.d = 1$
 9:     $T.last = H$
10:     if $T.d == \zeta$ then
11:         $T.d = 0$
12:         if $H \neq C$ then
13:             if $T.ncore == H$ then
14:                 Migrate $T$ to $H.nctxt$
15:             else
16:                 if $T'$ occupies $H.gctxt$ then
17:                     Evict $T'$ to $T'.ncore.nctxt$
18:                 Migrate $T$ to $H.gctxt$
19:         else
20:             $T$ stays at core $C$
```

to deadlock scenarios. To prevent deadlock, each core is made to have two thread contexts for which the core can multiplex execution [6], [9], [12], [13]. One context on each core H is marked as the *native context* $H.nctxt$ for the thread initialized to that core; and the other context is marked as the *guest context* $H.gctxt$. Only the thread initialized to a core can execute on its native context; we will use $T.ncore$ to denote the core that can run T in its native context. As a result, for every thread T, there exists a native context, on the core $T.ncore$, which can only be occupied by T; this guarantees that T can always return to an unoccupied thread context where it can resume execution after being evicted from a guest context. Furthermore, to prevent livelock scenarios in which threads are repeatedly migrated before making any progress, each thread is required to execute at least one instruction upon migrating to a new core. The migration policy implemented by TM is shown in Algorithm 1. There, the depth counter attribute $T.d$ keeps track of the number of consecutive accesses made by thread T to the same core, whereas $T.last$ denotes the core at which the most recently accessed memory address was located.

B. Reinforcement Learning

RL is a type of machine learning which attempts to maximize a cumulative reward signal that an agent receives after performing actions in an environment. The reward is a scalar value describing progress toward a goal after being in a state and taking an action, and can be sparse and noisy. The formal setting for RL is the Markov decision process (MDP), represented as a tuple (S, A, P, R, γ) where:

1) S is the set of all states s;
2) A is the set of all actions a;
3) $P : S \times A \to S$ is the model dynamics; it gives the probability of transitioning from the current state s to the next state s' upon taking the action a;
4) $R : (S \times A) \times S \to \mathbb{R}$ yields rewards for state transitions;
5) γ is a discount factor that controls how far into the future the agent will optimize the reward.

The goal of the agent is to learn an optimal policy $\pi^* : S \to A$ which maps states to actions such that the long-term expected reward is maximized. It can be shown that the optimal policy is $\pi^*(s) = \arg\max_a Q^*(s, a)$, where the action-value function Q^* satisfies the Bellman optimality equation [19]

$$Q^*(s, a) = \sum_{s', r} P\big(s', r|s, a\big)\left(r + \gamma \max_{a'} Q^*\big(s', a'\big)\right). \quad (1)$$

When the state space is small and discrete, the tabular Q-learning algorithm [19] is guaranteed to find the optimal action-value function $Q^*(s, a)$ by storing the value of every possible state–action pair in a table, provided that the agent visits every possible state–action pair a sufficient number of times. One common approach to ensure this is to use an ϵ-greedy policy, which chooses the action that maximizes the action-value function with probability $1 - \epsilon$, but reserves a small probability ϵ for selecting random actions [20].

1) Double Q-Learning With Linear Function Approximation: When the state space is large, as we will show later is the case for our proposed problem formulation, it is difficult or impossible to store the entire state–action table in memory and visit every state–action pair in a reasonable amount of time. In these cases, the action-value function can be approximated by some differentiable function $Q(s, a; \theta)$ where θ is a set of real-valued parameters, and the states are represented as vectors $s \subseteq \mathbb{R}^d$; this allows the agent to generalize to unseen state–action pairs and learn more quickly. However, Q-learning with function approximation is known to be unstable when the parameters are updated via gradient descent [20]. To improve stability during learning, [14] added a *target network*, i.e., a copy of the action-value function that updates more slowly, and *experience replay*, i.e., a buffer that stores old experience then samples mini-batches from it to get a better estimate of the gradient. We adopt a version of this algorithm that exclusively updates parameters offline and uses linear function approximation in lieu of a deep neural network. This is to

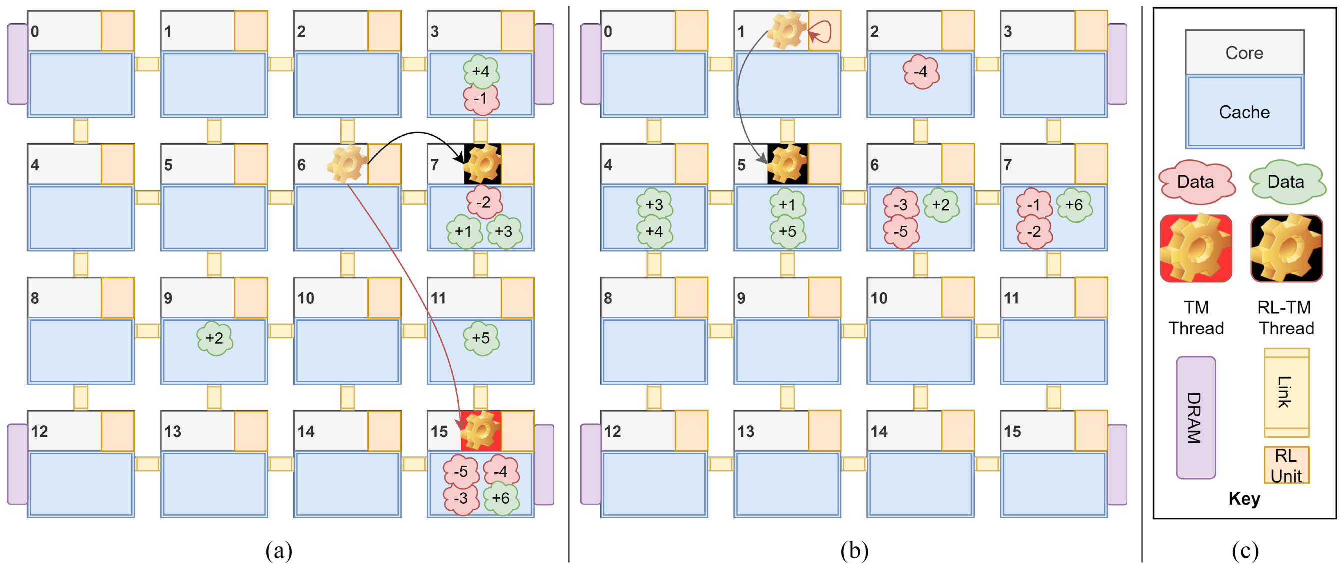


Fig. 3. (a) and (b) Show two examples of thread migrations made by the RL policy RL-TM and the baseline hand-engineered policy TM. The transparent gear shows the thread at its current location. The black lines show migrations made by RL-TM, whereas the red lines correspond to migration made by TM. RL-TM migrates in both scenarios: in (a) from Core 6 to Core 7, and in (b) from Core 1 to Core 5. TM migrates the thread only in (a), from Core 6, after observing three consecutive accesses to Core 15, whereas in (b), it never observes three consecutive accesses to the same core. The legend key is shown in (c). A detailed walkthrough of each scenario is provided in the text.

Algorithm 2 Double Q-Learning With Linear Function Approximation and Offline Learning [14], [15]

```
1:  Input: A Linear Function $Q : S \times A \times \mathbb{R}^d \rightarrow \mathbb{R}$
2:  Initialize Replay Memory $D$ to capacity $M$
3:  Initialize online $Q$ parameters $\theta$ randomly
4:  Initialize target $\hat{Q}$ parameter $\theta^- = \theta$
5:  for each Episode do
6:      Observe state vector $s_0 \in \mathbb{R}^d$
7:      for each $t$ in Episode do
8:          Select $a_t$ via an $\epsilon$-greedy policy on $Q(s_t, \cdot; \theta)$
9:          Execute $a_t$ then observe $r_{t+1}$ and $s_{t+1}$
10:         Store $(s_t, a_t, r_{t+1}, s_{t+1})$ in $D$
11:     for each Update Step do
12:         Sample a minibatch of $(s_j, a_j, r_{j+1}, s_{j+1})$ from $D$
13:         Select each $a'_j = \arg\max_a Q(s_{j+1}, a; \theta)$
14:         $y_j = \begin{cases} r_{j+1} & \text{if terminal} \\ r_{j+1} + \gamma \hat{Q}(s_{j+1}, a'_j; \theta^-) & \text{else} \end{cases}$
15:         Calculate the TD error $\delta_j = (y_j - Q(s_j, a_j; \theta))^2$
16:         Perform a gradient descent step on $\delta$ w.r.t. $\theta$
17:         Every $C$ steps set $\theta^- = \theta$
```

keep the simulator simple and fast, and to minimize the overhead of computing the optimal action, respectively. The target network is used in a double Q-learning setting [15], [21] to alleviate maximization bias and speed up learning. The full procedure that we used to learn the optimal policy is described in Algorithm 2. As seen in lines 13 and 14 of Algorithm 2, the online network parameterized by θ is used to select the next actions, while the offline target network parameterized by θ^- is used to provide their corresponding state–action values, effectively decorrelating action selection from action evaluation. Finally, when the buffer is full, new transitions are added by replacing older transitions at random. In Section III-B, we explain the use of this algorithm for training thread migration RL agents.

III. Reinforcement Learning for Hardware-Level Thread Migration

This section discusses the proposed RL-based thread migration (RL-TM). First, we use a walkthrough example of RL-TM that shows the advantages of RL-TM over the prior TM approach. Second, we introduce a formulation of hardware-level thread migration as a low-overhead RL problem. Last, we will discuss the RL overhead and implementation complexity.

A. Walkthrough Example of Proposed RL-TM Versus Prior TM

Fig. 3 illustrates the different migration decisions made by the proposed RL-TM and the prior TM algorithms on two real examples. The red $-k$ clouds indicate memory accesses k time steps in the past, e.g., -1 for the most recent memory access, and -2 for the memory access before it. Similarly, the green $+k$ clouds indicate memory accesses k time steps in the future, e.g., $+1$ indicates the upcoming memory access. At the time of the migration decision, RL-TM is aware of the location of the upcoming memory access as well as the prior five memory accesses, whereas TM operates using the location of the upcoming and previous two memory accesses.

Consider the example of the thread executing on Core 6 in Fig. 3(a). Upon observing the location of the access labeled $+1$, but prior to performing the actual memory access, RL-TM migrates to Core 7 which is one hop away from Core 6 to its right. This moves the thread controlled by RL-TM closer to five future accesses, while moving it further from only a single future access. On the other hand, the same thread, when controlled by TM, it observes three consecutive accesses (labeled -5, -4, and -3) to Core 15 and immediately migrates to Core 15 prior to performing the access labeled -3. As a result, under TM, the thread made a more costly three-hop migration to Core 15, moving it closer to three future memory accesses, but also farther from five future memory accesses.

In Fig. 3(b), based on the five prior accesses labeled with red clouds and the upcoming access labeled $+1$, RL-TM migrates the thread one hop away, from Core 1 to Core 5, which ends up closer to all six future memory accesses. On the other hand, TM does not migrate the thread, because no three consecutive accesses are made to any single core. This is a good example of the relative complexity of our policy. While TM was unable to identify a pattern which indicated a migration would be beneficial, RL-TM was able to consider a weighted combination of prior memory access locations to make a migration which ultimately reduced overall on-chip data movement. In both examples, by exploiting learned patterns that correlate previous with future access locations, RL-TM is more effective than TM at reducing the total distance of future accesses and also at lowering the cost of migrations.

B. Threads as Reinforcement Learning Agents

Each thread is associated with an RL agent that attempts to minimize the overall on-chip traffic. An agent collects local information about recent memory accesses from its operating core and then uses that information to make migration decisions prior to completing each individual memory access. As shown in Fig. 2, the RL-TM algorithm identifies the home core for a memory access instruction in the execution stage and passes this information to the RL agent. This location along with the previous five memory access locations forms an access history of length $k = 6$ which is used as input to the RL policy to select a core. If the core is the current location of the thread, the thread does not migrate and either executes the memory access locally or issues a remote request on the interconnection network. If the core is nonlocal, execution is interrupted, the thread context is serialized, and the thread is sent to the remote core where it resumes execution of the instruction.

Formally, there are N possible *actions*, where N is the number of cores in the network. For a thread running on core m, selecting action n corresponds to migrating to core n if $m \neq n$, or staying at the same core otherwise. The *state* contains information about the current location of the thread and the core at which the data for the k most recent memory accesses is located. Importantly, the most recent memory access has not been completed yet when the state vector is formed; the agent has identified the location of the upcoming access and is making a decision on whether or not it should migrate before completing that access, as shown in Fig. 2. The location of each of the k most recent memory accesses $\text{ma}_1, \text{ma}_2, \ldots, \text{ma}_k$ and the current thread location tl are encoded as one-hot vectors of length N. The resulting $k + 1$ one-hot vectors are then concatenated to form the overall state vector, i.e., $s_t = [\text{ma}_1, \text{ma}_2, \ldots, \text{ma}_k, tl]$.

The *reward* is computed after each completed memory access as a sum of a *data* cost and a *thread* migration cost

$$r(s_t, a_t) = -\text{dhops}(s_t.\text{ma}_1) - t\text{size} * \text{thops}(a_t, s_t.tl) \quad (2)$$

where *dhops* is the round-trip number of hops the current memory access needs to reach the requesting core, *tsize* is the size of the thread in flits, and *thops* is the number of hops the thread travels to migrate from the current core $s_t.tl$ to the core indicated by action a_t. The first term thus corresponds to the network traffic cost incurred from accessing data, whereas the second term estimates the network traffic cost caused by thread migration. Note that when $a_t = s_t.cl$, no migration happens, and thus $\text{thops}(a_t, s_t.tl) = 0$. Because the reward decreases as the amount of traffic increases, an agent seeking to maximize the reward is encouraged to make migration decisions that reduce the distance traveled by data in a way that optimally offsets the cost of migrating the thread.

During training, the agents follow Algorithm 2. One episode is considered to be a complete run of a single benchmark application, so each iteration of the outer for loop on lines 5–17 begins when a benchmark application starts. A new timestep begins when an agent (operating from the perspective of a thread) identifies the location of a memory access, but before the memory access has been completed; for this reason, each agent has its own set of timesteps and follows lines 7–10 asynchronously. At line 8, each agent uses previously observed memory access locations and the location of the upcoming memory access as the state s_t. The state s_t is used as input to the function Q to select the action a_t that determines the core to move to (or stay at). At line 9, the agent executes the migration decision then waits until the memory access location for the next memory access instruction is known; the reward r_{t+1} is computed and the new location is used to compute the next state s_{t+1}. Finally, at line 10, the agent stores the full transition information, $(s_t, a_t, r_{t+1}, s_{t+1})$ in the experience replay buffer. At this point, one iteration of the first inner for-loop has completed and the agent is ready to make another migration decision for timestep $t+1$. After the benchmark application has finished, lines 11–17 are executed for a fixed number of steps. In these steps, the algorithm repeatedly samples minibatches of the transitions stored by each agent to update the parameters θ. Because all agents try to solve the same optimization problem, i.e., minimize traffic due to on-chip data movement, they use the same policy. Therefore, they share the same action-value function $Q(S, A; \theta)$ and experience replay buffer D, which requires the simulator API to globally communicate transition, action selection, and update information during training. Allowing the agents to share experience and use the same policy has the added benefit that training will have a lower sample complexity and thus proceed much faster. Furthermore, for the first ten episodes of training, the behavior policy in line 8 of Algorithm 2 is replaced with the deterministic policy from [9], which migrates a thread to a core C if and only if it made three consecutive accesses to that core. After ten episodes, the agent follows its own ϵ-greedy policy shown on line 8, to improve on the initial deterministic policy.

At test time, each thread performs thread migration according to Algorithm 3. In step 2, a copy of the trained parameters

Algorithm 3 Low-Overhead RL at Test Time

```
1: Input: A Linear Function $Q : S \times A \rightarrow \mathbb{R}$
2: Load previously trained $Q$ parameters $\theta$
3: Observe state vector $s_0 \in \mathbb{R}^d$
4: for each $t$ do
5:     Select action $a_t$ using an $\epsilon$-greedy policy on $Q(s_t, \cdot; \theta)$
6:     Execute $a_t$ then observe next state $s_{t+1}$
```

is loaded into the L1-D cache for each core so that threads can access the trained policy without the need for nonlocal communication. Additionally, the specialized hardware described in Section III-C is added to each core to facilitate low-overhead computation of migration decisions, which results in a very low-overhead RL policy at test time. Similar to training, each agent has its own set of timesteps and executes lines 4–6 of Algorithm 3 asynchronously. At line 5, each agent uses the location of previously observed memory access locations and the upcoming access location to form the state vector s_t; s_t is then used as input to the function Q to select the action a_t, which determines the core to migrate to (or stay at). Finally, at line 6, the agent executes a_t and waits to observe the location of the next memory access to form the next state s_{t+1}. This loop continues indefinitely as long as threads need to execute memory access instructions. While using a fixed policy may lead to lower adaptability at test time, prior work has shown that a single, well-selected thread migration policy can work well across a wide range of application benchmarks [9], [11], [13].

C. Overhead of Reinforcement Learning at Test Time

The total number of parameters requiring storage at test time is calculated as follows. There are $k+1$ one-hot vectors used as input, each of length N, where N denotes the number of cores. The first k vectors represent the location of the k most recent memory accesses by a thread, whereas the last one-hot vector represents the thread's current location. In our experiments, k and N were 6 and 16, respectively, as seen in Tables I and II. Given that a distinct set of parameters is needed for each of the N actions for computing their action value, this implies that the total number of parameters is $(k+1)N^2$. However, at test time, the policy is copied into each core; the one-hot vector that represents the location of the thread is not needed anymore. Thus, only kN^2 parameters must be stored at each core. To further reduce the storage and computation overhead of Algorithm 3, the 32 b floats used at training time are converted to 8 b integers at test time. In total, the action-value parameters are stored in approximately 1.5 kB of the L1-D cache at each core.

To further reduce the overhead, during testing, we restricted the agents to make migration decisions only on every 6th memory access. Fig. 4 shows the energy and execution time of RL-TM as the period of migration decisions is varied. While the performance on splash2-fft is slightly better for a period of 5, the 0.3% performance gain would be offset by the increased overhead cost (Section III-D) of making more frequent migration decisions. Additionally, we found that it was possible to reduce the number of cores considered at test time with a simple heuristic: a minimal bounding box was formed to contain all cores corresponding to the last k memory accesses. On average, this reduced the number of actions considered from $N = 16$ (the number of cores) to $N_0 = 9.87$, which is a 38.31% decrease in computation.

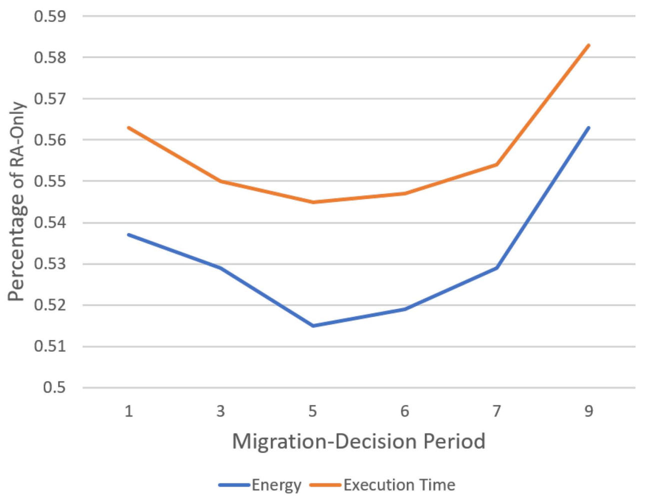


Fig. 4. Energy and execution time performance of RL-TM relative to the baseline RA-Only on splash2-fft as the migration period is varied.

TABLE I
SYSTEM CONFIGURATION

Parameter	Settings
Cores, N	16
Thread Contexts per Core	2
Process Size	45nm
L1/L2 cache per core	32KB/128KB
L1/L2 Associativity	8/8 way set associative
Cache Block Size	64b
Interconnection Network Shape	2D Mesh
Interconnection Network Routing	XY routing
Hop Latency	2 cycles
Flit Size	128b
Thread Context Size	3.1kb

Because all k input vectors use one-hot encoding, multiplication with the corresponding parameter vectors simply requires looking up the parameters corresponding to the index of the nonzero element in the one-hot vector. Given that there are k input vectors, this implies that k parameters must be summed up for each of the N_0 pruned actions. The resulting N_0 action-values must then be compared to find the maximum. Finally, selecting actions on every 6th access reduces the total number of selections to $(1/6)$ of the number of memory accesses. Overall, the time complexity consists of kN_0 8 b integer lookups, $(k-1)N_0$ 8 b integer additions, and $N_0 - 1$ 8 b integer comparisons.

D. RL-TM Unit Implementation

The RL unit first uses adders to sum k 8 b integers for an average of ten actions on each migration decision; these sums represent the action-value for each of the pruned actions. To sum the k values, we utilize carry-save adders (CSA) organized in a Wallace tree [23] to ensure a low propagation

TABLE II
RL HYPERPARAMETERS

Parameter	**Settings**
Training Steps	$1.5e4$ per episode
Memory Access Location History	6 accesses
Learning Rate, α	$2.5e-5$
Gamma, γ	0.99
Optimizer	Adam [22]
Adam Epsilon	$1e-8$
Experience Replay Size	$1.5e6$
Update Batch Size	32
Target Network Update Frequency, C	$1e3$
Exploration Rate, ϵ	$0.1 * 0.9^E$
Train/Test Decision Frequency	Every 1/6 access(es)
Length of Access History, k	6

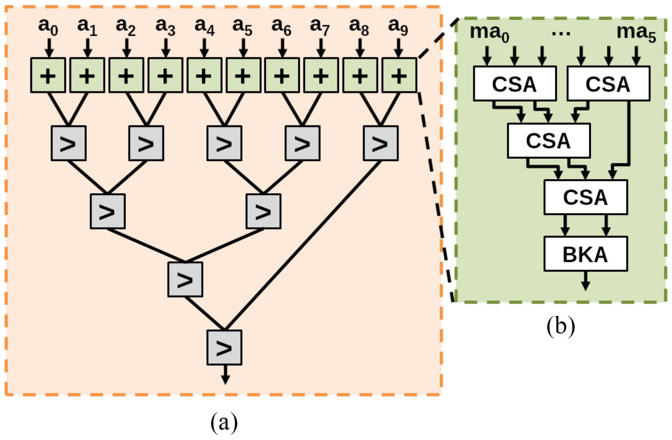


Fig. 5. RL-TM unit block diagram showing: (a) adder and comparator tree organization and (b) Wallace tree adder.

delay. Furthermore, a Brent–Kung adder (BKA) [24] is utilized to minimize power and area overhead. The ten sets of k weights are assigned to a unique adder unit to compute the sums asynchronously; the resulting ten sums are passed to a comparator tree to find the maximizing action, which is returned and used to make a migration decision. The RL unit implementation is shown in Fig. 5. After modeling the hardware unit in Verilog HDL and synthesizing via Synopsys DC Ultra using the NanGate 45-nm open-cell library, results show a timing overhead of 1.7 ns (< 5 cycles) and a power overhead of 5.23 pJ, while the area overhead is 7813 μm^2. Including the RL units, the total area of the chip is 746.14 mm^2; thus, collectively the 16 RL units occupy 15.9% of the total area of the chip. Because migration decisions are computed every six memory accesses in this design, the timing and energy overhead per memory access is effectively 0.28 ns and 0.87 pJ, respectively. Using the specialized hardware added to each core, this yields a timing and energy overhead of 1 cycle and 0.87 pJ on average for every migration decision. In context, 1.4% of the total cycles are spent computing migration decisions. Overall, the resulting energy and timing penalty are negligible relative to the total energy consumption and execution time. While already small, these overhead numbers will decrease significantly as the architecture is modernized into the sub-10-nm scale.

IV. Performance Evaluation

This section discusses the network simulator, its configuration, and its use in training and evaluation of the proposed RL-TM scheme.

A. Simulation Setup

The evaluation was performed using the Snipersim multicore simulator [16] on a set of Splash2 [17] and Parsec [18] benchmarks. Snipersim is designed to use a directory-based memory hierarchy when organized in a network on chip mesh architecture. Accordingly, the simulator was modified to enable the remote access memory architecture described throughout this document. Furthermore, the simulator does not support hardware-level thread migration out of the box. To enable these experiments, the simulator was also modified to compute the cost in latency and energy associated with migrating threads over the interconnection network. For a fair comparison to [9], which was originally evaluated on a different simulator, we implemented their migration algorithm on the modified copy of Snipersim using the same simulation parameters. The relevant chip design specifications can be seen in Table I; notably, the system uses only 16 cores due to the time complexity of making migration decisions frequently at every core during the simulation. As simulator technology improves along with computational power, training thread migration models on a larger number of cores will become more feasible, with the results in Section V indicating that it will scale effectively.

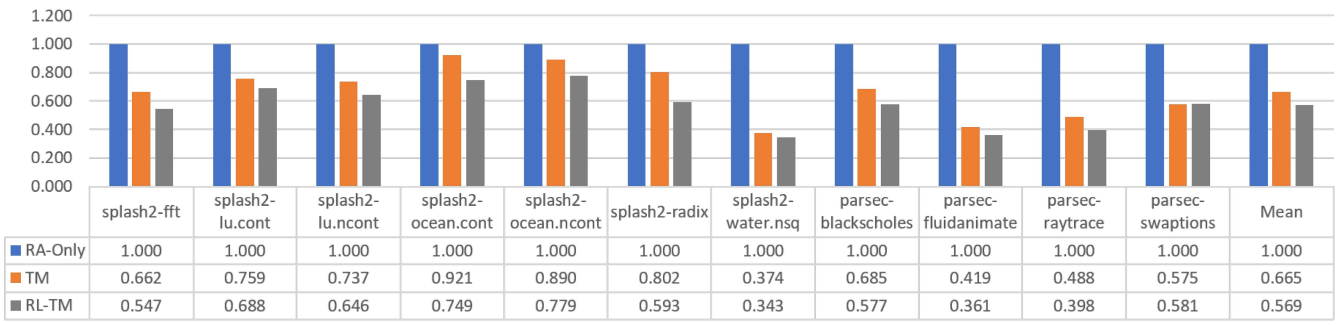
For the RL-enabled thread migration described in previous sections, henceforth referred to as RL-TM, a trained RL policy will be queried on every memory access by every thread on the chip as seen in Fig. 2. The migration policy of the TM model compares the number of consecutive accesses to memory on the same core to a predetermined threshold; RL-TM replaces that simple manually engineered policy with a policy learned using a more complex combination of network statistics via Algorithm 2.

B. RL-TM Evaluation Procedure

We use a leave-one-out evaluation scenario, where at every step a single Splash2 or Parsec benchmark is used for testing, whereas all the remaining benchmarks are used to gather experience and train the RL policy; this process is repeated so that each benchmark is used for testing once. By never using the test benchmark during training, the leave-one-out evaluation results reflect how well a trained policy generalizes to an unseen benchmark. The hyperparameters used during training can be seen in Table II.

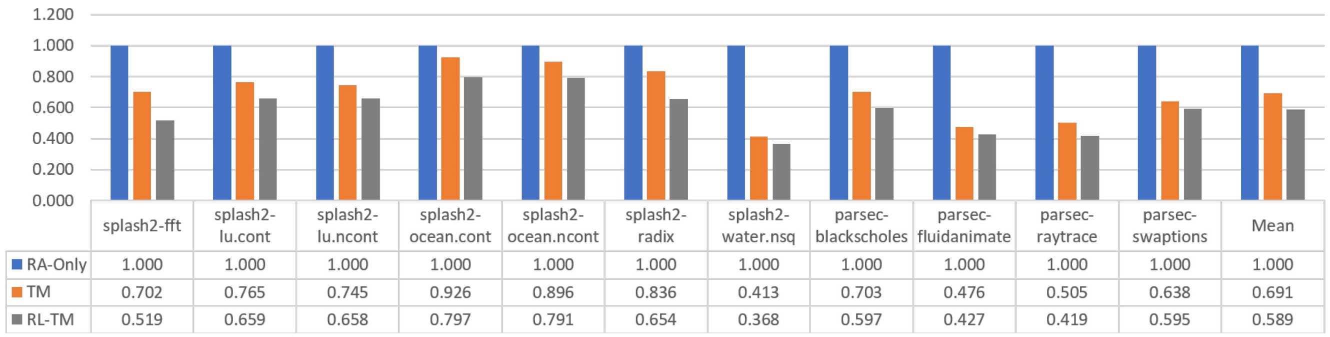
V. Results and Analysis

We compare the proposed RL-TM to both a remote-access-only (RA-Only) baseline which can never migrate threads and TM. A breakdown of the execution time for each of the benchmarks can be viewed in Fig. 6. It can be seen that the RL-enabled thread migration algorithm reduces the total execution time by an average of 43.1% relative to RA-Only, and



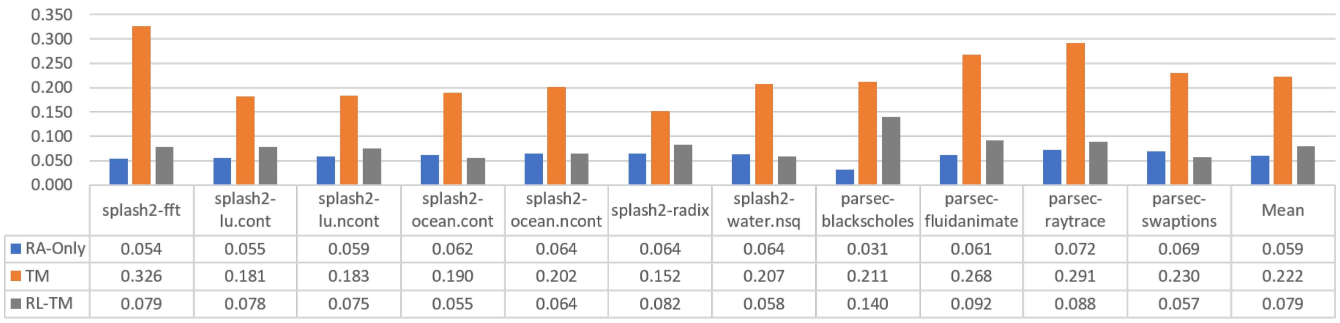
	splash2-fft	splash2-lu.cont	splash2-lu.ncont	splash2-ocean.cont	splash2-ocean.ncont	splash2-radix	splash2-water.nsq	parsec-blackscholes	parsec-fluidanimate	parsec-raytrace	parsec-swaptions	Mean
🟦 RA-Only	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
🟧 TM	0.662	0.759	0.737	0.921	0.890	0.802	0.374	0.685	0.419	0.488	0.575	0.665
■ RL-TM	0.547	0.688	0.646	0.749	0.779	0.593	0.343	0.577	0.361	0.398	0.581	0.569

Fig. 6. Shown is execution time of the thread migration frameworks discussed. All values are normalized to the value of the RA-Only migration framework.



	splash2-fft	splash2-lu.cont	splash2-lu.ncont	splash2-ocean.cont	splash2-ocean.ncont	splash2-radix	splash2-water.nsq	parsec-blackscholes	parsec-fluidanimate	parsec-raytrace	parsec-swaptions	Mean
🟦 RA-Only	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
🟧 TM	0.702	0.765	0.745	0.926	0.896	0.836	0.413	0.703	0.476	0.505	0.638	0.691
■ RL-TM	0.519	0.659	0.658	0.797	0.791	0.654	0.368	0.597	0.427	0.419	0.595	0.589

Fig. 7. Energy consumption of the three thread migration policies.



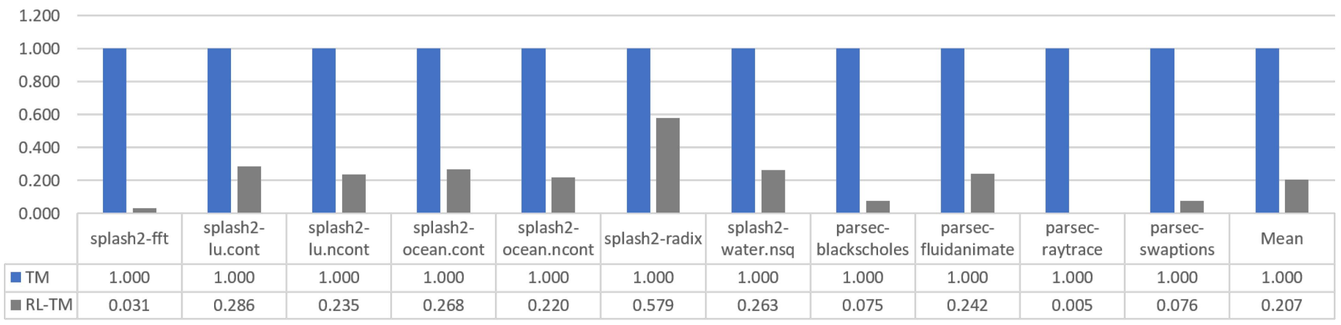
	splash2-fft	splash2-lu.cont	splash2-lu.ncont	splash2-ocean.cont	splash2-ocean.ncont	splash2-radix	splash2-water.nsq	parsec-blackscholes	parsec-fluidanimate	parsec-raytrace	parsec-swaptions	Mean
🟦 RA-Only	0.054	0.055	0.059	0.062	0.064	0.064	0.064	0.031	0.061	0.072	0.069	0.059
🟧 TM	0.326	0.181	0.183	0.190	0.202	0.152	0.207	0.211	0.268	0.291	0.230	0.222
■ RL-TM	0.079	0.078	0.075	0.055	0.064	0.082	0.058	0.140	0.092	0.088	0.057	0.079

Fig. 8. Percentage of cache accesses which were local to the accessing thread's core.

by an additional 9.6% relative to TM. Additionally, the comparison of energy consumption is presented in Fig. 7. Here, it is shown that the RL-enabled thread migration algorithm reduced the overall energy consumption by 41.1% relative to the RA-Only baseline and an additional 10.2% compared to TM. Figs. 9 and 10 show that across all benchmarks, the number of migrations and evictions using RL-TM were reduced by 79.3% and 54.0%, respectively. The average distance (in hops) each memory access must travel round-trip on chip is shown in Fig. 11. RL-TM reduces the average access distance by 23.7% relative to RA-Only, and an additional 1.1% relative to TM. Finally, Fig. 12 shows the normalized energy consumption of RL-TM when it is trained using applications from one benchmark suite evaluated using the application from the other benchmark; results are normalized to the energy consumption of RL-TM when trained using benchmark applications from both benchmark suites, excluding the application being evaluated. Fig. 12(a) shows an average increase of 2.6% in energy consumption, whereas Fig. 12(b) shows a mean increase of 3.4%. While it is possible that similarity among applications within the same benchmark suite (see splash2-lu.cont and splash2-lu.ncont) helps boost performance to some degree, it is also possible that the relative drop in performance when restricting the training set is simply due to a less diverse training set.

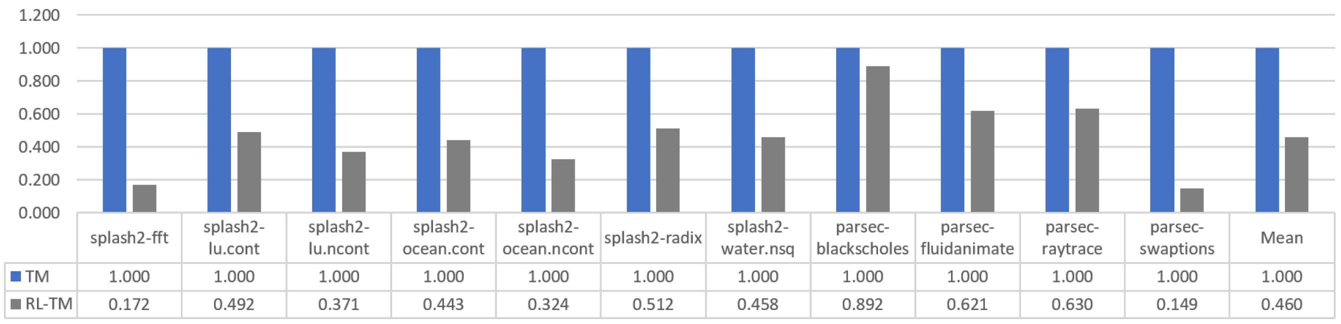
A. Analysis of Learned Policy

Results in Figs. 6 and 7 show that RL-TM is a more effective thread migration algorithm than prior work while also maintaining enough generality to work across multiple benchmark suites using the same set of hyperparameters. Thus, the results indicate that using the hyperparameters shown in Table II would lead to effective thread migrations on a wide variety of parallel applications with little or no added design complexity, allowing the methodology to be extended to new applications easily, and could improve with a wider variety of training data. The rest of this section will analyze and explain the performance of RL-TM.



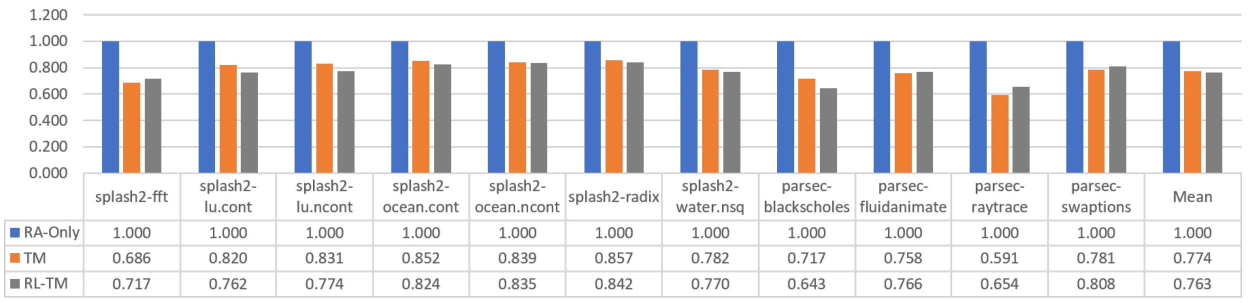
	splash2-fft	splash2-lu.cont	splash2-lu.ncont	splash2-ocean.cont	splash2-ocean.ncont	splash2-radix	splash2-water.nsq	parsec-blackscholes	parsec-fluidanimate	parsec-raytrace	parsec-swaptions	Mean
🟦 TM	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
⬛ RL-TM	0.031	0.286	0.235	0.268	0.220	0.579	0.263	0.075	0.242	0.005	0.076	0.207

Fig. 9. Number of migrations performed by RL-TM relative to the TM algorithm of [9]; RA-Only is omitted because it never migrates threads.



	splash2-fft	splash2-lu.cont	splash2-lu.ncont	splash2-ocean.cont	splash2-ocean.ncont	splash2-radix	splash2-water.nsq	parsec-blackscholes	parsec-fluidanimate	parsec-raytrace	parsec-swaptions	Mean
🟦 TM	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
⬛ RL-TM	0.172	0.492	0.371	0.443	0.324	0.512	0.458	0.892	0.621	0.630	0.149	0.460

Fig. 10. Number of evictions caused by RL-TM relative to the algorithm of [9]; RA-Only is omitted because it never migrates threads.



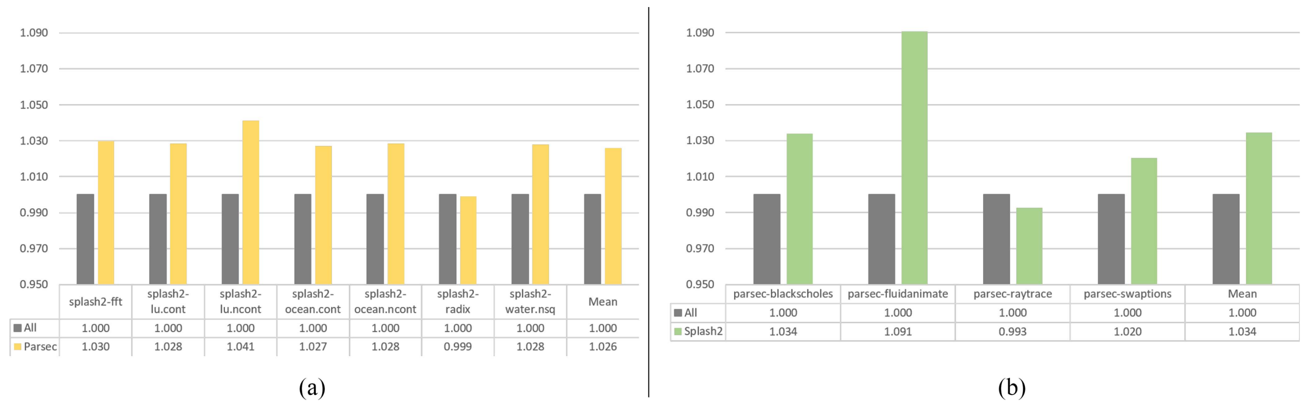
	splash2-fft	splash2-lu.cont	splash2-lu.ncont	splash2-ocean.cont	splash2-ocean.ncont	splash2-radix	splash2-water.nsq	parsec-blackscholes	parsec-fluidanimate	parsec-raytrace	parsec-swaptions	Mean
🟦 RA-Only	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
🟧 TM	0.686	0.820	0.831	0.852	0.839	0.857	0.782	0.717	0.758	0.591	0.781	0.774
⬛ RL-TM	0.717	0.762	0.774	0.824	0.835	0.842	0.770	0.643	0.766	0.654	0.808	0.763

Fig. 11. Shown is the normalized mean memory access distance (in hops) for each method; for each algorithm on each benchmark, this is the expected distance each memory access will have to travel on-chip to be completed.

To understand why RL-TM is working, it is useful to first look at the number of migrations and evictions shown in Figs. 9 and 10. These show that the overall number of migrations and evictions for RL-TM is significantly reduced when compared to TM. Additionally, Fig. 11 shows that RL-TM manages to reduce the average access distance by a smaller amount. Together, these three results show that RL-TM is able to achieve better data locality with a much smaller number of migrations. Migrations have a very large overhead due to the large thread context that must be sent over the interconnection network, so making less migrations significantly reduces timing and energy overhead for thread migration schemes. Interestingly, Fig. 8 shows that RL-TM slightly improves over the RA-Only baseline by 2% increased local access rate, but makes 14.3% less local accesses than TM. This displays the other strength of RL-TM; where TM is only ever able to migrate to a core where it has made multiple consecutive memory accesses, RL-TM is also able to migrate to intermediate cores that are close to multiple caches from which it needs data. In other words, while RL-TM less frequently moves directly to cores that can cache the data it needs, the benefit of moving to intermediate cores allows it to migrate less frequently while maintaining better data locality than RA-Only and TM. This reliance on intermediate cores suggests that RL-TM will become increasingly beneficial as the number of cores grows when compared to either baseline, as RL-TM will minimize expected distance to data, which will be increasingly distributed among a larger number of cores. While this indicates that a trained RL policy could become even more useful with higher core counts, the performance of the algorithm under practical constraints imposed by a growing state–action space remains an empirical question and is left for future work.

B. Need for Bootstrapping

In the proposed RL formulation, there are approximately 1.76×10^{16} possible states. Rather than throwing out the valuable policy used by TM, we use it as the initial policy for the RL agents. Effectively, this forces the RL agent to learn the



	splash2-fft	splash2-lu.cont	splash2-lu.ncont	splash2-ocean.cont	splash2-ocean.ncont	splash2-radix	splash2-water.nsq	Mean
■ All	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
■ Parsec	1.030	1.028	1.041	1.027	1.028	0.999	1.028	1.026

	parsec-blackscholes	parsec-fluidanimate	parsec-raytrace	parsec-swaptions	Mean
■ All	1.000	1.000	1.000	1.000	1.000
■ Splash2	1.034	1.091	0.993	1.020	1.034

Fig. 12. (a) and (b) Show the energy consumption of RL-TM when restricting the training set to include applications from only one benchmark suite, normalized to the energy consumption when RL-TM is allowed to train on all benchmark applications except the one being evaluated. In (a) RL-TM was trained using the four Parsec benchmarks then evaluated on the seven Splash2 benchmarks. In (b) RL-TM was trained on Splash2 benchmarks and evaluated on Parsec benchmarks.

value of a known, good solution, and then fine-tune it into a better, more complex policy. This bootstrapping step was instrumental for obtaining good performance with RL-TM, which effectively makes RL-TM a policy improvement algorithm. A similar approach was taken in [25], where a policy gradient RL algorithm was used to improve a simpler search algorithm to play the game of Go. This view suggests that collecting data from other manually engineered policies, which make good migrations based on other simple access patterns, and using that data to pretrain RL-TM could further improve performance.

VI. Related Work

The most similar and a direct inspiration to our work is the work in [6], [8], and [9]. There, the authors focused on introducing the concept of a directoryless, shared-memory protocol that relied on informed, fast, provably deadlock-free thread migration to reduce cache miss rates. The authors used a program counter to track the number of consecutive accesses to remote cores; after a predetermined, hand-selected number of remote accesses to the same core, a thread would interrupt execution and migrate via the chip interconnection network to the location of the remote data. Our work adopts the idea of a NUMA memory design that relies on migrations to improve data locality, but uses a history of memory access locations to train a low-overhead RL policy to make decisions to migrate to any core in the network, rather than just the core associated with the remote access. In [11], this idea was improved upon by introducing a stack-based core architecture. This allowed the system to dynamically decide to migrate differing amounts of the thread context, thus reducing the overhead incurred by migrating a thread over the interconnection network. Reference [3] introduces the idea of computation migration; this is shown as an alternative to bringing data to the locus of computation, where programmers annotate the location where procedures should be run to improve data locality. Rather than copying memory to multiple locations and using a coherence protocol, [26] explore systems to reactively place pages and threads on chip according to dynamic measurements made at runtime at the programming language level. Weissman *et al.* [27] introduced a thread migration technique in an SMP multiprocesor system which is performed by the operating system at timeslice granularity; OS-based migration protocols such as [27] were effective for achieving long-term goals like load balancing, but due to the high overhead of such methods, optimizing data locality for short bursts of data accesses is better suited for methods like those presented in this article. Jiang and Chaudhary [28] used a programming language-level method to identify idle cycles at compile time which could be utilized to hide the overhead of these relatively high-overhead programming language level and OS-level migration schemes. Similarly, [29] attempts to alleviate the overhead of migrating call-stacks by compiling functions to a machine-independent string so that a programmer can specify when to move function execution on a heterogeneous system. Forbes *et al.* [10] introduced a system that makes migration decisions both via compile-time optimization and via an OS-level system that analyzes the runtime behavior of threads. On the other hand, [4] and [5] utilize both dynamic voltage and frequency scaling and thread migration to save energy while sidestepping the area/performance overhead penalties of using DVFS alone; likewise, all methods rely on a central arbiter or handshake that makes the decision to migrate threads to a higher or lower frequency core based on their current workload, and cannot be considered truly low-overhead or fine-grained. In an attempt to further reduce the latency of the actual migration, [10] introduces the Teleportation Register File and special migration hardware to reduce the amount of time it takes to migrate threads on a heterogeneous chip in a register-based ISA, rather than the stack-based ISA used by [11].

Additional work has been done which focuses on the heterogeneous nature of modern system-on-chips (SoCs). Huang *et al.* [30] attempted to map tasks to the appropriate type of processing element in heterogeneous SoCs; this idea is improved in [31] by dynamically scheduling tasks which communicate to a single processing element. Similarly, Masood *et al.* [32] decomposed applications to tasks then dynamically maps these tasks onto processing elements at runtime to minimize communication between tasks. Xiao *et al.* [33] utilized compile-time optimization to identify dependent instructions, then partitions them into tasks; an RL-based task scheduler then attempts to place tasks on their preferred type of processing element such that the distance to other tasks which they communicate with is minimized. While similar in spirit to thread migration, the aforementioned task mapping algorithms dynamically provide only initial mappings for tasks, and they could benefit from thread migration algorithms, such as the proposed algorithm RL-TM, which allow threads to move after their initial mapping; this is especially true in parallel applications which require frequent communications between many threads.

In addition, RL has been applied to many other tasks on the network-on-chip (NoC). Power management is frequently addressed by training RL agent(s) to make dynamic voltage and frequency scaling decisions for on-chip components [34]–[36]. Additionally, RL policies have been used to save energy by learning to power-gate NoC components with the goal of reducing static power consumption [34], [37]. Wang *et al.* [37] used RL to provide adaptive error detection and correction in the NoC. Ipek *et al.* [38] used RL to learn an adaptive scheduling policy for the memory controller in a chip multiprocessor. Another common application of RL in the NoC is learning dynamic/adaptive routing schemes [39]–[41]; each of these works attempt to alleviate network congestion by introducing a dynamic routing algorithm learned via RL. On the other hand, our proposed RL-TM algorithm attempts to dynamically assign threads to cores for any parallel processing application, and is orthogonal to all of the above RL-for-NoC work.

VII. Conclusion

Researchers predict the number of cores on multicores will reach the thousands in the near future. The proposed RL-TM offers a proof of concept that low-overhead approximate RL can be used to train a policy which effectively utilizes intermediate cores to make migrations to minimize the expected access distance for future memory accesses. By stripping out test-time learning which is characteristic of most contemporary RL algorithms and making several other improvements, RL-TM requires negligible overhead to compute migration decisions. Compared with a simple baseline that never migrates threads, the proposed methodology reduces execution time by 43.1% and energy consumption by 41.1%. When compared to a similar algorithm which utilizes a human-engineered rule, RL-TM reduces execution time by an additional 9.6% and energy consumption by an additional 10.2%. Data collected from experiments shows that this decrease can largely be attributed to a 79.3% decrease in the number of migrations made, a 54.0% reduction in thread evictions and the use of intermediate cores as migration locations to improve data locality. RL-TM is able to do this by bootstrapping from the simpler human-engineered policy. Finally, the experiments on both Splash2 and Parsec benchmarks suggest that our proposed methodology generalizes to many parallel applications. Future work could diversify the set of benchmarks even further as simulators become more efficient and computers to run experiments become faster. Altogether, RL-TM is a general approach to hardware-level thread migration which effectively reduces on-chip data movement to both reduce execution time and save energy.

Acknowledgment

The authors sincerely thank all the anonymous reviewers for their excellent feedback on this work.

References

[1] M. Ottavi *et al.*, "Dependable multicore architectures at nanoscale: The view from europe," *IEEE Design Test*, vol. 32, no. 2, pp. 17–28, Apr. 2015.

[2] A. Pattnaik *et al.*, "Opportunistic computing in GPU architectures," in *Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 210–223. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322212

[3] W. C. Hsieh, P. Wang, and W. E. Weihl, "Computation migration: Enhancing locality for distributed-memory parallel systems," *ACM SIGPLAN Notices*, vol. 28, no. 7, pp. 239–248, Jul. 1993. [Online]. Available: https://doi.org/10.1145/173284.155357

[4] K. K. Rangan, G.-Y. Wei, and D. Brooks, "Thread Motion: Fine-grained power management for multi-core systems," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, Austin, TX, USA, 2009, pp. 302–313, [Online]. Available: http://doi.acm.org/10.1145/1555754.1555793

[5] Q. Cai, J. González, G. Magklis, P. Chaparro, and A. González, "Thread shuffling: Combining DVFS and thread migration to reduce energy consumptions for multi-core systems," in *Proc. IEEE/ACM Int. Symp. Low Power Electron. Design*, Fukuoka, Japan, Aug. 2011, pp. 379–384.

[6] M. Lis, K. Shim, M. H. Cho, O. Khan, and S. Devadas, "Directoryless shared memory coherence using execution migration," in *Proc. IASTED Int. Conf. Parallel Distrib. Comput. Syst.*, Dec. 2011, doi: 10.2316/P.2011.757-081.

[7] C. Ravishankar, S. Ananthanarayanan, S. Garg, and A. Kennings, "Analysis and evaluation of greedy thread swapping based dynamic power management for MPSoC platforms," in *Proc. 13th Int. Symp. Qual. Electron. Design (ISQED)*, Santa Clara, CA, USA, Mar. 2012, pp. 617–624.

[8] M. Lis, K. S. Shim, B. Cho, I. Lebedev, and S. Devadas, "Hardware-level thread migration in a 110-core shared-memory multiprocessor," in *Proc. IEEE Hot Chips 25 Symp. (HCS)*, Stanford, CA, USA, Aug. 2013, pp. 1–27.

[9] K. S. Shim, M. Lis, O. Khan, and S. Devadas, "Thread migration prediction for distributed shared caches," *IEEE Comput. Archit. Lett.*, vol. 13, no. 1, pp. 53–56, Jan.–Jun. 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6313584/

[10] E. Forbes *et al.*, "Under 100-cycle thread migration latency in a single-ISA heterogeneous multi-core processor," in *Proc. IEEE Hot Chips 27 Symp. (HCS)*, Cupertino, CA, USA, Aug. 2015, p. 1.

[11] K. S. Shim, M. Lis, O. Khan, and S. Devadas, *The Execution Migration Machine: Directoryless Shared-Memory Architecture*, IEEE, Piscataway, NJ, USA, Sep. 2015. [Online]. Available: https://dspace.mit.edu/handle/1721.1/108136

[12] K. S. Shim, M. Lis, M. H. Cho, O. Khan, and S. Devads, "System-level optimizations for memory access in the execution migration machine (EM2)," in *Proc. 2nd Workshop Comput. Archit. Oper. Syst. Co-Design (CAOS)*, 2011, pp. 11–23.

[13] M. N. M. N. Lis, "Hardware-level fine-grained thread migration," M.S. thesis, Dept. Elect. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 2014. [Online]. Available: https://dspace.mit.edu/handle/1721.1/93066

[14] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: https://www.nature.com/articles/nature14236

[15] H. V. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double *Q*-learning," in *Proc. 13th AAAI Conf. Artif. Intell.*, Mar. 2016, pp. 2094–2100. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12389

[16] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, p. 28, 2014.

[17] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. 22nd Annu. Int. Symp. Comput. Archit.*, 1995, pp. 24–36. [Online]. Available: http://doi.acm.org/10.1145/223982.223990

[18] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Dept. Doctor Philos., Princeton Univ., Princeton, NJ, USA, Jan. 2011.

[19] C. J. C. H. Watkins and P. Dayan, "*Q*-learning," *Mach. Learn.*, vol. 8, no. 3, pp. 279–292, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992698

[20] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.

[21] H. V. Hasselt, "Double *Q*-learning," in *Advances in Neural Information Processing Systems 23*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, Eds. Red Hook, NY, USA: Curran Assoc., Inc., 2010, pp. 2613–2621. [Online]. Available: http://papers.nips.cc/paper/3964-double-q-learning.pdf

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," Jan. 2017. [Online]. Available: http://arxiv.org/abs/1412.6980.

[23] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electron. Comput.*, vol. EC-13, no. 1, pp. 14–17, Feb. 1964.

[24] R. B. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. Comput.*, vol. C-31, no. 3, pp. 260–264, Mar. 1982.

[25] D. Silver *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017. [Online]. Available: https://www.nature.com/articles/nature24270

[26] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proc. 3rd Conf. Comput. Front.*, Ischia, Italy, May 2006, pp. 29–40. [Online]. Available: https://doi.org/10.1145/1128022.1128029

[27] B. Weissman, B. Gomes, J. W. Quittek, and M. Holtkamp, "Efficient fine-grain thread migration with active threads," in *Proc. 1st Merged Int. Parallel Process. Sympo. Symp. Parallel Distrib. Process.*, Mar. 1998, pp. 410–414.

[28] H. Jiang and V. Chaudhary, "MigThread: Thread migration in DSM systems," in *Proc. Int. Conf. Parallel Process. Workshop*, Aug. 2002, pp. 581–588.

[29] R. Veldema and M. Philippsen, "Near overhead-free heterogeneous thread-migration," in *Proc. IEEE Int. Conf. Clust. Comput.*, Burlington, MA, USA, Sep. 2005, pp. 1–10.

[30] J. Huang, A. Raabe, C. Buckl, and A. Knoll, "A workflow for runtime adaptive task allocation on heterogeneous MPSoCs," in *Proc. Design Autom. Test Eur.*, Grenoble, France, Mar. 2011, pp. 1–6.

[31] A. K. Singh, T. Srikanthan, A. Kumar, and W. Jigang, "Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms," *J. Syst. Archit.*, vol. 56, no. 7, pp. 242–255, Jul. 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762110000330

[32] A. Masood, E. U. Munir, M. M. Rafique, and S. U. Khan, "HETS: Heterogeneous edge and task scheduling algorithm for heterogeneous computing systems," in *Proc. IEEE 17th Int. Conf. High Perform. Comput. Commun. 7th Int. Symp. Cyberspace Safety Security 12th Int. Conf. Embedded Softw. Syst.*, New York, NY, USA, Aug. 2015, pp. 1865–1870.

[33] Y. Xiao, S. Nazarian, and P. Bogdan, "Self-optimizing and self-programming computing systems: A combined compiler, complex networks, and machine learning approach," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 6, pp. 1416–1427, Jun. 2019.

[34] H. Zheng and A. Louri, "An energy-efficient network-on-chip design using reinforcement learning," in *Proc. 56th Annu. Design Autom. Conf.*, Las Vegas, NV, USA, 2019, pp. 1–6, [Online]. Available: http://doi.acm.org/10.1145/3316781.3317768

[35] R. Jain, P. R. Panda, and S. Subramoney, "Cooperative multi-agent reinforcement learning-based co-optimization of cores, caches, and on-chip network," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 4, pp. 1–25, Nov. 2017. [Online]. Available: https://doi.org/10.1145/3132170

[36] Q. Fettes, M. Clark, R. Bunescu, A. Karanth, and A. Louri, "Dynamic voltage and frequency scaling in NoCs with supervised and reinforcement learning techniques," *IEEE Trans. Comput.*, vol. 68, no. 3, pp. 375–389, Mar. 2019.

[37] K. Wang, A. Louri, A. Karanth, and R. Bunescu, "IntelliNoC: A holistic design framework for energy-efficient and reliable on-chip communication for manycores," in *Proc. 46th Int. Symp. Comput. Archit.*, Phoenix, AZ, USA, Jun. 2019, pp. 589–600. [Online]. Available: https://doi.org/10.1145/3307650.3322274

[38] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," *ACM SIGARCH Comput. Archit. News*, vol. 36, no. 3, pp. 39–50, Jun. 2008. [Online]. Available: https://doi.org/10.1145/1394608.1382172

[39] Y. Xiang, J. Meng, and D. Ma, "A *Q*-routing based self-regulated routing scheme for network-on-chip," in *Proc. IEEE 9th Int. Conf. Commun. Softw. Netw. (ICCSN)*, Guangzhou, China, May 2017, pp. 177–181.

[40] M. A. Kinsy, S. Khadka, and M. Isakov, "PreNoc: Neural network based predictive routing for network-on-chip architectures," in *Proc. Great Lakes Symp. VLSI*, May 2017, pp. 65–70. [Online]. Available: https://doi.org/10.1145/3060403.3060406

[41] S.-C. Kao, C.-H. H. Yang, P.-Y. Chen, X. Ma, and T. Krishna, "Reinforcement learning based interconnection routing for adaptive traffic optimization," in *Proc. 13th IEEE/ACM Int. Symp. Netw. Chip*, Oct. 2019, pp. 1–2. [Online]. Available: https://doi.org/10.1145/3313231.3352369