J Supercomput (2014) 69:1491–1516
DOI 10.1007/s11227-014-1240-8

Optimizing memory access traffic via runtime thread migration for on-chip distributed memory systems

Weiwei Fu · Tianzhou Chen · Chao Wang · Li Liu

Published online: 24 June 2014

Abstract On-chip distributed memory system has become an attractive solution for massive parallel memory accesses found in future many-core processors. However, increasing number of on-chip cores and memory controllers inevitably introduce many remote memory accesses, which generate a large amount of on-chip traffic and put great pressure on the interconnection. This paper tries to optimize on-chip memory access traffic via runtime thread migration. We first analyze memory access behaviors in multi-threaded applications and find that the memory access targets and volumes are similar during short periods, which makes runtime prediction feasible. But the memory access targets exhibit great mobility during long periods, motivating us to dynamically move threads towards the data. Based on these observations, we propose a novel low-cost and distributed thread migration algorithm which adjusts thread placement in chains based on benefit estimation. We present details of the workflow, including the trigger and arbitration of migration requests and the procedures to determine the migration chains. Simulation results show that our algorithm achieves system performance speedup of 11.5 % and reduces average memory access latency by 11.0 %. It can find a few but effective thread migrations to optimize on-chip memory access traffic with acceptable hardware and runtime overheads.

Keywords Predictability · Thread migration · On-chip distributed memory system and runtime algorithm

W. Fu (✉) · T. Chen · C. Wang
College of Computer Science, Zhejiang University, Hangzhou 310027, Zhejiang, People's Republic of China
e-mail: dapan@zju.edu.cn

L. Liu
School of Information, Zhejiang Sci-Tech University, Hangzhou 310059, Zhejiang, People's Republic of China
e-mail: liuli@zstu.edu.cn

1 Introduction

On-chip distributed memory system (DMS) is considered as a promising memory organization for future many-core systems to hit the great memory wall [1], which exploits memory-level and communication-level parallelism deeply via integrating multiple distributed memory interfaces and interconnect components inside to deal with massive concurrent memory accesses. However, increasing number of on-chip cores and memory controllers (MCs) inevitably introduces many remote memory accesses and, therefore, a large amount of memory traffic. It puts great pressure on the on-chip interconnection, which may become another bottleneck of system performance.

The performance of memory access traffic is determined by many factors. In addition to the architecture and bandwidth supply of the on-chip interconnection, actual traffic patterns of memory accesses also play a crucial role, which is closely related to the mappings of tasks to cores and data pages to memory banks. Colocation of computation and data is a popular optimization scheme [2–4], which elaborates to shorten the paths to fetch data. If the positions of data pages are fixed, the placement of threads will determine the average distance of memory accesses. Thread migration is known as an efficient optimization method to improve the overall communication performance [5–7] since the changing behaviors of execution make static task mapping insufficient to guarantee high performance in all phases. If the system acquires some knowledge of future memory access patterns, it may let some threads migrate to cores which are closer to their dataset.

In this paper, we propose a novel runtime thread migration algorithm to optimize memory access traffic in on-chip manycore processors with distributed memory systems. The cores and memory controllers are interconnected by a network-on-chip (NoC) [8–10], which has emerged as a primary substrate for on-chip communication. Our mechanisms are based on runtime prediction of future memory access patterns. We first examine the predictability of memory accesses found in real applications from PARSEC [11] benchmark suite. We find that for most applications, the targets and volumes of memory accesses are similar between two continuous sample periods when the data pages are bank-interleaved. It implies that the behaviors of memory accesses for most applications are highly predictable. Besides, we observe great mobility of memory access targets during long periods of execution, which motivates us to dynamically move the threads towards the memory access center (MAC) node. Using simple information exchanging and benefit estimation schemes [6,7], our mechanisms are distributed and allow multiple threads to move together in chains, which can achieve a few profitable migrations to optimize overall memory traffic with acceptable overheads. We further discuss the workflow and architectural support of our algorithm. The hardware and runtime overheads are also characterized. Using cycle-accurate simulation, we show the effectiveness of the suggested mechanisms.

The remainder of this paper is organized as follows: We elaborate on the motivation for runtime thread migration in Sect. 2. The framework of our distributed chained thread migration and the detailed mechanisms are studied in Sect. 3. Section 4 shows some experimental results. We review related researches and make some comparisons with this work in Sect. 5 and finally Sect. 6 concludes this paper.

2 Motivation

2.1 Predictability of memory traffic

In on-chip manycore processors with distributed memory systems, the main purpose of thread migration is to shorten the distance between computation and data. Since the efforts to transmit memory requests and data to the destination are alleviated, the average transmission latency and link contention can be significantly reduced. However, thread migration introduces non-trivial overheads, which may incur performance degradation if the benefit of a migration is not significant enough. To make efficient migration decisions, we need sufficient and accurate knowledge of memory access targets and frequency in the near future. The positions of targets determine where to migrate the thread, and the frequency of accesses affects the benefits of migration. Profiles built beforehand can provide precise memory access traces [12], but they are very costly and sometimes impractical. Besides, if the datasets are changed, profiles should be regenerated even for the same application. We, therefore, focus on policies that rely on runtime estimation of memory access behaviors, which are adaptive and insensitive to the datasets. However, the accuracy of runtime estimation has a strong impact on the efficiency of the algorithm. Misprediction may introduce serious negative impacts.

History-based prediction is widely adopted in previous works [13]. In a simplest scheme, the execution of an application is divided into multiple sample periods, and the memory access pattern sampled in the last period becomes the predicted result of the next period. The principle of prediction lies in temporal and spatial locality of memory accesses during a short period. In order to verify the accuracy of the simple prediction scheme for various applications, we make some definitions to analyze the predictability of both memory access targets and volumes.

A memory access list of a thread i in a sample period p ($\mathrm{MAL}_{i,p}$) is a list which records the number of memory accesses falling on each memory interface during the period. If the elements in a MAL are normalized to the total memory access number, it is called a normalized memory access list (NMAL). A NMAL reflects the relative distribution of memory accesses towards different targets. If there is no memory accesses for a thread during a sample period, all the values of its NMAL are zero.

We define the similarity of memory access targets (ST) for two sample periods (a and b, b is after a in timeline) of an application to be

$$\mathrm{ST}(a,b) = \sum_{i \in S} \left(w_i \times \sum_{j \in T} \mathrm{Min}(\mathrm{NMAL}_{i,a}(j), \mathrm{NMAL}_{i,b}(j)) \right) \tag{1}$$

in which S is the set of threads in the application and T is the set of memory access targets, i.e. the memory interfaces involved. ST is the weighted sum of similarities of the memory access targets from all the threads. The weight for thread i (w_i) is the proportion of memory accesses from thread i compared with the overall number of memory accesses during the period. Apparently, ST is between 0 and 1. If the NMALs of all the threads in the two sample periods are identical, the ST value is 1.

Considering the spatial locality of memory accesses, adjacent memory interfaces may be frequently accessed during a short period. We extend the definition of ST which can accept a new parameter r (in number of hops) to introduce r-range ambiguity in calculating ST:

$$\mathrm{ST}_r(a,b) = \sum_{i \in S} \left(w_i \times \sum_{j \in T} \mathrm{Min} \left(\sum_{D(k,i) \le r} \mathrm{NMAL}_{k,a}(j), \mathrm{NMAL}_{i,b}(j) \right) \right), \quad (2)$$

where $D(k, i)$ is the distance between node k and the node where thread i is running. In (2), if the sum of relative memory access volumes falling inside the r-neighborhood of a certain node in the last period is equal to that in the next period, they are still considered similar. For example, assume that the NMALs of a single thread application in two periods A and B are (1, 0, 0, 0, 0, 0, 0, 0) and (0.5, 0.5, 0, 0, 0, 0, 0, 0) for a system with eight MCs; if the distance from MC-1 to MC-2 is one hop, the value of ST_0 is 0.5 while ST_1 is 1 in this scenario.

If a and b are adjacent periods, ST is used to compute the predictability of memory access targets. We define the r-hop predictability of memory access targets (PT_r) as the average ST_r of all adjacent sample periods. If r is 0, we call it strict predictability:

$$\mathrm{PT}_r = \sum_{(a,b) \in P} \mathrm{ST}_r(a,b) \quad (3)$$

in which P is the set of all the pairs of adjacent sample periods in the execution of an application. Using the above definitions, we analyze the predictability of real applications from PARSEC benchmark suite [11]. We target a 64-core tiled chip-multiprocessor (CMP) with 64 MCs distributed along with the cores. The tiles are interconnected by an 8 × 8 mesh network-on-chip. The other configurations will be shown in Sect. 4. We use overall number of memory accesses to divide the periods, i.e. a new period begins when the amount of memory accesses in the system reaches a threshold N, which is swept from 5,000 to 40,000. Three static data page mapping schemes called Random, Global Interleaving (GI) and Single Thread Interleaving (STI) are assumed in our experiments. In Random placement, each new data page is randomly mapped to a memory interface; GI allocates new pages to every memory bank globally in a round-robin fashion. The page placements of STI are also interleaved, but the round-robin allocation is performed by each single thread independently.

Figure 1a and b demonstrates the average strict and 1-hop predictabilities of memory access targets under the three mapping schemes. We find that the strict PTs under GI and STI are similar and higher than Random mapping. For most applications, the strict PTs are more than 0.55 under STI mapping. On average 60.3 % of memory target distribution in the next period is similar with that in the last period. 1-hop PTs are much higher than strict PTs, with the average being 0.81. For applications like blackscholes and streamcluster, their 1-hop predictability is more than 0.9, which illustrates very high similarity of memory target distributions between every adjacent periods if 1-hop ambiguity is introduced. It is a strong proof that the locality of memory accesses makes the targets predictable.

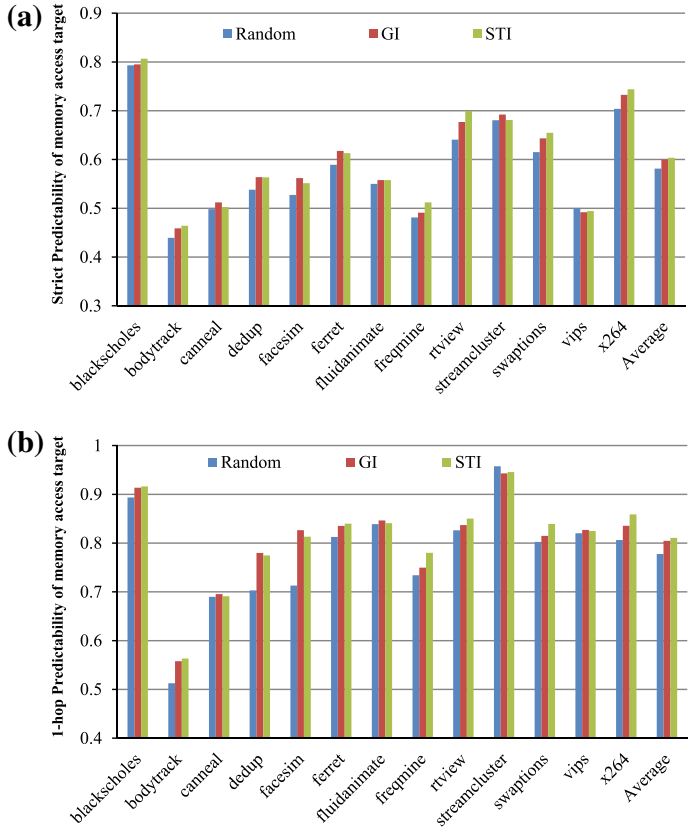

Fig. 1 **a** Average strict and **b** 1-hop predictability of memory access targets under Random, GI and STI data page mapping schemes

The granularity of partitioning periods is also important to the predictability. Figure 2 plots 1-hop predictability for ferret and streamcluster under STI mapping as the function of N. In ferret, the 1-hop PT under $N = 20,000$ is much higher than PT values under other granularities of sample period, necessitating careful selection of N to improve the accuracy of prediction. The case for streamcluster is very different, in which the predictability increases with the rise of N and gradually approximates to 1. It means that the distribution of memory access targets is getting statistically stable. Since very large N will shrink the benefit of optimization schemes, we choose $N = 15,000$ for streamcluster since it still achieves up to 0.95 predictability.

Next we analyze the predictability of memory access volume (PV), which may influence the reliability of benefit estimation in thread migration algorithms. The value of PV is defined as the weighted sum of similarity of memory access volumes for all the threads between every adjacent period:

$$\mathrm{PV} = \sum_{(a,b)\in P} \left(\sum_{i\in S} w_i \times \frac{\mathrm{Min}\left(\mathrm{NA}_i(a), \mathrm{NA}_i(b)\right)}{\mathrm{NA}_i(a)} \right) \Big/ (n-1) \tag{4}$$

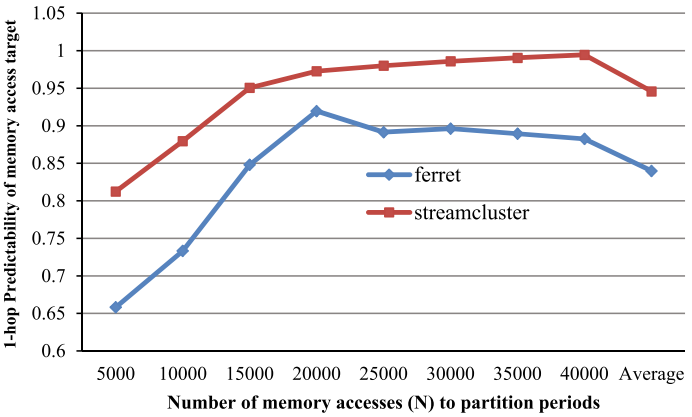

Fig. 2 1-hop predictability under STI mapping as the function of N for ferret and streamcluster

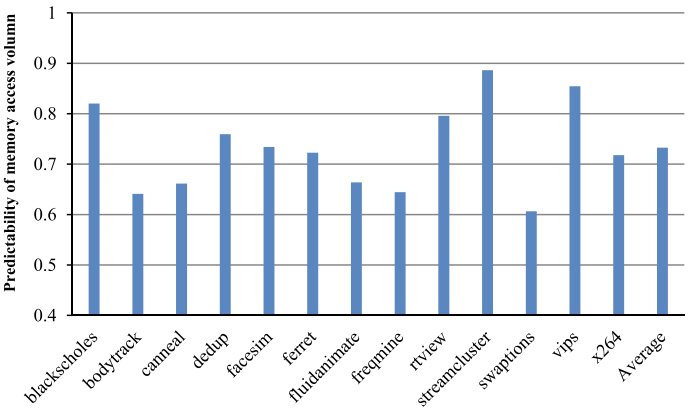

Fig. 3 Predictability of memory access volume

in which $NA_i(a)$ is the Number of memory Access of thread i during period a, n is the total number of sample periods during the execution. The above definition implies to what extent each thread can keep the access frequency of the last period. If there is a significant reduction of memory access volume, misprediction may occur and the benefit of the corresponding thread migration may be overestimated. The weight of frequency maintenance of each thread is also assigned as the proportion of memory access number during last period.

Figure 3 shows the predictability of memory access volume when N is 20,000. For most applications PV is over 0.7. For some applications like streamcluster and vips, the capability of memory access frequency maintenance is more than 0.85, which proves that the memory-intensive threads during last period will probably maintain the memory access frequency in these applications.

From the above results, we observe that the memory access behaviors in most applications exhibit high predictability for both targets and frequency. So the simple and widely used prediction scheme mentioned above is efficient and accurate enough.

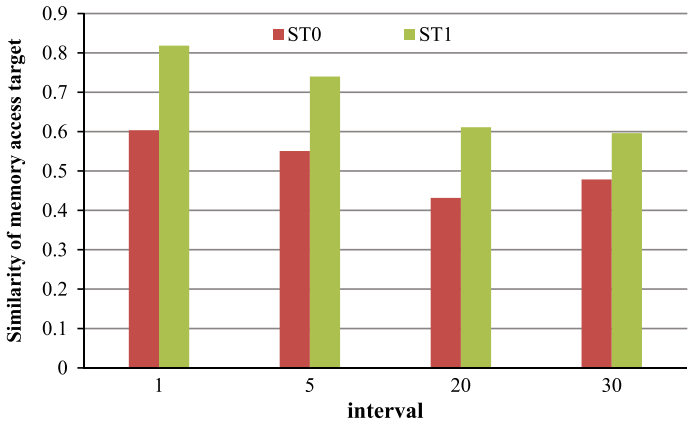

Fig. 4 1-hop similarity of memory access target between every pair of sample epochs with an interval of 5, 20 and 30 periods

Our thread migration algorithm proposed in this paper also employs runtime prediction to acquire knowledge of memory access patterns. The actual page placement and the granularity of partitioning periods also affect the predictability, so we should carefully decide these configurations to optimize memory access traffic. However, some applications such as bodytrack have relatively low predictability, which makes it unsuitable to apply prediction-based runtime mechanisms directly. Misprediction is more possible to occur in applications with low predictability, which may introduce serious performance penalty.

2.2 Mobility of memory access targets

We have witnessed high similarity of memory access behaviors between adjacent periods. But changing phases of execution may result in variation in memory access patterns. To prove this, we conduct the strict and 1-hop similarity of memory access target between every pair of sample epochs with an interval of 5, 20, 30 successive sample periods. We can see from Fig. 4 that the average ST_1 values significantly drop to 0.74, 0.61 and 0.60, respectively. In order to further explain the mobility of memory access targets, we define the MAC of a thread i in a 2D mesh distributed memory system as a 2D coordinate:

$$\mathrm{MAC}_i = \left(\sum_{j \in T} C_{jx} \times \mathrm{NMAL}_i(j), \sum_{j \in T} C_{jy} \times \mathrm{NMAL}_i(j) \right), \tag{5}$$

where T is the set of memory access targets, C_{jx} and C_{jy} are the x and y coordinates of the MC j. The definition is derived from least square method which refers that the total distance for accessing the data in its MAL is shortest when the thread is running at the MAC node. Note that the coordinate values of MAC are continuous number. We

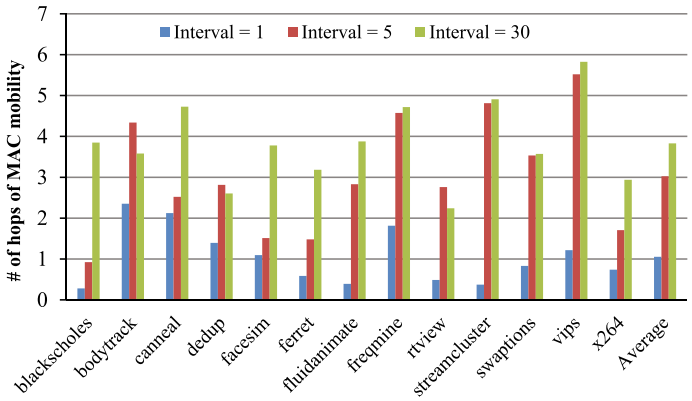

Fig. 5 The average mobility of MAC with intervals 1, 5 and 30

define the mobility of MAC as the weighted mean of Manhattan distances between the MACs for each thread of two epochs with a certain interval.

Figure 5 demonstrates the average mobility of MAC with the interval being 1 (adjacent periods), 5 and 30 (long periods). Since the similarity of memory access target for adjacent periods is high, the mobility is only 1 hop on average. For long periods, however, the changing application behaviors and datasets will cause remarkable MAC shift (3 hops during 5 periods and 3.8 hops during 30 periods on average). So static thread mapping is not sufficient to keep the average memory access distance low. It motivates us to use runtime thread migration to respond quickly to the mobility of memory access targets. The proposed thread migration algorithm will be shown in the next section.

3 Proposed runtime thread migration algorithm

3.1 Target system and framework of the mechanisms

Our target system is a CMP with on-chip distributed memory, as shown in Fig. 6. Memory banks and the corresponding MCs are distributed along with CMP cores. Each core with its private caches and a memory controller constitute a tile. The tiles are interconnected by a 2D mesh network-on-chip to deal with remote memory accesses. The mapping scheme of data pages is single thread interleaving (STI).

In this section we propose the framework of our novel chained thread migration (CTM) algorithm for the above system, which is shown in Fig. 7. We count the overall number of memory accesses in the system during the execution of threads. When it exceeds N, thread migrations can be activated and the counter and other statistics are cleared. We use the simple prediction scheme mentioned in Sect. 2 to conduct MALs for all the threads, which are the guidelines of our algorithm. The algorithm is distributed since each computation or memory node participates in the processes of thread migration in parallel, including the trigger, arbitration and final determination of thread migrations.

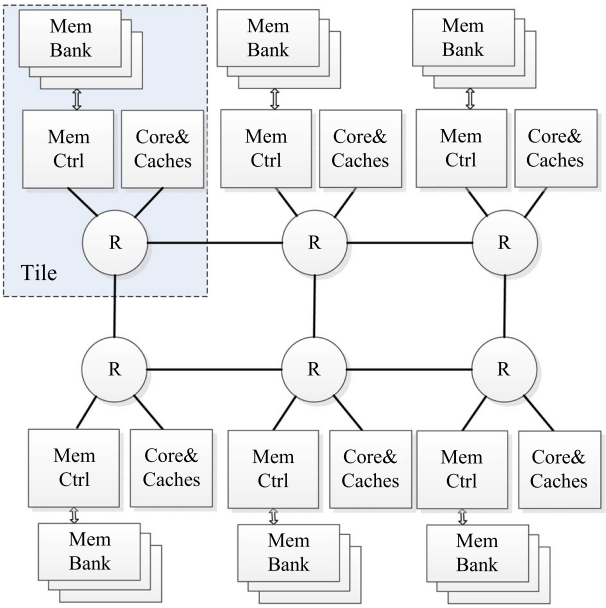

Fig. 6 Architecture of the target system

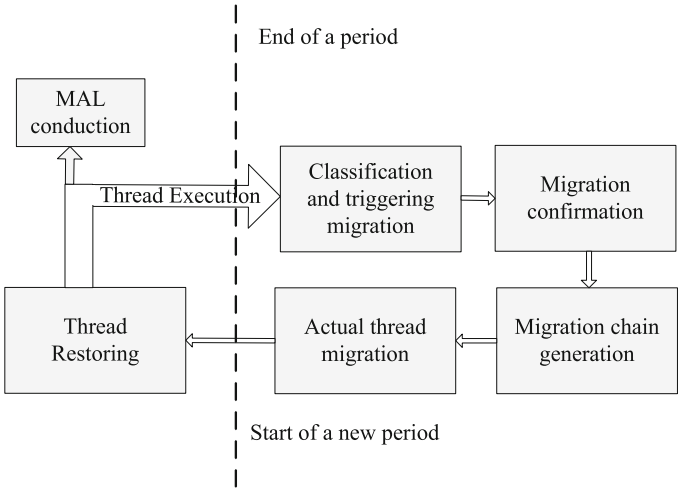

Fig. 7 Framework and basic workflow of our algorithm

Migration triggering: Each core with a thread running on it should decide whether to trigger a migration according to its MAL. If the thread wants to move, it should designate a group of nodes as the candidates of migrate destination. Migration requests as well as the benefit estimation should be sent to the target nodes.

Migration arbitration: We assume that the total number of threads is not more than the number of on-chip cores, and there is only a single thread running on each core. If a migration target receives multiple requests, or there has been a thread running

on it in the last period, it should arbitrate among the candidate threads to handle the conflicts.

Migration determination: When the migration requests are confirmed, each node should further assess the benefits and overheads of migration to decide whether to move the thread. In this paper, we use chained thread migration, in which a couple of threads may exchange their positions in chain. Simple exchanging algorithm proposed in previous works [6,7] is a special case of our algorithm, whose maximum chain length is 2. After the remappings of all the threads have been confirmed, i.e. the migration chains have been generated, actual thread migrations are activated then. The execution of a mobile thread can be restored after transferring its context to the destination. The details of the algorithm will be discussed next.

3.2 Classification of thread mobility

When a sample period terminates, each node should be classified into a certain state according to its MAL, to decide the mobility of the thread. The state will affect the trigger and confirmation of a thread migration. In our work we define the following five states:

IDLE: If there is no thread running on the core, the node is in IDLE state, which can be the destination of migration for another thread.

VOLUNTARY: The state VOLUNTARY refers that the thread running on the core is voluntary to migrate to any other position passively, to make room for other threads requesting the node. We suppose that if the number of memory accesses is below a threshold T_{minor}, the performance is insensitive to the memory traffic. So the thread can migrate to anywhere without introducing significant performance impacts:

$$\text{NA}_i < T_{\text{minor}} \tag{6}$$

Besides, we observe a scenario that the placement of data accessed by a thread during an epoch can be quite divergent, i.e. the thread issues many remote memory accesses to different memory interfaces throughout the network. In this case, a great amount of memory access traffic is unavoidable regardless of the thread's location. The state of the node is also VOLUNTARY, because moving the thread to other place has very small impact on the network performance. We use memory access radius (MAR) to indicate the degree of divergence of memory accesses:

$$\text{MAR}_i = \left(\sum_{j \in T} D(C_j, \text{MAC}_i) \times \text{NMAL}_i(j) \right), \tag{7}$$

which is the average Manhattan distance to the MAC of thread i and also the minimum average distance to access the data. We define a circle of memory access (CMA) as a circle with the center being MAC and the radius being MAR. If the MAR is larger than a threshold T_{div}, the memory access targets are divergent enough and the state is VOLUNTARY, as shown in Fig. 8a.

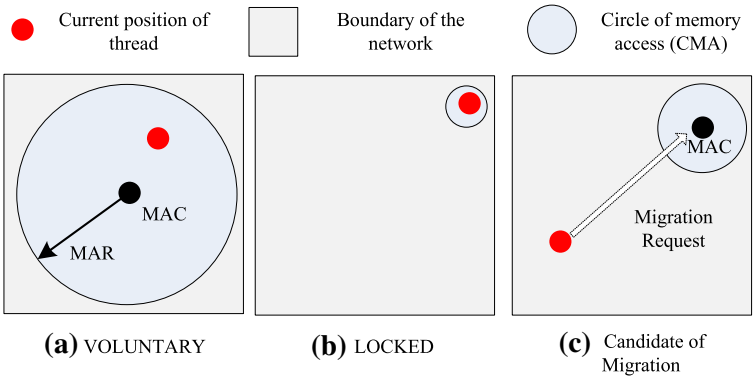

Fig. 8 Classification of thread mobility

$$\mathrm{MAR}_i > T_{\mathrm{div}} \tag{8}$$

If (6) or (8) is satisfied, the thread on the core must be expelled to any particular node without concerning the benefits or overheads when the core is required by other threads.

LOCKED: If the thread running on the core in last epoch is very sensitive to its placement, and passive migration may dramatically lengthen the average access distance, the state is set as LOCKED, which means that the current thread will not migrate and the node cannot be the destination of any migrations. In our algorithm, if the current thread has a large memory access volume, very concentrated memory access targets and its location is inside the CMA, the node should be locked, just as the case illustrated in Fig. 8b. If the thread migrates outside the CMA, the distance of memory accesses will rise rapidly, leading to performance degradation. Formally,

$$\mathrm{NA}_i \geq T_{\mathrm{major}} \tag{9}$$

$$\mathrm{MAR}_i \leq T_{\mathrm{con}} \tag{10}$$

$$D(i, \mathrm{MAC}_i) \leq \mathrm{MAR}_i \tag{11}$$

If (9)–(11) are satisfied, we fix the current thread on the node.

CANDIDATE of migration (CM): when there are plenty of memory accesses and the MAC is far away from the current location of the thread, the thread i can be the candidate for migration. Formally, if both (9) and

$$D(i, \mathrm{MAC}_i) - \mathrm{MAR}_i > T_{\mathrm{far}} \tag{12}$$

are satisfied, the state of the node is CM, which will trigger a migration request to the nodes next to its MAC, as shown in Fig. 8c. Since the coordinates of MACs are continuous values, e.g. (3.35, 4.12), we choose several nearby nodes from (3,4), (3,5), (4,4) and (4,5) as the candidate targets and sort them by their distances to MAC in ascending order. The CM node should send a migration request to the first target. The arbitration and benefit assess schemes will be discussed later.

NORMAL: If a node does not belong to any of the above categories, its state is NORMAL. The thread on it will not trigger an active migration, but it is sensitive to passive migrations. If the node is designated as a target of migration, it should carefully search for another node to place the old thread, according to the actual benefits and migration overheads.

3.3 Workflow of chained thread migration

When the state of each node is determined, the CM nodes send migration requests to their destinations, which is the beginning of our CTM algorithm. We employ a global clock to synchronize each step in the processes of the algorithm, which will be detailed as follows:

Benefit estimation: The benefits of a thread migration lie in (1) shortening average transmission distance and, therefore, a reduction in memory access latency, and (2) lower load on the interconnect links and routers between the original core and its MAC, which will diminish the possibility of traffic contention and improve the overall network performance. We employ a novel distributed mechanism to estimate the benefits. When the message of a migration request sent by each CM node travels along the NoC routers and links to the migration targets, it estimates the migration benefits at each hop along the way. The benefit of a migration towards the MAC by one hop is supposed to be the saving of transmission latency at the hop, i.e. the latency of link traversal, router pipeline stages and queuing (due to contention). The benefits are conducted and accumulated at each hop until reaching the destination to contend for the opportunity of migration. Since the memory access traffic is bi-directional (we assume that the memory request and response messages travel along the same route towards opposite directions [14]), we should consider the loads on both of the opposite output ports of each router. For example, a thread migration message penetrates a router from the west port to east; we should reckon in the average hop latency of flits destined for both west and east output ports, because the memory traffic generated by the thread must experience these two ports and the corresponding links during last epoch. If the thread has migrated through the router, the efforts to transfer packets through the router in both directions can be avoided. So we conducted the saved latency (TL) as the benefit value. Besides, since each hop of thread migration introduces additional traffic, we should deduct the firm overhead G from the benefit value of every hop. We assume that the traffic latency after migration is MAR $\times\, t$, where t is the router pipeline latency. So the final benefit function is

$$\mathrm{BN} = \mathrm{NA} \times \left(\sum_{h \in R} \mathrm{TL}_h - \mathrm{MAR} \times t \right) - \sum_{h \in R} G, \tag{13}$$

where R is a collection of all the nodes along the route and NA is the number of accesses for the thread. Many previous works [6,7] count transmission hops as the benefit function, which is simple to implement. But they fail to consider the traffic contention in the benefit assessments. Consider the scenario where both nodes A and B are CM nodes and the threads want to migrate to an IDLE node C in the very

center between A and B. Assume that both threads only access data in C for equal times, but the load on the interconnection between C and A is much heavier than that between C and B. In hop-based scheme, the estimated benefits of the migrations are the same. However, the benefits of B-to-C migration should be much greater than A-to-C migration because the contention on the B-to-C links can be avoided. Node B should be the winner of arbitration. Our latency-based estimation scheme will be more feasible, since the hop latency values will reflect the relative intensity of interconnection load.

Request confirmation: when a target node receives some migration requests, and its state is not LOCKED, it must arbitrate from the requests (R_1, R_2, ...) and choose one with the greatest benefits, which satisfy

$$\mathrm{Max}(\mathrm{BN}(R_1), \mathrm{BN}(R_2), \ldots) \geq T_{\mathrm{bn}} \tag{14}$$

as the winner for request confirmation. The requests which are not confirmed should be forwarded to their next targets (note that each migration request message contains 2–4 candidate targets) for next-round arbitration until all the migration requests in the system have been confirmed or rejected. It is a simple process which is also distributed and can be finished quickly under the coordination of the global clock. Once a migration is confirmed, the target node sends back an ACK signal to inform the source core of a migration acknowledgement. The requestor should start preparing for thread migration by halting the execution, packetizing the context, etc. CM nodes that fail to acquire any confirmation will receive an NACK signal and switch their states to NORMAL.

Chained thread migration: After all the requests have been confirmed or rejected, we run our chained thread migration algorithm to determine the migrations. The basic principle is shown in Fig. 9. A thread running on node A in last period requests to migrate to node B, and the state of node B is also CM, whose current thread has to migrate to node C. An A–B–C migration chain has been generated. We assume that node A is the head of the chain, i.e. node A is not the migration destination of any other nodes, and node C is the tail of the chain (unless the chain is circular). The migration chain can have the following patterns.

1. If the state of node C is also CM and the thread wants to migrate to A, a circular chain (A–B–C–A) has been formed (Fig. 9a). The migrations in chain will not affect other threads. Since each section of the migration chain has sufficient positive benefits, the migrations will be estimated effective, which can be performed without further consideration.
2. If node C is IDLE. No conflicts occur at the tail. So the migrations of (A–B–C) can be activated immediately (Fig. 9b).
3. If node C's state is VOLUNTARY, i.e. the thread on node C must migrate passively to any other place. In our algorithm, we move it back to the head (node A) to avoid conflicts between different chains. The chained migrations (A–B–C–A) can be determined without hesitation, because the impact of passive migration from C to A is minor (Fig. 9c).
4. If the state of the tail is NORMAL, it should be the most complicated condition. The thread on node C must migrate to make room for the thread on node B, but

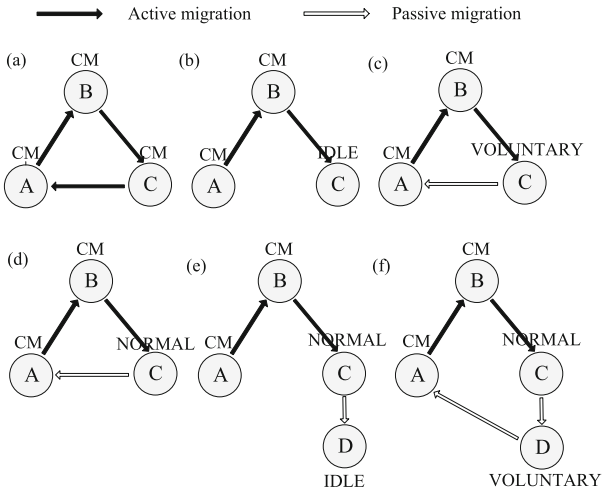

Fig. 9 Chained thread migration

the destination should be carefully chosen. Migration back to A is an option since a circular migration chain is simple and contention-free, but we must conduct the benefit of the passive reverse migration (from tail node C to head node A), which is the difference of the estimated route latency minus the migration cost:

$$\mathrm{BN}_{\mathrm{reverse}} = \sum_{j \in T} (\mathrm{MAL}_i(j) \times (D(C, j) - D(A, j)) \times t) - G \times D(C, A) \quad (15)$$

We should also accumulate the positive benefits of active migrations along the chain to calculate the total benefits of the circular migration chain:

$$\mathrm{BN}_{\mathrm{circular}} = \sum \mathrm{BN}_{\mathrm{chain}} + \mathrm{BN}_{\mathrm{reverse}} \quad (16)$$

If the total benefits satisfy

$$\mathrm{BN}_{\mathrm{circular}} \geq T_{\mathrm{bn}} \times m \quad (17)$$

the migration chain (A–B–C–A) is determined (Fig. 9d). Here m is the length of the chain. Otherwise, we should find another node outside the chain to place the thread. We consider the nodes next to C because moving a thread to a core nearby has a small impact on performance and introduces minimum costs. So node C distributes signals to search for a nearest node D that is IDLE or VOLUNTARY to form an A–B–C–D (Fig. 9e) or A–B–C–D–A (Fig. 9f) chain, respectively.

If the migration chain has been determined, the nodes with a new immigration switch their states to LOCKED, and the node A should be switched to IDLE if the chain is not circular. Since multiple independent migration chains can be generated concurrently, the procedures of chained migration are highly parallel.

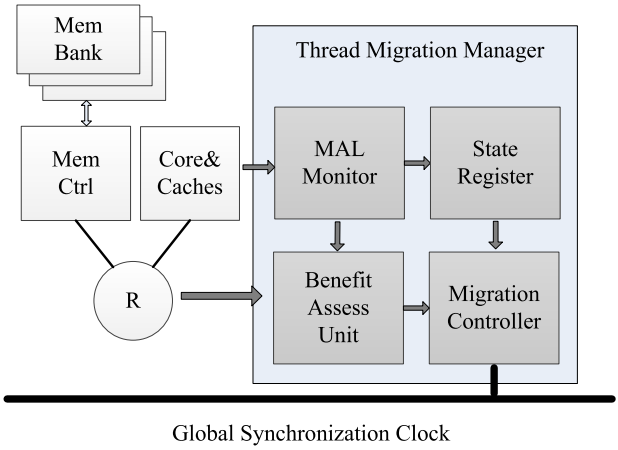

Fig. 10 Architecture of thread migration manager

Actual thread migration: The above algorithm has decided the mapping of each thread. Then the thread labeled to migrate should launch actual migration processes. As mentioned, some preparations can be made once it is determined to move. The thread migration procedure has been studied in many previous works, which is beyond the scope of this paper.

3.4 Architecture support and hardware overhead analysis

In this section we discuss the architecture support for our runtime algorithms and analyze the hardware overhead briefly. We add a simple module, called thread migration manager (TMM), in each tile, which conducts information from the tile and works in concert with other TMMs to guide the migrations. The basic structure of TMM is shown in Fig. 10, which is comprised of the following sub-modules.

MAL monitor: the MAL monitor contains a table to record MAL information of the thread running on the core. It monitors each memory request message to count the number of memory accesses (NA) towards every memory interface. In a CMP with 64 memory controllers, we assume $N = 20,000$ and use 10 bits to record NA towards each MC, necessitating 640 bits of table storage. Actually, we may only conduct NA of the most significantly accessed MCs, because many MCs are seldom accessed during a sample period by a thread, which will not affect the accuracy of prediction. So we may further compress the storage space.

State register and calculation logic: we need state calculation logic and a register to keep the state of each node and trigger migrations. The logic is very simple with negligible overhead.

Benefit assess unit: We employ this sub-module to estimate the benefits and overheads of thread migration. As mentioned in Sect. 3.3, it should conduct the average flit latency at the NoC router as the benefit value, which will be transferred and accumulated from the source to the migration target.

Migration controller: the migration controller is the key component of TMM, which takes charge of running the multiple procedures of our algorithm discussed above. The migration controllers are interconnected by a global synchronization clock to keep in coordination.

We can see that our TMM modules are compact units with low computational cost and storage space. So our mechanisms will not bring excessive hardware overheads.

3.5 Runtime overhead analysis

The thread migration process is similar to that of a context switch, which introduces non-trivial runtime overhead. The processes of thread halting, context packing and transferring, and execution recovery significantly affect the runtime performance. After being confirmed for migration, the core should halt current processing operations and flush the processor pipeline. The process control block needs to be packetized and transmitted across the network. The size of the context to be transferred is typically 256 KB [6,15], which takes about 30–50 cycles to reach the destination through NoC. Once the context has been transferred, the target core can resume nominal operation within hundreds of cycles.

The above runtime overheads are inevitable. In our work, we try to reduce runtime overhead of thread migration by exploiting the intrinsic parallelism of the algorithm. First, we activate the preparation of thread migration before the final determination of the migration chain to hide latency. Even for the tail node, it can switch the context beforehand for passive migration without knowing the actual migration target. Second, the migration mechanisms such as protocol message exchanging, benefit calculating, etc. are fully parallel and distributed. We deploy multiple low-cost TMMs to enable thread migration, which fit in well with the distributed nature of the NoC and DMS. In comparison, centralized algorithms such as GA remapping [16] require a central node (called global manager) to gather information from all over the system, compute for an optimal remapping and distribute the decisions to all the nodes. The computational and traffic overheads are very large, which hurts the scalability of the mechanisms. Besides, GA remapping may result in too many migrations at every epoch to achieve the minimum memory access distance. But some migrations may have very little benefit while introducing large runtime overhead. In our work, we run distributed benefit estimation for each migration candidate and permit migrations with sufficient benefits only. In this way we perform a few but beneficial migrations to minimize runtime overheads.

4 Evaluation

4.1 Methodology

We target a 64-core chip-multiprocessor (CMP) with 64 MCs distributed along with the cores, as described in Sect. 3.1. We assume 64 DRAM DIMMs with four memory banks each. Each DIMM is connected to a memory controller via a single channel, with a memory capacity of 16 MB to reach a total of 1 GB main memory. The cores

and MCs are interconnected by an 8 × 8 2D mesh network-on-chip. Each core has private 4 KB L1s (2 ways) and 16 KB L2 (8 ways) caches. Open-page row-buffer policy is assumed in our DRAM modules. We use three-stage wormhole routers in NoC and FR-FCFS scheduling [17–19] policy for memory arbitration.

We use a C++ based cycle-accurate trace-driven simulator to evaluate the performance of our mechanisms. We faithfully model the MCs, DRAM modules and the NoC, simulating the transmission and buffering of memory requests, data and other necessary messages. Scheduling of the MCs, contentions on memory buses and banks, and timing constraints of DDR3 DRAM are also involved for simulation, according to the Micron datasheet [20]. Memory access traces from Gem5 full-system simulator [21] executing applications from PARSEC [11] with sim-large dataset are conducted to feed our simulator. The initial mappings of both threads and data are the same for all the scenarios. We randomly map the threads to the cores at first and place the data pages to the memory banks using single thread interleaving (STI) scheme.

We carefully choose the sample period N according to the predictability, which is discussed in Sect. 2.1. We conduct the MALs of each thread as the guidelines of thread migration. The set of thresholds (T_{minor}, T_{major}, T_{div}, T_{con}, T_{far}) assumed in our algorithm is set as ($N/250$, $N/50$, 4, 1.5, 3). The thresholds of benefit (T_{bn}) of different applications are set case by case in our simulation. We model the runtime overheads of thread migration mentioned in Sect. 3.5. The additional traffic introduced by our CTM algorithm is also simulated, including the transmission of migration requests, responses and the contexts of mobile threads.

4.2 Results

In this section we show performance results of the memory system with our CTM mechanisms, which are compared with static thread placement (no migration, "None" in the following results) and other thread migration schemes. First we show results of average latency breakdowns of accessing memory in Fig. 11. We break the total memory access latency into three parts: network transmission latency, memory read/write latency and MC queuing latency. Our thread migration mechanisms can reduce transmission latency by 25.4 %, but have almost no impact on read/write latency and queuing latency. This is because thread migration schemes focus on improving memory traffic, which will not change the loads and read/write performance of memory modules. For swaptions, MC queuing latency dominates due to load imbalance on different memory interfaces, which makes the effect of thread migration negligible (only 2.3 % reduction of total latency). However, for most applications, the latency of transferring memory request/data is the main contributor to the total latency. It shows that the on-chip interconnection will be one of the bottlenecks of memory access performance in DMSs and our thread migration scheme can reduce total latency by 11.0 %.

Next we compare our CTM algorithm with other two algorithms: Simple Exchange (SE) and Genetic Algorithm (GA), which are derived from the similar task migration schemes proposed in our previous work [7]. In SE, the five most memory-intensive threads directly migrate to their MAC nodes and exchange the positions with the

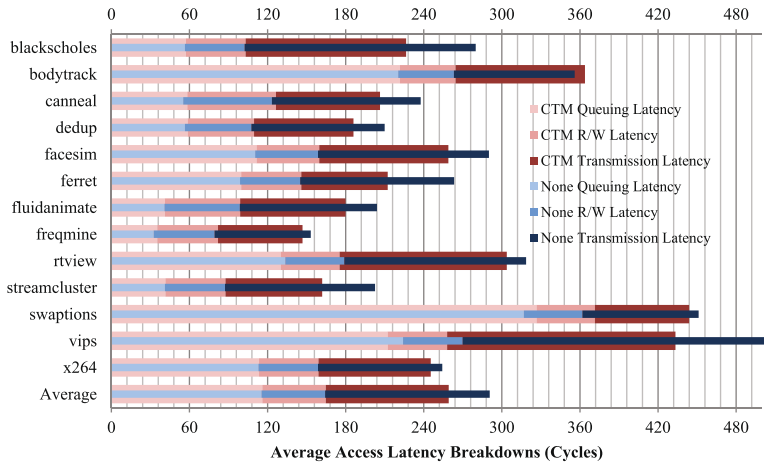

Fig. 11 Average latency breakdowns of accessing memory

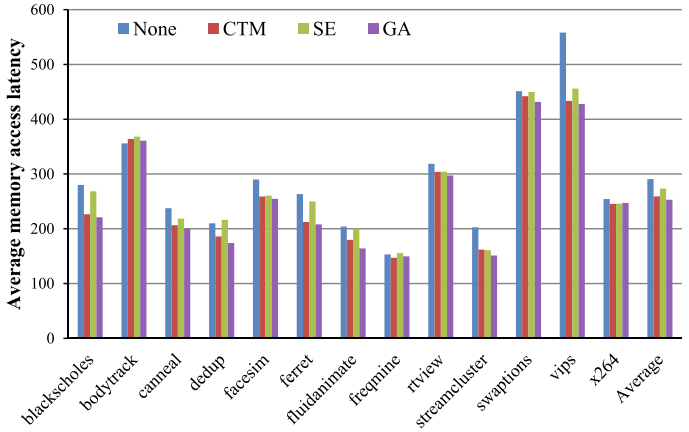

Fig. 12 Comparisons of average memory access latency

current threads on the MACs. We also make some modifications to GA remapping algorithm [16], with the fitness function being the reciprocal of average hops. Multiple iterations are experienced to reach correct convergence, to minimize average memory access distance. We evaluate the average memory access latency and IPC speedup of various thread migration algorithms, compared with the system with immobile thread placements. The results are shown in Figs. 12 and 13. We find that CTM, SE and GA improve latency by 11.0, 7.1 and 13.1 %, and achieve IPC speedup by 11.5, 6.4 and 8.9 % on average, respectively. For SE algorithm, the improvements are less than our CTM scheme, because SE only considers the benefits of the initiative migrations and ignores the effects of the reverse ones. Some passive migrations may bring negative impacts on the reciprocal performance. Besides, a thread with massive but divergent memory accesses will acquire very limited benefits from the migration to its MAC, which may also become a new hotspot. For GA algorithm, although multiple

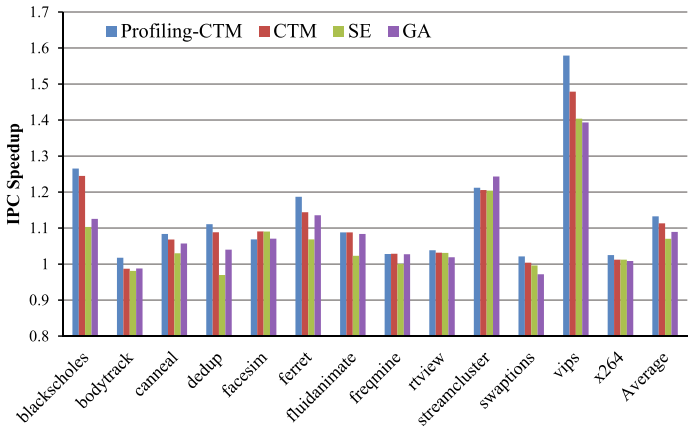

Fig. 13 Comparisons of average IPC speedups

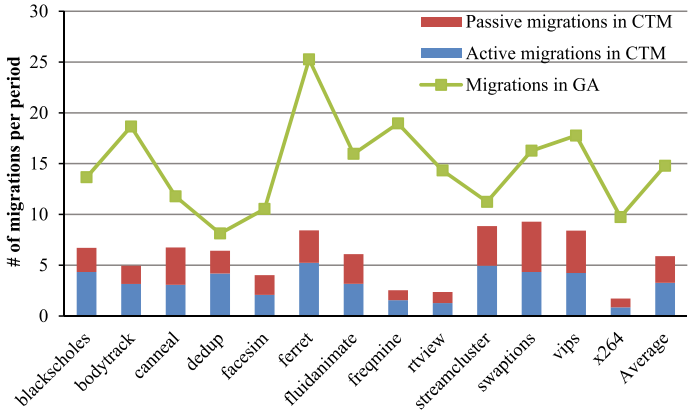

Fig. 14 Number of migrations of GA and CTM per sample period

iterations of optimization can achieve minimum access latency, the IPC speedup is a bit lower than CTM (about 2.6 %). GA has been a centralized algorithm which requires a global manager (located in the center of the network in our evaluation) to gather MAL information, compute the mapping of every thread and distribute the determinations. It introduces a great amount of extra traffic and creates a network hotspot in the center. The processes are serial and slow. Furthermore, we compare the average number of migrations in each period between GA and CTM in Fig. 14 and find that GA requires much more thread migrations to complete the remapping processes which introduces non-trivial runtime overheads. In our CTM algorithm, however, on average only 5.8 migrations are elaborated in each period to achieve considerate performance promotion. It proves that our mechanisms can optimize overall memory traffic through a few efficient thread migration, while some excessive migrations in GA are not beneficial enough to compensate for the runtime overheads. Most of the migrations found in CTM are active migrations, and the other 44 % passive migrations

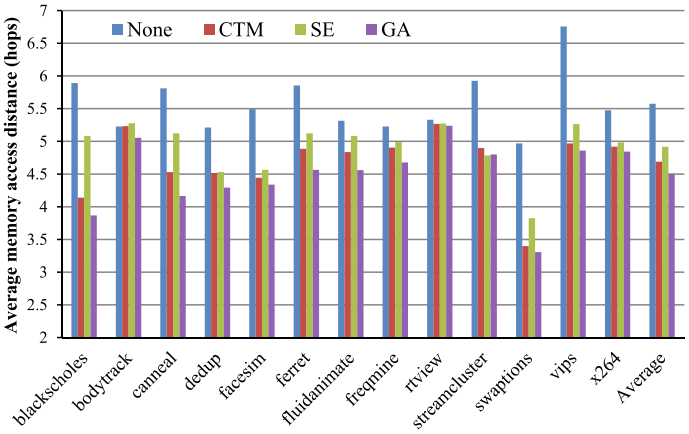

Fig. 15 Comparisons of average memory distance

must experience strict estimation of overheads to minimize the negative impacts from them. Besides, as discussed in Sect. 3.5, the computation efforts of our CTM algorithm are much lower than GA due to its distributed and parallel features.

However, for bodytrack, we observe performance degradation for all the runtime algorithms, which mainly results from misprediction. As discussed in Sect. 2, the predictability of bodytrack is relatively low. The prediction is unreliable to serve as the guidelines of runtime algorithms, which leads to serious deviation in benefit/overhead estimation. We evaluate a profiling-based CTM algorithm in which the MALs are precisely profiled to eliminate the effect of misprediction. We find in Fig. 13 that the performance are better than runtime schemes for most applications, including bodytrack. In spite of this, our runtime CTM algorithm still achieves up to 97 % performance on average due to high predictability in most benchmarks. For applications with low predictability, we must employ other schemes such as profiling to acquire an accurate knowledge of future memory access behaviors in order to avoid performance penalty caused by misprediction.

We then evaluate the impact of the three algorithms on the two important NoC performance indicators: memory access distance and average router latency, which are shown in Figs. 15 and 16. GA is dedicated in minimizing average memory access hops after massive migrations, while CTM takes advantages of a small number of thread migrations to shorten the average hop count by 0.89 hops and achieves the least hop latency. As discussed in Sect. 3, we use hop latency, rather than hop count to estimate benefits. Our algorithm can distinguish transmission links with different loads and prioritize migration requests which can significantly reduce link contention. SE has the least promotion of both memory access hops and router latency due to the limited benefits of active migrations and disadvantages of reverse ones. Then we use ORION [22] to estimate the values of dynamic power dissipation in the interconnection, which are normalized and shown in Fig. 17. CTM, SE and GA save dynamic power by 19.9, 15.4 and 22.8 %, respectively. Obviously, we can significantly reduce power consumption via thread migration, since the numbers of buffering, arbitration, switching and link traversals are remarkably decreased.

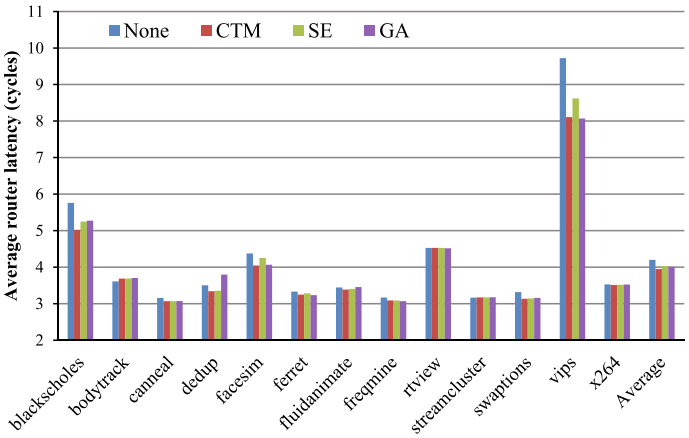

Fig. 16 Comparisons of average router latency

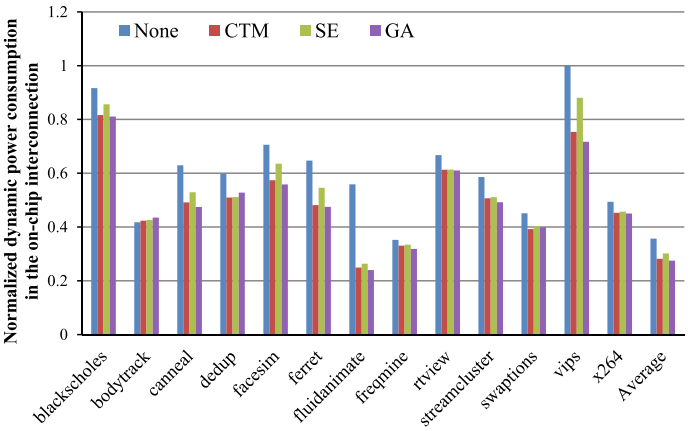

Fig. 17 Comparisons of dynamic power on the interconnection

Finally, we analyze the impact of granularity of dividing the periods (N). We plot the IPC speedup and average memory access latency as the function of N for two typical applications: ferret and streamcluster in Fig. 18. In ferret, if the length of each period is too short, the collection of MALs is insufficient to predict future behavior (since the predictability shown in Fig. 2 is low when N is small). Misprediction may occur and significantly affect the performance of our runtime algorithm. For streamcluster, although it is also predictable when $N = 5{,}000$, frequent migrations introduce large overheads and shrink the benefits of every thread migration. When N increases to 15,000, ferret has the highest predictability and also the best performance. The benefits of our algorithm in both ferret and streamcluster decrease sharply when the periods of migrations are too long, since many opportunities of optimization are missed. So we choose $N = 20{,}000$ for streamcluster, which achieves very high predictability and the largest performance improvement. Above all, we should carefully choose the granularity of partitioning periods with comprehensive considerations of predictabil-

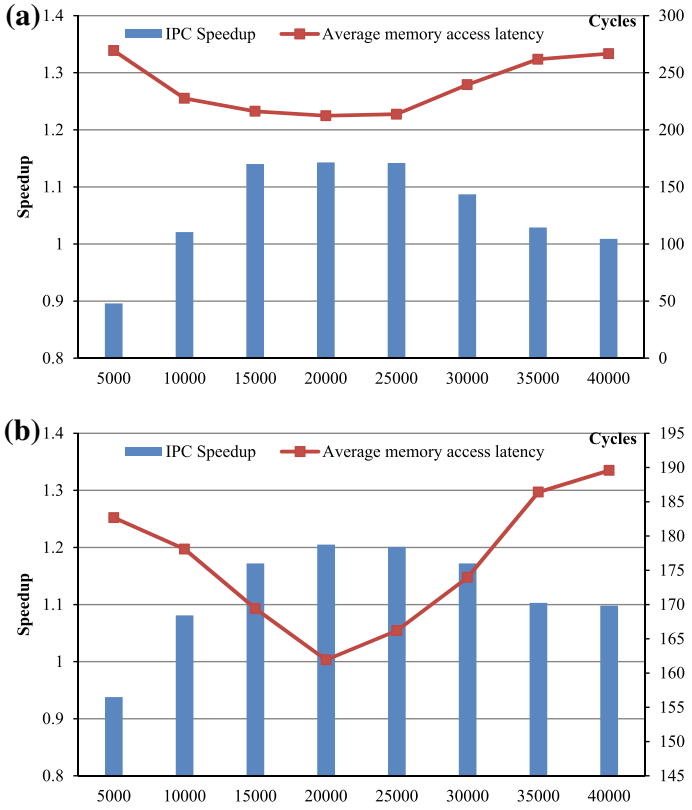

Fig. 18 IPC speedup and average memory access latency as the function of N for **a** ferret and **b** streamcluster

ity, runtime overheads of migrations and reaction time for the ever changing memory access behaviors.

5 Related work

5.1 Task/thread migration

Runtime task migration methods have been studied in various scenarios for years. Chen [5] proposed hybrid task migration schemes in 2D mesh wormhole-routed multicomputers to minimize transmission latency. The algorithms share some similarities with our work, but the mechanisms in multi-computer systems cannot be directly applied in NoC-based systems, which are featured with limited chip area and computational capability. The authors in [23] proposed remote core locking mechanism aiming to improve the performance of critical sections in legacy applications on multicore architectures. In [24] the authors proposed a task management infrastructure that is well suited for the constraints of single chip multiprocessors with distributed operating systems. User-managed migration scheme based on code check-pointing

is assumed to keep load balance and the overheads of migration are characterized. Task migration in NoC systems were also well studied, which mainly focused on specific migration mechanisms [25], including hardware support [26] and mechanisms to alleviate negative impact caused by migration [27]. In our recent work [28], an online task mapping algorithm is proposed for NoC-based systems, which optimizes task mapping/remapping to reduce communication energy consumption according to runtime communication status. The authors in [29] described an execution migration mechanism, called EM2, in which threads always migrate to the core where data are statically stored. Misler et al. [6] proposed Moths, a distributed runtime task migration mechanism for NoC platform. The mechanism is based on traffic volume monitoring between every pair of nodes. Simple exchanging of only one pair of threads is assumed during each epoch, and the algorithm complexity for searching the pair is very high. In this work, we avoid to employ global benefit competition algorithms like [6] and significantly reduce computing complexity, while allowing multiple threads to move together in chains. Most of these researches tried to directly optimize NoC traffic, and few of the previous works focused on memory access traffic found in modern on-chip distributed memory systems.

Many approaches employ application behavior prediction to guide task migration or cache line migration. In [30] the authors proposed approaches to increase the timing predictability in multicore architectures for task migration in embedded systems. The scheme shows increased predictability in the presence of cross-core migration. Other works on the predictability of migration focused on the migration overhead and cache-related behaviors [31,32], but not the memory access or network behavior itself. Our previous work [7] analyzed the predictability of communication volume between every pair of nodes and proposed several simple thread migration schemes based on Genetic Algorithm, Simple Exchange and Benefit Assess. The source of packets can be migrated closer to its destination. However, the process of information gathering and migration are based on approaches that only distinguish different network node, neglecting the type of packet source. Actually, in NoC-based DMSs the source can be the core, cache controller or the memory controller, while task migration only affects packets which are sent from the core (such as memory request and write data messages). The algorithms in [7] may incur misprediction due to confusion of packet types. Our work can avoid such confusion by employing memory level investigation, since the memory access pattern directly determines memory traffic in NoC.

5.2 Optimization of memory access traffic

Most prior researches on on-chip communication are concentrated in processor-to-processor traffic and very few work discussed the influence of processor-to-memory traffic. Abts [14] examined the placement of MCs in NoC to improve performance predictability. Bakhoda et al. [33] proposed hardware optimizations in NoC routers which handle many-to-few-to-many memory traffic appeared in many-core accelerators like GPGPUs. In [34], Kim proposed a communication-aware MC design to deal with network congestion in which transactions waiting in the DRAM queue that have to be sent back to cores in areas of the NoC which are less congested are given

higher priority. Memory-centric NoC architectures with real chip implementations are studied in [35]. Recent work [36] tried to reduce end-to-end memory access latency in NoC-based multicore system by prioritizing memory response packets and messages destined for idle memory banks. Memory placement mechanisms in NUMA systems such as page migration [37–39], interleaving [2] and replication [2,40] have been studied for decades. In [41] the authors proposed a mechanism to deal with resource contention in NUMA systems, memory migration strategies are studied in the work, which are the necessary part of the NUMA contention-aware scheduling algorithm. They tried to reduce the distance between computation and data via moving the data, while our work achieves the same goal by moving the computation closer to the data.

6 Conclusion

This paper proposed a runtime thread migration algorithm for CMPs with on-chip distributed memory system to optimize memory access traffic. We found that for most applications, the memory access targets and volumes are similar in short periods, which enable us to predict the memory access pattern of the next sample period. Besides, mobility of MAC during long period motivates us to move the threads towards the MAC dynamically. A distributed chained thread migration algorithm was studied then, including the processes of node classification, migration trigger and arbitration, benefit estimation and the final determination. We further generally discussed architecture support and overheads of the algorithm. Simulation results show that the mechanisms presented here can reduce transmission latency and increase system performance with acceptable overheads. For future work, we will study algorithms to support multi-threading systems and migration schemes based on affinity of memory accesses between threads. We will also focus on the topic of high-level (compiler, language, application levels) analysis of memory access behaviors and imbalanced computational workloads.

Acknowledgments This paper is supported by the National Natural Science Foundation of China under Grant No. 61379035, the National Natural Science Foundation of Zhejiang Province No. LY14F020005, Open Fund of Mobile Network Application Technology Key Laboratory of Zhejiang Province, Innovation Group of New Generation of Mobile Internet Software and Services of Zhejiang Province.

References

1. Wulf WA, McKee SA (1995) Hitting the memory wall: implications of the obvious. ACM SIGARCH Comput Archit News 23(1):20–24
2. Dashti M, Fedorova A, Funston J, Gaud F, Lachaize R, Lepers B, and Roth M (2013) Traffic management: a holistic approach to memory placement on numa systems. In: Proceedings of the 18th international conference on architectural support for programming languages and operating systems, pp 381–394, ACM
3. Kamali A (2010) Sharing aware scheduling on multicore systems. Applied Science, School of Computing Science, USA
4. Tam D, Azimi R, Stumm M (2007) Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. ACM SIGOPS Oper Syst Rev 41(3):47–58
5. Chen TS (1999) Task migration in 2D wormhole-routed mesh multicomputers. In: High performance computing. Springer, Berlin, pp 354–362

6. Misler M, Jerger NE (2013) Moths: mobile threads for on-chip networks. ACM Trans Embed Comput Syst (TECS) 12(1s):56
7. Wang C, Yu L, Liu L, Chen T (2012) Packet triggered prediction based task migration for network-on-chip. In: 20th IEEE Euromicro international conference on parallel, distributed and network-based processing (PDP), pp 491–498
8. Dally WJ, Towles B (2001) Route packets, not wires: on-chip interconnection networks. In: Proceedings of the IEEE design automation conference, pp 684–689
9. Benini L, De Micheli G (2002) Networks on chips: a new SoC paradigm. Computer 35(1):70–78
10. Dally WJ, Towles BP (2004) Principles and practices of interconnection networks. Access online via Elsevier, London
11. Bienia C, Kumar S, Singh JP, Li K (2008) The PARSEC benchmark suite: characterization and architectural implications. In: Proceedings of the 17th international conference on parallel architectures and compilation techniques, ACM, pp 72–81
12. Lachaize R, Lepers B, Quma V (2012) MemProf: a memory profiler for NUMA multicore systems. In: USENIX ATC 12
13. Shen X, Zhong Y, Ding C (2007) Predicting locality phases for dynamic memory optimization. J Parallel Distrib Comput 67(7):783–796
14. Abts D, Enright Jerger ND, Kim J, Gibson D, Lipasti MH (2009) Achieving predictable performance through better memory controller placement in many-core CMPs. ACM SIGARCH Comput Archit News 37(3):451–461
15. Rangan KK, Wei G, Brooks Y (2009) Thread motion: fine-grained power management for multi-core systems. In: Proceedings of the international symposium on computer architecture
16. Lei T, Kumar S (2003) A two-step genetic algorithm for mapping task graphs to a network on chip architecture. In: Proceedings of the Euromicro symposium on digital system design, pp 180C187
17. Rixner S, Dally WJ, Kapasi UJ, Mattson P, Owens JD (2000) Memory access scheduling. ACM SIGARCH Comput Archit News 28(2):128–138
18. Rixner S (2004) Memory controller optimizations for web servers. In: Proceedings of the 37th annual IEEE/ACM international symposium on microarchitecture, pp 355–366
19. Jog A, Bolotin E et al (2004) application-aware memory system for fair and efficient execution of concurrent GPGPU applications [C]. In: Proceedings of workshop on general purpose processing using GPUs
20. (2003) Micron, 1gb, x4, x8, x16, ddr3 sdram datasheet. http://www.micron.com/products/dram/ddr3-sdram. 25 Sep 2013
21. Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S et al (2011) The gem5 simulator. ACM SIGARCH Comput Archit News 39:1C7
22. Wang HS, Zhu X, Peh L-S, Malik S (2002) Orion: a power-performance simulator for interconnection networks. In: Proceedingsof the 35th annual IEEE/ACM international symposium on microarchitecture, MICRO-35, pp 294–305
23. Lozi JP, David F, Thomas G et al (2012) Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In: Proceedings of the Usenix annual technical Conference, pp 65–76
24. Bertozzi S, Acquaviva A, Bertozzi D, Poggiali A (2006) Supporting task migration in multi-processor systems-on-chip: a feasibility study. In: Proceedings of the conference on design, automation and test in Europe, pp 15–20. European Design and Automation Association
25. Katre KM, Ramaprasad H, Sarkar A, Mueller F (2009) Policies for migration of real-time tasks in embedded multi-core systems. In: Real-time systems symposium, pp 17–20
26. Goodarzi B, Sarbazi-Azad H (2011) Task migration in mesh NoCs over virtual point-to-point connections. In: 19th IEEE Euromicro international conference on parallel, distributed and network-based processing (PDP), pp 463–469
27. Briao EW, Barcelos D, Wagner FR (2008) Dynamic task allocation strategies in MPSoC for soft real-time applications. In: Proceedings of the conference on design, automation and test in Europe, ACM, pp 1386–1389
28. Xie B, Chen T, Hu W, Tang X, Wang D (2013) An energy-aware online task mapping algorithm in NoC-based system. J Supercomput 64(3):1021–1037
29. Shim KS, Lis M, Cho MH, Khan O, Devadas S (2011) System-level optimizations for memory access in the execution migration machine (EM2), CAOS

30. Sarkar A, Mueller F, Ramaprasad H, Mohan S (2009) Push-assisted migration of real-time tasks in multi-core processors. ACM Sigplan Not 44(7):80–89
31. Hardy D, Puaut I (2009) Estimation of cache related migration delays for multi-core processors with shared instruction caches. In: 17th international conference on real-time and network systems, pp 45–54
32. Bastoni A, Brandenburg B, Anderson J (2010) Cache-related preemption and migration delays: empirical approximation and impact on schedulability. In: Proceedings of the 6th international workshop on operating systems platforms for embedded real-time apps, pp 33–44
33. Bakhoda A, Kim J, Aamodt TM (2010) Throughput-effective on-chip networks for manycore accelerators. In: Proceedings of the 2010 43rd annual IEEE/ACM international symposium on microarchitecture, pp 421–432, IEEE Computer Society
34. Kim D, Yoo S, Lee S (2010) A network congestion-aware memory controller. In: 2010 IEEE 4th ACM/IEEE international symposium on networks-on-chip (NOCS), pp 257–264
35. Kim D, Kim K, Kim JY, Lee SJ, Yoo HJ (2007) Solutions for real chip implementation issues of NoC and their application to memory-centric NoC. In: IEEE 1st international symposium on networks-on-chip, NOCS 2007, pp 30–39
36. Sharifi A, Kultursay E, Kandemir M, Das CR (2012) Addressing end-to-end memory access latency in NoC-based multicores. In: Proceedings of the 2012 45th annual IEEE/ACM international symposium on microarchitecture, pp 294–304, IEEE Computer Society
37. Chandra R, Devine S, Verghese B, Gupta A, Rosenblum M (1994) Scheduling and page migration for multiprocessor compute servers. ACM SIGPLAN Not 29(11):12–24
38. Corbalan J, Martorell X, Labarta J (2004) Page migration with dynamic space-sharing scheduling policies: the case of the SGIO 2000. Int J Parallel Program 32(4):263–288
39. LaRowe RP Jr, Ellis CS (1991) Page placement policies for NUMA multiprocessors. J Parallel Distrib Comput 11(2):112–129
40. LaRowe RP Jr, Ellis CS, Holliday MA (1992) Evaluation of NUMA memory management through modeling and measurements. IEEE Trans Parallel Distrib Syst 3(6):686–701
41. Blagodurov S, Zhuravlev S, Fedorova A et al (2010) A case for NUMA-aware contention management on multicore systems. In: Proceedings of the 19th international conference on parallel architectures and compilation techniques, ACM, pp 557–558