Defragmentation of Tasks in Many-Core Architecture

ANUJ PATHANIA, Karlsruhe Institute of Technology
VANCHINATHAN VENKATARAMANI, National University of Singapore
MUHAMMAD SHAFIQUE, Vienna University of Technology
TULIKA MITRA, National University of Singapore
JÖRG HENKEL, Karlsruhe Institute of Technology

Many-cores can execute multiple multithreaded tasks in parallel. A task performs most efficiently when it is executed over a spatially connected and compact subset of cores so that performance loss due to communication overhead imposed by the task's threads spread across the allocated cores is minimal. Over a span of time, unallocated cores can get scattered all over the many-core, creating fragments in the task mapping. These fragments can prevent efficient contiguous mapping of incoming new tasks leading to loss of performance. This problem can be alleviated by using a task defragmenter, which consolidates smaller fragments into larger fragments wherein the incoming tasks can be efficiently executed. Optimal defragmentation of a many-core is an NP-hard problem in the general case. Therefore, we simplify the original problem to a problem that can be solved optimally in polynomial time. In this work, we introduce a concept of exponentially separable mapping (ESM), which defines a set of task mapping constraints on a many-core. We prove that an ESM enforcing many-core can be defragmented optimally in polynomial time.

CCS Concepts: • **Computer systems organization → Self-organizing autonomic computing;**

Additional Key Words and Phrases: Many-core, task defragmentation, multiagent systems

ACM Reference Format:
Anuj Pathania, Vanchinathan Venkataramani, Muhammad Shafique, Tulika Mitra, and Jörg Henkel. 2017. Defragmentation of tasks in many-core architectures. ACM Trans. Archit. Code Optim. 14, 1, Article 2 (March 2017), 21 pages.
DOI: http://dx.doi.org/10.1145/3050437

1. INTRODUCTION

Many-cores are processors with dozens of processing cores, which will replace current multicore processors that have only a few cores [Henkel et al. 2012]. A many-core executes several multithreaded tasks in parallel to exploit its full parallel processing potential. In addition, threads of the task can be allocated to one or more cores and can be migrated among the cores at runtime, but the number of cores allocated to the task remains constant—by definition, a *moldable* task [Dutot et al. 2004]. To

This work was supported in part by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89) and in part by the National Research Foundation, Prime Minister's Office, Singapore, under its Industry-IHL Partnership Grant and Huawei International Pte. Ltd. NRF2015-IIP003.
Authors' addresses: A. Pathania and J. Henkel, Chair for Embedded Systems (CES), Karlsruhe Institute of Technology (KIT), Haid-und-Neu-Str. 7, 76131 Karlsruhe, Germany; emails: {anuj.pathania, henkel}@kit.edu; V. Venkataramani and T. Mitra, School of Computing (SoC), National University of Singapore (NUS), 13 Computing Drive, 117417 Singapore; emails: {vvanchi, tulika}@comp.nus.edu.sg; M. Shafique, Embedded Computing Systems Group, Vienna University of Technology (TU Wien), Treitlstrasse 3, 1040 Wien, Austria; email: muhammad.shafique@tuwien.ac.at.

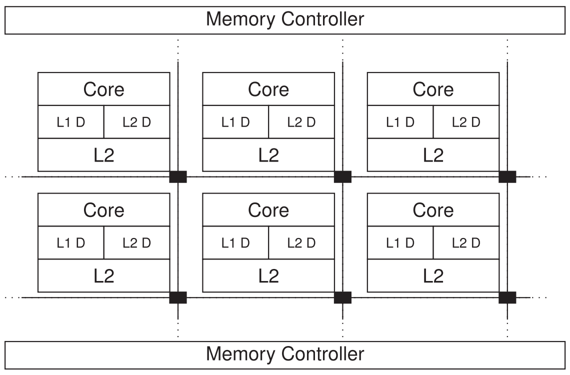


Fig. 1. Abstract architectural diagram of a many-core with cores arranged in a 2D lattice connected together by a mesh NoC.

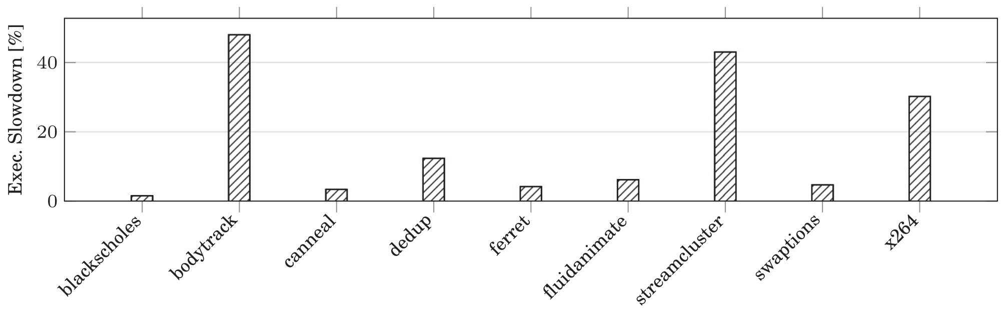


Fig. 2. Observed slowdown in execution time of different multithreaded benchmarks in an isolated execution (executing alone) when its four threads are pinned across four corner cores (cores with only two neighboring cores) of a 64-core many-core in comparison to when they are compactly placed together.

reduce context-switching overhead, many-cores also operate with one thread per core execution model [Pathania et al. 2016]. This also keeps the problem of tasks-to-cores mapping on the many-core discrete.

The cores in a many-core are arranged in a 2D lattice connected by a 2D-mesh network on chip (NoC) as shown in Figure 1. The cores have private L1 instruction and data caches, and a larger L2 cache. The processing core along with its caches form a tile. Each tile is connected through a router to the four adjacent tiles—one in each direction. Tiles are kept coherent using cache directories distributed alongside L2 (the last-level cache (LLC)). Multiple memory controllers attached to the perimeter cores provide access to the off-chip dynamic random access memory (DRAM).

A multithreaded task on the many-core in general performs more efficiently when the set of cores allocated to it are contiguous (spatially connected by an isolated NoC link) and compact (the shape formed by the allocated cores has minimum perimeter). Under such an allocation, the communication cost between the task's threads spread over the allocated cores is minimal. Figure 2 shows the slowdown in an isolated execution time experienced by different multithread benchmarks on a 64-core many-core when their four threads are pinned to the four corner cores of the many-core against when they are placed together.[1] Slowdowns observed in the benchmarks strongly correlate with their originally characterized interthread communications [Bienia et al. 2008].

[1]Refer to Section 5 for complete details of our experimental setup.

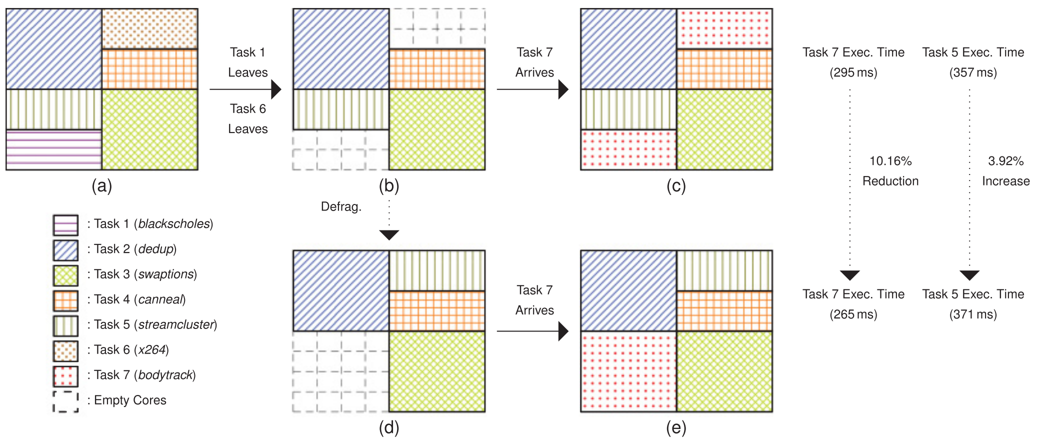


Fig. 3. Example showing 10.16% reduction in an arriving task's (Task 7) execution time if mapped after defragmentation against when mapped without defragmentation. An increase of 3.92% in execution time of the task migrated for defragmentation (Task 5) is also observed as an overhead.

The relative benefits from thread co-location would generally increase with the increase in number of spawned threads of a benchmark because of an increase in interthread synchronizations. Further, under a contiguous allocation, interthread NoC traffic generated by one task remains isolated and does not interfere with another task's NoC traffic. This isolation reduces NoC congestion, enhancing the many-core's multiprogram performance.

Many-cores are now being deployed in embedded servers [Guan and Gu 2010], wherein tasks can arrive at any time and permanently leave the system once they have finished execution—by definition, an *open* system [Feitelson and Rudolph 1998]. Depending on instruction lengths (input sizes) and compositions, the number of cores allocated (threads spawned) and interthread communications execution time of different tasks can differ widely. Therefore, neither the arrival nor the departure time of the tasks are known in an open system. Over a span of time, this results in unallocated cores getting scattered all over the many-core, generating fragments in the task mapping. Formation of these fragments leads to the problem of fragmentation.

Fragmentation makes it difficult to perform efficient compact contiguous mapping of new incoming tasks. Fragmentation can be reduced by using a defragmenter, which consolidates smaller fragments into larger fragments. Defragmentation would lead to a more responsive open system. A centralized defragmenter is sufficient for a multicore. However, for a many-core, given the large optimization search space, it would not scale up. Therefore, a distributed defragmenter that distributes its processing across all cores in the many-core and allows multiple fragments to merge in parallel is required.

Defragmentation motivation. Figure 3 shows a simple illustration of how fragmentation leads to inefficiency on a 64-core processor. Initially, the processor is executing a total of six tasks as shown in Figure 3(a). Tasks 1 (*blackscholes*) and 6 (*x264*) finish and leave the system, changing the many-core state to Figure 3(b). Task 7 (*bodytrack*) then arrives with a requirement of 16 cores. Figure 3(c) shows the state and corresponding execution time of Task 7 if it is mapped without defragmentation. Figure 3(d) shows the state in an alternate timeline if defragmentation is performed first by migrating Task 5 (*streamcluster*) in the middle of its execution before Task 7 is mapped. Figure 3(e) shows the state and corresponding execution time of Task 7 if it is mapped after defragmentation. Experiments show that the execution time of Task 7 is reduced by 30 ms (10.16%) in the state depicted by Figure 3(e) in comparison to the state in Figure 3(c) because of the optimized interthread NoC communications. In contrast, the

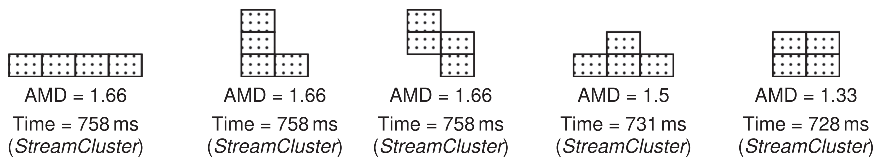


Fig. 4. Different distinct shapes of a tetromino; polyomino of size four.

performance penalty of migration on Task 5 for defragmentation is comparatively less at 14ms (3.92%). Task 5 experiences an elongated execution because thread migrations force its threads to wait until the caches on the newly assigned core are refilled from DRAM. Nevertheless, we observe that the net gain in overall performance of the system is positive.

Computation complexity. The problem of many-core defragmentation is identical to the mathematical problem of mapping polyominoes. *Polyominoes* are complex geometric shapes formed by a combination of simple unit square shapes. Figure 4 shows five distinct shapes that can be formed by a tetromino (polyomino of size four) alongside each shape's average Manhattan distance (AMD). The AMD of a polyomino shape is the average of all rectilinear distances between unit squares forming the shape in 2D space. Rectilinear distance is the shortest distance between two points on an *XY* grid. A unit square is analogous to a thread to task mapping on a many-core. A task's performance, when allocated a contiguous set of cores, is negatively correlated to the AMD of the polyomino shape formed by the allocated cores. This can be observed in the execution time of the *streamcluster* benchmark when it is allocated cores in the different tetromino shape as shown in Figure 4.

The number of distinct shapes that a polyomino of a given size can form grows superexponentially due to its combinatorial nature and is given by *OEIS* sequence *A000105* [Sloane 2003]. Further, mapping a set of polyominoes onto a larger underlying polyomino without overlapping is an NP-hard problem [Demaine and Demaine 2007]. The problem exactly manifests itself while defragmenting many-cores, making the problem of many-core defragmentation NP-hard as well.

Our novel contributions. In this work, we present our idea of exponentially separable mapping (ESM), which defines task mapping constraints on a many-core. We show that this mapping exhibits properties that allow optimal many-core defragmentation to be performed distributively. We also introduce a defragmenter, *McD* (short for *many-core defragmenter*), that shows how ESM properties can be exploited for optimal defragmentation of the many-core. *McD* disburses all of its processing overhead across all unallocated cores in the many-core, allowing it to scale up as the number of cores in the many-cores continue to increase in the future.

2. RELATED WORK

Singh et al. [2013] summarize the prior research that has been conducted on mapping incoming tasks on many-cores. This work focuses on many-cores in open systems wherein both task arrivals and departures are unknown. Incoming tasks can be mapped on a many-core either contiguously or noncontiguously. Contiguous mapping stipulates that cores allocated to a task must always be spatially connected to each other, whereas no such restriction is enforced in noncontiguous mapping.

Contiguous mapping ensures that NoC latency experienced by task threads when communicating is minimal as they are spatially collocated. Further, it also ensures isolation of interthread NoC communication traffic between tasks executing in parallel, reducing NoC congestion. Note that even under contiguous mapping, there will be some external NoC interference due to the OS, cache directories, and DRAM controllers on the periphery. Thus, contiguous mapping improves system performance by optimizing

NoC communications. *SHiC*, a contiguous mapping heuristic introduced in Fattah et al. [2013], uses smart stochastic hill climbing in an attempt to map the incoming tasks contiguously with minimal fragmentation.

Noncontiguous mapping, on the other hand, optimizes system utilization at the expense of communication. It occurs often in an open system with many-cores that there are enough unallocated cores available to satisfy the requirements of incoming tasks, but they are not spatially connected. Noncontiguous mapping does not wait for the required number of spatially connected cores to become available but instead maps them immediately on whichever location the cores are available. This reduces the waiting time for the task but affects its performance throughout execution due to increased communication overhead. In multiprogram workload execution, noncontiguous allocation also degrades the performance of not just the new incoming task that is being mapped but also previously mapped tasks due to NoC congestion. The *CASqA* noncontiguous mapping heuristic introduced in Fattah et al. [2014] initially attempts to map the incoming task contiguously, but if there are not enough contiguous unallocated cores available, it uses nearby cores to map the remaining task noncontiguously.

Both *SHiC* and *CASqA* attempt to reduce fragmentation; however, their efficacy is limited because they to do not perform any thread migrations. The task defragmenter [Ng et al. 2016] is an OS subroutine that combines multiple sets of noncontiguous unallocated cores into a single contiguous set of unallocated cores by performing thread migrations. The incoming task is then mapped efficiently into this newly created contiguous unallocated core set. However, thread migrations involved in the process of defragmentation introduce performance penalties on threads being migrated due to cold cache misses on the newly assigned cores. Therefore, a defragmenter needs to be careful to not perform too many thread migrations or risk being detrimental instead of beneficial to the overall system performance.

Furthermore, a many-core defragmenter should also be scalable so that it continues to operate efficiently as the number of cores in a many-core increases. For scalable solutions, multiagent systems (MAS) [Ebi et al. 2009] are often employed, which are an inherently distributed constructs. Faruque et al. [2008] presented *ADAM*, a heuristic MAS defragmenter for many-cores in which cores are divided into clusters, with an agent assigned to each cluster. Neighboring cluster agents exchange unallocated cores to make space for an incoming task and perform task migrations when necessary. A MAS that operates only locally is bound to get stuck in local minima, which may not be optimal.

Past research tackles the many-core fragmentation problem by proposing suboptimal defragmentation heuristics. However, we solve a constrained version of the many-core fragmentation problem optimally in this work. Our evaluations show that our constraint-optimal approach can result in substantial performance gains on a many-core compared to the state-of-the-art heuristics when constraints are enforced.

In the future, we plan to extend our work to heterogeneous (asymmetric) many-core architectures. A heterogeneous many-core couples together processing cores with different power-performance characteristics alongside application-specific accelerators on a single chip. Scheduling support for heterogeneous multicores is already well developed [Li et al. 2007]. Development of scalable power performance–oriented schedulers for heterogeneous many-cores is a subject of active research [Winter et al. 2010]. However, to best of our knowledge, the problem of defragmentation on heterogeneous many-cores is yet to be studied.

3. EXPONENTIALLY SEPARABLE MAPPING

We begin by introducing our novel idea of ESM that specifies a set of task mapping constraints for many-cores. ESM puts constraints on the number of cores that can

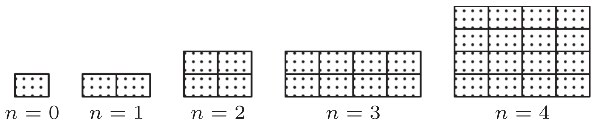
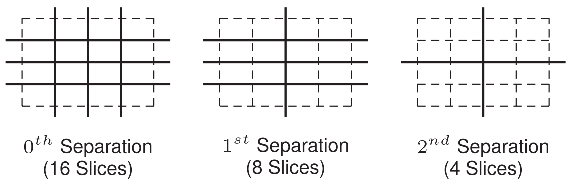


Fig. 5. Different size ES polyominoes.



0^{th} Separation (16 Slices)
1^{st} Separation (8 Slices)
2^{nd} Separation (4 Slices)

Fig. 6. Different size separation of a 16-core processor.

be allocated to a task, the shape of polyominoes that these cores can form, and the physical location of those polyominoes. ESM is akin to a projection of the binary buddy system [Knowlton 1965] in a 2D space [Li and Cheng 1990] but with inherent support for distributed optimization.

Number constraint. ESM requires that a task must always be allocated a number of cores in an exponentiation series with base 2 (or power of two), for instance, 1, 2, 4, 8, ... 2^n, cores. If a task comes with a core requirement that is not a power of two, its requirement is buffered up to the next highest (ceiling) power of two. Speedup in a task execution time is generally monotonically nondecreasing with the number of assigned cores [Pathania et al. 2016]. Thus, by spawning more threads, task would experience an equal or lower response time than it would have if the buffered cores were left idle, preventing system underutilization. However, this constraint also limits the defragmentation benefits for nonscalable tasks, which are not allowed to or are incapable of spawning additional threads on the buffered cores.

Shape constraint. ESM requires that cores allocated to the task form a contiguous minimum perimeter rectangular polyomino. We define polyominoes that follow the shape constraint along with the number constraint as exponentially separable (ES) polyominoes. An ES polyomino of size 2^n can be obtained by symmetrically reflecting a 2^{n-1} ES polyomino along one of its edges. If n is odd, reflection happens along the x-dimensional edge. If n is even, reflection happens along the y-dimensional edge. Figure 5 visualizes the ES polyominoes of different sizes. ESM also requires the underlying many-core on which tasks are being mapped to also be an ES polyomino.

Location constraint. ESM requires that an ES polyomino of size 2^n is physically placed on a many-core at a location so that it does not get separated (dissected) in n^{th} separation of the many-core. A n^{th} separation of the many-core divides the many-core in nonoverlapping ES polyominoes of size 2^n, individually referred to as an n^{th} slice. Figure 6 shows some different size separations for a 16-core processor. Note that mapped ES polyominoes that do not get separated in the n^{th} separation will also not get separated in the $(n+1)^{th}$ separation, as the latter produces slices that are larger and subsume the slices produced by the former.

Figure 7 shows simple examples of mappings that are not ESM. Figure 7(a) has a task with three cores allocated to it, which violates the number constraint. Figure 7(b) has a task with four cores (i.e., 2^2 cores) allocated to it, but the polyomino that these cores form is not an ES polyomino, violating the shape constraint. Figure 7(c) has a task mapped to 2^2 cores, and the polyomino that these cores form is an ES polyomino. However, this polyomino gets split in the 2^{nd} separation, violating the location constraint.

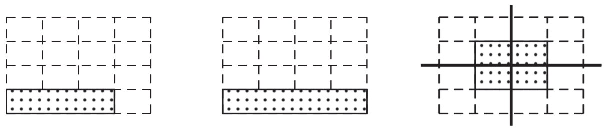


Fig. 7. Example of mappings that are not ESM.

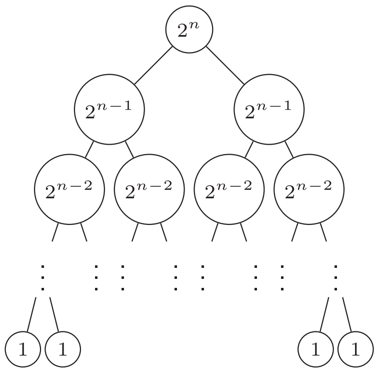


Fig. 8. Split of a power-of-two number 2^N into a smaller power of two numbers.

We now present the theoretical proofs for the properties that ESM possesses. *McD* can exploit these properties to perform optimal and distributed defragmentation of a many-core that enforces ESM.

LEMMA 3.1. *A set of ES polyominoes with total polyomino size $\leq 2^n$ can be split into two sets of ES polyominoes with total polyomino size $\leq 2^{n-1}$.*

PROOF. A singleton set of ES polyomino contains only one element and cannot be split further. For a set of ES polyominoes with cardinality greater than one, the problem is equivalent to proving that a power of two number 2^n when expressed as a sum of smaller positive power of two numbers can be split into two sets each having sum equal to 2^{n-1}. Figure 8 shows the binary tree representing all possible combinations in which smaller exponential numbers can be combined together to form a larger exponential number 2^n. It can be seen that the root (i.e., 2^n) can be reached only when the constituent numbers form two separate sets, each with a total sum equal to 2^{n-1}. The proof can be extended to total sum $\leq 2^n$ by removing the same numbers from both original sets and the two derived separated sets, hence proved. □

LEMMA 3.2. *A set of ES polyominoes of total polyomino size $\leq 2^n$ can always be mapped without overlapping onto a many-core in the shape of an ES polyomino of size 2^n.*

PROOF. If the set of ES polyominoes to be mapped is a singleton, then it can be mapped on the many-core, the latter being an ES polyomino of greater or equal size in comparison to the former. If the set has a cardinality greater than one, then based on Lemma 3.1 it can be separated into two sets of polyominoes of total polyomino size $\leq 2^{n-1}$ each. In parallel, the $(n-1)^{th}$ separation of the many-core will divide it into two equivalent many-cores (two $(n-1)^{th}$ slices), each in the shape of an ES polyomino of size 2^{n-1}. Each part of the original many-core can be assigned to one of the separated sets of ES polyominoes for mapping. The argument can be repeated recursively until all

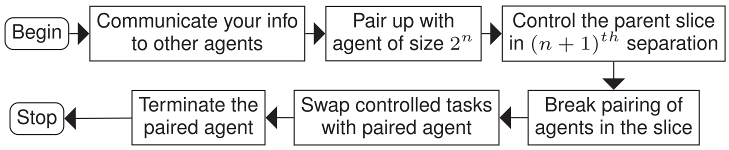

Begin
Communicate your info to other agents
Pair up with agent of size 2^n
Control the parent slice in $(n+1)^{th}$ separation
Break pairing of agents in the slice
Swap controlled tasks with paired agent
Terminate the paired agent
Stop

Fig. 9. Actions performed by an agent of size 2^n to merge with another agent of the same size in a MAS round.

polyominoes are mapped without overlapping on the many-core. The mapping obtained is also ESM since none of the constraints is violated, hence proved. □

Lemma 3.3. *If any of the n^{th} slices obtained from the n^{th} separation of an ESM enforcing the many-core contains more than one ES polyomino mapped to it, then all mapped ES polyominoes in that slice are of size* $\leq 2^{n-1}$.

Proof. Under location constraint imposed by ESM, a mapped ES polyomino of size 2^n cannot get separated in the n^{th} separation of the underlying ESM enforcing many-core. In other words, it would not share the n^{th} slice on which it is mapped with any other ES polyomino. Sharing can happen only in an $(n+1)^{th}$ or larger slice, hence proved. □

4. DEFRAGMENTATION WITH *McD*

System overview. *McD* uses a MAS to perform defragmentation for an N-core many-core that enforces ESM. Whenever a fragment in the shape of an ES polyomino is generated on the many-core, an agent is assigned to it. The sole goal of the agent is to merge with another ES polyomino of the same size. Note that only two ES polyominoes of the same size can be combined to form the next larger size ES polyomino ($\because 2^n + 2^n = 2^{n+1}$). After the merging, the agent of one of the merged fragments terminates itself by transferring the ownership of the entire fragment to the remaining agent. ESM properties allow agents all over the system to merge in parallel, independent of other agents. This allows for distributed defragmentation of the underlying many-core.

System model. Let there be A agents in the system indexed by x, one for each of the fragments that need to be merged in the system. Let S_x be the size of the ES polyomino in the number of cores, associated with an Agent x. Let S be the number of cores that are unallocated, with N being the total number of cores in the underlying many-core.

$$S = \sum_{x=1}^{A} S_x \leq N$$

Trivially, $S = N$ when the system is empty. We assume a first-in, first-out (FIFO) waiting queue in our open system. Let $N' \leq N$ be the requirement of the task waiting in front of the queue. Under ESM, N' must be a number that is power of two. Based on Lemma 3.2, we know that if $N' \leq S$, then the waiting task can always be mapped on the ESM enforcing many-core provided it is optimally defragmented. In a nonpreemptive system like ours, if $N' > S$, then incoming tasks can never be mapped and need to wait in the queue for one or more executing tasks to leave the system.

Objective. The objective of *McD* is to keep $S \leq N'$ at all times. In other words, the number of free cores in the system should always be less than the core requirement for the task in front of the FIFO task queue.

Merging fragments. Merging fragments or defragmentation happens in a series of MAS rounds. Figure 9 shows the actions performed by an individual agent in a round using a flowchart. At the start of every round, an agent communicates with other agents and pairs up with one of the same size. For example, let Agents a and b pair up with each other such that $S_a = S_b$. Now they want to merge to form a fragment of

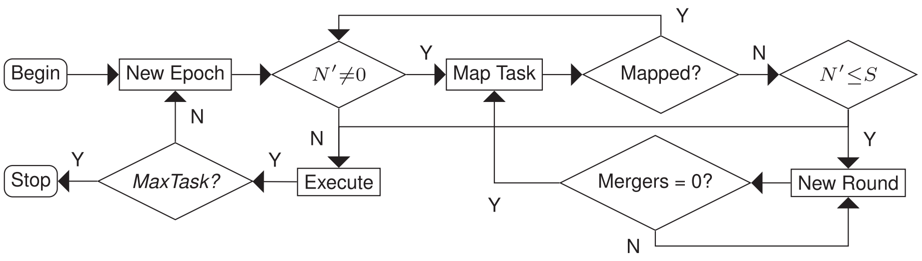


Fig. 10. Execution flow for *McD*.

size $S_a + S_b = 2S_a = 2S_b$. Let ρ_a and ρ_b be the two slices in an $\ln(S_a) + 1$ separation containing Agents a and b, respectively. Agent a (or b) holds half of the ρ_a (or ρ_b) slice as a fragment. Based on Lemma 3.3, all other tasks mapped in the ρ_a (or ρ_b) slice are self-contained in the remaining filled half of the slice. Thus, Agent a (or b) can swap the filled half of the ρ_a (or ρ_b) slice with the fragment in the ρ_b (or ρ_a) slice to merge together with Agent b (or a) without violating ESM. Another pair of agents, c and d, can do the swap in parallel as long as neither Agent c nor d is inside the ρ_a (or ρ_b) slice that is being swapped by Agent a (or b). Otherwise, the merging of Agents c and d is skipped in this round. Therefore, the pairing of smaller fragments is delayed in favor of the pairing of larger fragments when overlapping. After all parallel swaps finish, the next round begins. Rounds end when no more pairings occur.

THEOREM 4.1. McD *performs optimal defragmentation of a many-core that enforces ESM constraints.*

PROOF. Let us assume that *McD* is suboptimal. After *McD* has finished defragmentation, $\nexists x$ such that $S_x = N'$ when $S \geq N'$, where N' is the cores requirement of the task in front of FIFO queue. Based on Lemma 3.3, there can exist at most one agent of size $N'/2$; otherwise, *McD* would have merged the two of them to create an agent of size N'. Recursively, there can exist only at most one agent of size $N'/2$, $N'/4$ and so on. Therefore, $S \leq N'/2 + N'/4 + \cdots + 1 \leq N' - 1$. Based on this argument, $S < N'$, which is a contradiction to our original statement $S \geq N'$, hence proved. □

It is important to note that *McD* only claims optimality with respect to system performance in the final stable state (defragmented) obtained. The problem of going from one state (fragmented) to another state (defragmented) in a minimum number of steps (an optimal number of task migrations) on a many-core is similar to optimally solving a sliding-puzzle problem [Demaine and Hoffmann 2001], which is also an NP-complete problem. The problem of defragmentation in a minimum number of task movements is beyond the scope of this article.

Execution flow. Figure 10 shows a flowchart depicting the execution flow of *McD*. *Scheduling epoch* defines the granularity at which scheduling in a system is performed; in our experiments, it is set at 10ms, which is the same as in default Linux schedulers [Pallipadi and Starikovskiy 2006]. At every scheduling epoch, *McD* can be invoked to perform task mapping of incoming tasks, and defragmentation if required. Initially, the system with *McD* is empty and there is only one agent controlling the unified fragment of N cores. *McD* then checks whether the FIFO queue is not empty ($N' \neq 0$). If it is empty, *McD* waits for the first task to arrive. If the queue is not empty, then tasks are picked from the FIFO queue and are mapped on the many-core under ESM constraints on a first-come, first-served (FCFS) basis until no more tasks can be mapped. FCFS also ensures that there is no task starvation in the system.

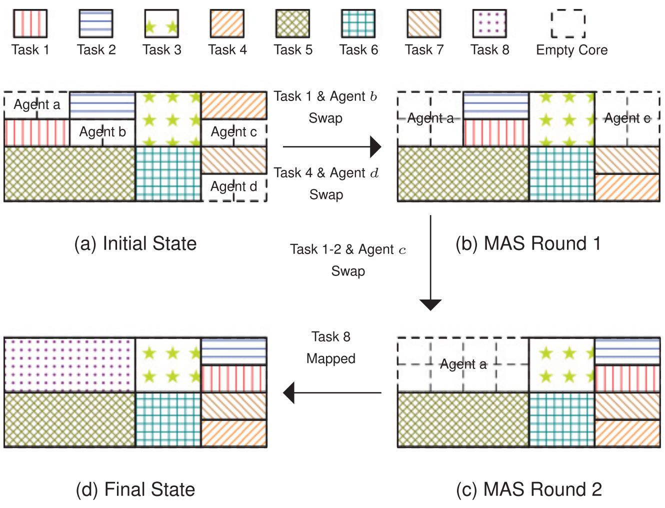


Fig. 11. Illustrative example of distributed defragmentation performed under *McD* on a 32-core processor.

Based on Lemma 3.2, if $N' \leq S$, then there is a scope for more tasks to be mapped on the many-core. However, if there is no contiguous fragment of size $\geq N'$, fragmentation prevents further new mappings. *McD* then invokes the MAS rounds to defragment the system. Once the MAS rounds are completed, the mapping of incoming tasks is invoked again. The process is iteratively repeated until the queue empties ($N' = 0$) or a task in front of the queue is too big to be mapped on the many-core ($N' \geq S$). Tasks mapped on the many-core are now executed for a time defined by the scheduling epoch. Tasks that are completed leave the system at the end of a scheduling epoch, creating new fragments (and spawning agents). If the number of completed tasks reaches the user-defined *MaxTasks*, the system simulation stops and the performance attained is reported. Otherwise, a new scheduling epoch commences in the system.

Illustrative example. Figure 11 illustrates a defragmentation performed under *McD* on a 32-core processor executing seven tasks. Incoming Task 8 requires 8 cores. Figure 11(a) shows that 8 unallocated cores are available on the processor in the initial state but that these unallocated cores are fragmented all over the processor in the form of four fragments of size two each. To map Task 8 contiguously on the processor, *McD* is invoked for defragmentation. Each fragment is assigned a unique private agent (a to d). MAS rounds now begin. In *Round 1*, Agent a pairs up with Agent b, and simultaneously Agents c and d pair up. Agent a takes control of Task 1 as they share the slice in the second separation and swaps it with Agent b. In parallel, Agent c takes control of Task 4 and swaps it with Agent d. Figure 11(b) shows the resultant many-core state after the swaps are performed. Agents b and d terminate themselves in favor of Agents a and c, respectively. In *Round 2*, the remaining Agents a and c pair up. Agent a takes control of Tasks 1 and 2, and swaps it with Agent c. Agent c then terminates itself in favor of Agent a. Figure 11(c) shows the resultant state. MAS rounds now stop because there are no more fragments to merge. *McD* now maps Task 8 over Agent a as shown in Figure 11(d), and all tasks resume execution.

Complexity. On an N-core many-core, there can be at most N fragments (or agents) to merge. This merging will take $O(\ln N)$ rounds under *McD*. In every round, every agent performs at worst $O(N)$ calculations. Thus, in total, there is $O(N^2 \cdot \ln N)$ processing overhead, with per-core processing overhead being $O(N \cdot \ln N)$. Every round involves broadcasting at most $O(N)$ messages; hence, in a worst-case, total communication overhead is $O(N \cdot \ln N)$. Since *McD* does not require any data structure to be maintained or created, the space overhead is $O(1)$.

5. EXPERIMENTAL EVALUATIONS

We evaluate *McD* using the Sniper interval simulator [Carlson et al. 2011]. In comparison to a timewise infeasible cycle-accurate simulator like *gem5* [Binkert et al. 2011], Sniper allows for multiprogram simulations in a reasonable time. In comparison to a trace-based simulator like Noxim [Catania et al. 2015], it allows for more accurate and realistic multiprogram simulations.

The conceptual block diagram of the many-core architecture used in this work is shown in Figure 1. We use OS-level page allocation instead of traditional address interleaved cache directories [Cho and Jin 2006] for managing distributed L2 caches. This architecture consists of 8×8 tiles connected using a 2D mesh interconnection network (NoC) implementing *XY* routing with latency of four cycles per hop and links with bandwidth of 256 bits/cycle. We used 64 cores for evaluation, as multiprogram executions on many-cores with larger numbers of cores were difficult to simulate due to simulation time and memory constraints.

Each tile consists of an out-of-order Intel Gainestown core running at 1GHz implementing x86-64 Instruction Set Architecture (ISA) with private four-way associative 16KB L1 instruction and data caches. An eight-way L2 cache (4MB) is distributed across the chip with a 64KB slice of private L2 residing near each tile. The caches are kept coherent using the directory-based Modified Shared Invalid (MSI) protocol. External memory requests are provided by 1GB off-chip DRAM accessed using four memory controllers along the four edges of the many-core. Hit latencies of L1 caches, L2 caches, and main memory are set at 3, 8, and 80 cycles, respectively. *McD* is equally applicable if the LLC was shared by all cores instead of being private. Conceptually, it makes no difference for *McD* if cache coherency was replaced by message passing. However, changes in topology, such as having more than one processing core per tile, can potentially make the problem of many-core fragmentation NP-hard again even under ESM constraints. Therefore, minor changes to the underlying system architecture may or may not break *McD*'s optimality and needs to be studied individually on a case by case basis.

In software, we use the PARSEC [Bienia et al. 2008] multithreaded benchmark suite with *sim-small* input. We chose *sim-small* input because the next smaller input, *sim-dev*, was not representative enough of real-world tasks, whereas the next larger input, *sim-medium*, took too long to simulate. We also found that the instruction count of *sim-small* inputs is sufficiently large enough to stress the caches on our simulated system in a meaningful way. Among 13 available benchmarks in the PARSEC suite, two benchmarks, *freqmine* and *vips*, were discarded due to unresolved PIN errors. *PIN* [Reddi et al. 2004] is a closed-source binary instrumentation tool from Intel that is used inside the Sniper simulator and prevents debugging of its error. Furthermore, the benchmarks *facesim* and *raytrace* were discarded due to lack of *sim-small* input. Table I summarizes the system configuration of the simulated many-core architecture and the benchmarks used in our evaluation.

Implementation details. *McD* is integrated with the default Pinned scheduler of Sniper. We implement *McD* as a multithreaded distributed system application written in C with the master-slave thread model. The main *McD* thread is spawned when the system starts and is then put to sleep. At the end of each scheduling epoch (10ms of simulated system time), a time-trigger interrupt wakes the main *McD* thread in any of the random free core. The main thread then checks the status of the many-core, such as the number of free cores and FIFO task queue, and then determines if defragmentation is required. If defragmentation is indeed needed, it then determines the

Table I. System Configuration of Simulated Many-core Architecture

Cores	64 x86-64 out-of-order cores
L1 Cache	Split I & D, 16 KB, 4-way, 64 B block, 3-cycle access latency
L2 Cache	Private 64 KB, 8-way, 64 B block, 8-cycle access latency
Directory	MSI coherence, distributed directory entries across tiles
Network	2D mesh, 4 cycles per hop latency, 256 bits/cycle bandwidth, *XY* routing
Memory	1GB, 80-cycle access latency, 4 memory controllers on 4 edges
Benchmarks	*blackscholes*, *bodytrack*, *canneal*, *dedup*, *ferret*, *fluidanimate*, *streamcluster*, *swaptions*, and *x264*
Task Model	Multiprogram, multithreaded
System Model	1 thread per core using private LLC, FIFO task queue

required number of *McD* agents necessary to perform defragmentation and free cores that each of those agents is responsible for merging. For each of the required agents, a slave thread is spawned on a free core that falls within its responsibility. The master thread itself also acts as an agent responsible for a few of the free cores. The threads synchronize with each other using memory, and all coherency traffic is sent via NoC. The thread migrations for PARSEC threads is performed by *McD* threads using custom extensions to Sniper's magic instructions. Suspension, resumption, and placement of *McD* threads is done by the Sniper scheduler using default native instructions. When all *McD* threads except the master have terminated, defragmentation is complete. The master thread then maps new tasks from the FIFO queue onto the many-core and then goes back to sleep. It is important to note that all *McD* threads mostly operate on free cores, and minimal context switching with PARSEC threads is required. *McD* threads operate by manipulating thread-to-core affinities of PARSEC threads, and the bulk of the actual thread migration and context switching heavy lifting is left to the default Sniper scheduler to be performed internally. This implementation is designed to work specifically with the Sniper simulator and may not be the best design for a real-world many-core.

For each reported result, multiple multiprogram workloads were evaluated and then averaged. Each workload comprised 20 tasks, with each task projecting a random core requirement. Simulation-time constraints prevented evaluation of a larger size or number of workloads. The arrival time of tasks in a workload follows Poisson (random) distribution. Due to the peculiar nature of the defragmentation problem, the system will not require defragmentation if core requirements of tasks are too small in comparison to size of the many-core, as most tasks then would fit inside easily. Similarly, if core requirements of tasks are too large, we will also not require defragmentation, as few tasks will occupy the entire many-core. Thus, we also set a lower limit and an upper limit of core allocation to a task to 4 and 16 cores, respectively. We empirically found the limits to be reasonable given the number of cores in our simulated many-core (64).

Optimizations. We implement a couple of optimizations over the default Sniper code base to improve the NoC traffic isolation in the shared-cache many-core architecture, as shown in Figure 12. By default, Sniper uses address interleaving for mapping data on distributed L2 caches and separating accesses from memory controllers to DRAM. Hence, the cache directory responsible for cache blocks of a given application can be present in any of the L2 banks. Additionally, they can access all DRAM controllers for fetching data from main memory. This can result in executing tasks accessing all L2 cache banks and DRAM controllers providing no isolation or clear benefits from defragmentation. This can be seen in Figure 12(a), in which two tasks, A and B, are placed on a 64-core many-core at its top left and bottom right corners, respectively.

Inspired by Cho and Jin [2006], we ensure that addresses accessed by a multi-threaded application are always confined within L2 cache banks near cores on which

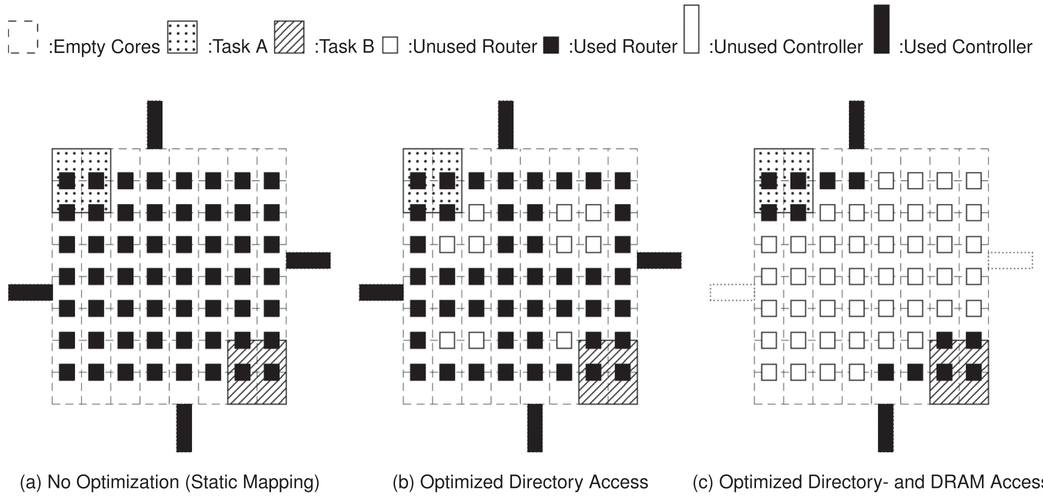

:Empty Cores
:Task A
:Task B
:Unused Router
:Used Router
:Unused Controller
:Used Controller
(a) No Optimization (Static Mapping)
(b) Optimized Directory Access
(c) Optimized Directory- and DRAM Access

Fig. 12. Optimizations for achieving NoC traffic isolation on a shared-cache many-core.

the threads are executing. Cho and Jin [2006] utilize interleaving at page granularity for allocating pages to distributed L2 caches. Whenever a new page is requested by an application/thread, the OS chooses free pages from physical memory that can be mapped closer to the requesting core's cache bank. Due to lack of an OS in the Sniper simulator, we were forced to achieve the equivalent effect by modifying the address lookup function in the Sniper code base. The advantages of using this mechanism can be seen in Figure 12(b), but we still have traffic going to all DRAM controllers.

For preventing applications from accessing all DRAM controllers, we make the entire address space accessible to all controllers. Parallel accesses to the same address are synchronized using an off-chip priority queue as mentioned in Kim et al. [2012]. We also ensure that NoC traffic on an L2 cache miss always goes to the closest DRAM controller among all available controllers. The benefits can be seen in Figure 12(c), where both mapped tasks achieve almost complete NoC isolation.

In this work, all simulations were performed in multiprogram mode with full modeling of cache contention, NoC contention, memory contention, and performance penalties from task migrations. A single simulation of a 20-task workload took approximately 10 hours on average to complete on an Intel Core i7 processor.

Performance metric. *Average response time* is the standard metric to gauge performance of an open system, and *lowest response time* is desired from the system [Feitelson and Rudolph 1998]. The response time of a task is the difference between its arrival and departure time. It is composed of two components: waiting and servicing time. *Waiting time* is the time spent by a task in the FIFO queue before it gets mapped. *Servicing time* is the time spent by a task to finish execution once it is mapped. Average response time for a workload is the mean of the response times of all of its tasks.

Basic comparative baselines. To demonstrate the effectiveness of *McD*, we chose to compare against two simple but predictable approaches, *Contig* and *NonContig*, representative for contiguous and noncontiguous mapping without defragmentation (task migration), respectively.

Contig exhaustively searches each core of the many-core for a contiguous compact mapping for an incoming task. It thereafter maps the task to the first set of contiguous cores that can satisfy the task's requirement. Since *Contig* always maps the incoming task in the best possible shape, it always results in the optimal servicing time for the executing task. However, since it waits passively for contiguous cores in an ideal shape to become available, it results in a suboptimal waiting time.

NonContig, on the contrary, assigns the core to any of the unallocated core on the many-core with no regard to contiguity. *NonContig* results in an optimal waiting time for a task, as the task is mapped as soon as the required number of unallocated cores is available on the many-core irrespective of spatiality. However, since the allocation is neither compact nor contiguous, it results in a suboptimal servicing time.

Contig and *NonContig* represent two extreme points in the performance spectrum, each guaranteeing optimality for one aspect of the performance myopically while disregarding the other. In multiprogram execution, waiting and servicing time are not completely independent. For example, a task that holds cores longer than necessary under a suboptimal servicing time ends up adding additional delay to the waiting time of all tasks in the wait queue. The *McD* defragmenter introduced in this work can optimize both aspects of performance (waiting and servicing time) together, provided that ESM constraints are enforced. Comparison of *McD* versus *Contig* and *NonContig* results in greater insight than comparison with the previously proposed heuristic approaches for preventing fragmentation. Given the lack of guarantees on either waiting time or servicing time in heuristics, it is difficult to say what part of the performance spectrum they represent. Heuristics are also very sensitive to the input workloads and can perform unexpectedly good or bad.

Heuristic baselines. For complete coverage, we also compare against heuristic approaches designed to address fragmentation. We believe that *SHiC* [Fattah et al. 2013], *CASqA* [Fattah et al. 2014], and *DeFrag* [Ng et al. 2016] are the most recent state-of-the-art heuristics for fragmentation-aware contiguous mapping, fragmentation-aware noncontiguous mapping, and defragmentation, respectively. All compared heuristics were designed originally to operate with profiled tasks whose task graphs (thread-spawning and interthread communication patterns) were assumed to be deterministic and predictable. Such task graphs are not readily available for real-world representative PARSEC benchmarks. Further, in multiprogram execution, it is also not trivial to predict when a particular thread will be spawned. We neither assume nor have the complete profile information of all tasks to implement the heuristics strictly in their original form. Hence, we were required to slightly adapt the heuristics to still make them work on our infrastructure. Originally, all compared heuristics were evaluated on Noxim, a trace-based simulator. We reimplement them on the more real-world representative Sniper interval simulator used in this work.

SHiC [Fattah et al. 2013] uses a stochastic approach in an attempt to map the incoming tasks contiguously with minimal fragmentation. It employs smart hill climbing for finding a suitable candidate unallocated core to perform contiguous mapping around in consideration with already mapped tasks to improve the overall contiguity. In each iteration of hill climbing, a random unallocated core is selected and a "square factor" around that core is calculated. The *square factor* is the number of unallocated cores in the largest square of unallocated cores that can be made around the selected core plus the number of unallocated cores in the next largest square of unallocated cores that cannot be completed. If the square factor is equal to the incoming task's requirement, then the selected unallocated core is chosen as a possible candidate for mapping. Otherwise, a random walk from the selected core is performed. Random walk is performed toward one of the eight adjacent cores of the selected core that has a lower or higher square factor depending on whether the selected core has a higher or lower square factor than the number of cores required by the incoming task, respectively. Random walk is terminated after $N/2$ steps if it fails to find a candidate. Hill climbing itself is terminated after $2 + \sqrt{APPS}$ iterations, where *APPS* is the number of task application (tasks) currently mapped on the many-core. If hill climbing finds multiple candidates for mapping the incoming task, the one closest to the edge of the many-core is selected. Once the candidate core is selected, a compact contiguous mapping is performed around it. The authors of *SHiC* have shown it to be superior to several previously proposed similar fragmentation-aware contiguous mapping heuristics.

CASqA [Fattah et al. 2014] is an extension of *SHiC* for noncontiguous mapping. It allows the user to adjust contiguousness of a mapping using a threshold α, but in this work we set the threshold to a value that allows for unbounded noncontiguousness ($\alpha = 1.0$). *CASqA* uses the same stochastic hill-climbing algorithm as *SHiC* to find the first candidate unallocated core to perform mapping around. It then starts exploring squares with an incrementally increasing radius for unallocated cores and stops only when enough cores are found to meet the requirements of a task.

DeFrag [Ng et al. 2016] decides whether to perform defragmentation after a task leaves based on a fragmentation metric. The *fragmentation metric* is defined as the difference between the expected number of unallocated cores required by an incoming task and the size of the largest contiguous set of unallocated cores available. To avoid excessive task migration overhead involved in defragmentation, *DeFrag* invokes defragmentation only when the fragmentation metric is positive. If and when the defragmentation is invoked, all unallocated cores calculate the distance from all other unallocated cores. The unallocated core with a minimum total distance is selected as the center core. A convex contiguous region of size equal to the total number of unallocated cores is found around the selected central core. The unallocated cores then travel hop by hop to a closest position in the convex contiguous region, performing thread migrations on busy cores in their paths. The authors' originally proposed algorithm to determine the "Minimal-Cost Migration Path" requires complete task profiles, which we neither assume nor possess. We instead choose the shortest path algorithm to find the migration path of an unallocated core to the convex contiguous region. Finally, the incoming task is mapped compactly in the convex contiguous region provided that it is large enough.

Performance under the power of two constraint. We begin by evaluating performance when tasks in workloads are only allowed to project the requirement of 2^n cores as stipulated under ESM. We execute workloads with different arrival rates under various approaches on an open 64-core many-core. For an open system, an increase in arrival rate translates to increased system load as more tasks start arriving in the system together for execution.

Figure 13(a) through (c) record the observed average waiting, servicing, and response time under different arrival rates, respectively. Recorded results also implicitly incorporate processing and communication overhead of the deployed defragmenter. It is important to note that *McD* always outperforms the comparative baselines in both waiting and servicing time. Hence, *McD* always results in a superior response time. Initially at lower arrival rates when the system is underloaded, executing tasks are sparsely distributed over the many-cores and most of the incoming tasks can be mapped efficiently without waiting by all approaches. The importance of defragmentation increases as the system load increases. We observe in Figure 13 that *McD* provides greater performance gains when the system is substantially loaded. The performance gains from *McD* saturate when the system is overloaded at very high arrival rates. The improved performance under *McD* comes from its ability to create a compact contiguous space for every incoming task as soon as possible, resulting in the minimum possible waiting time. It also ensures that all tasks are always executed efficiently with the least possible communication overheads, resulting in a minimal servicing time. Figure 13 shows that *McD* can result in up to 8.81% and 19.53% additional performance compared to *Contig* and *NonContig*, respectively.

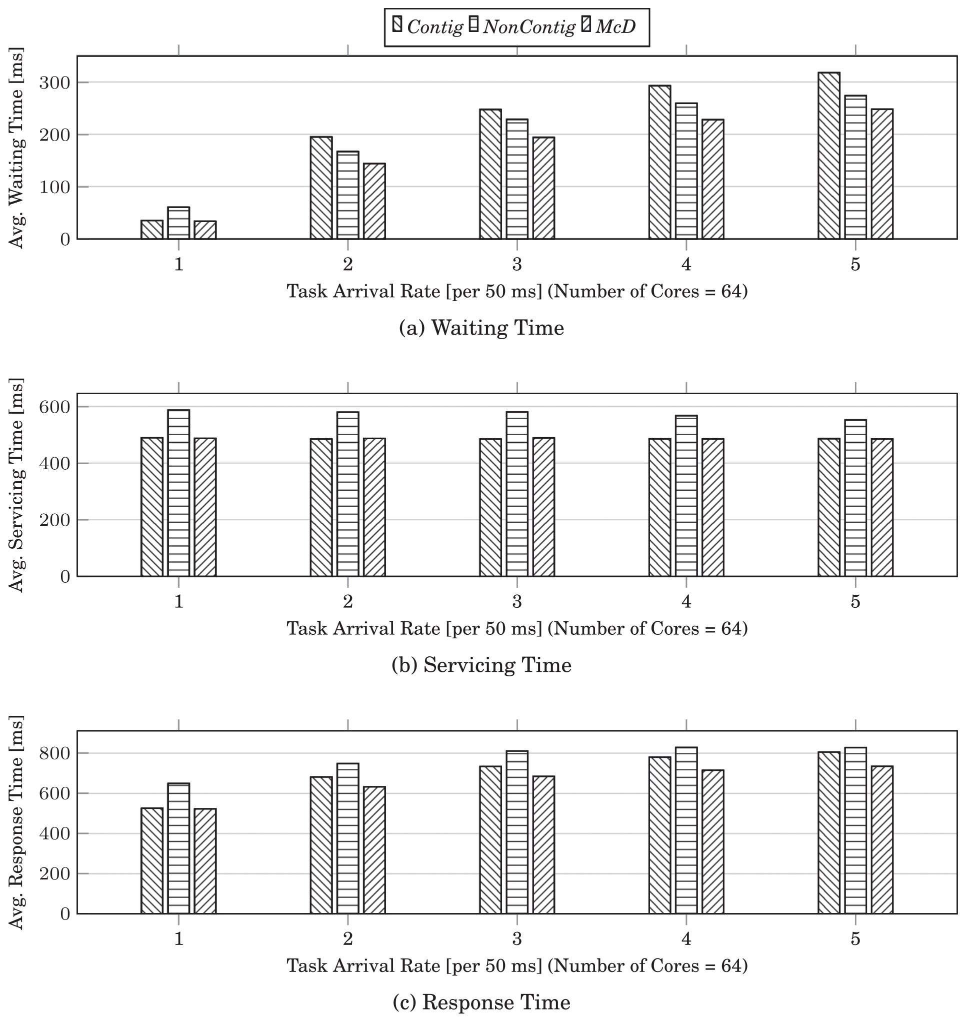


Fig. 13. Performance comparison between *McD* with *Contig* and *NonContig* under different arrival rates when workloads are enforcing a power of two ESM constraint.

In Figure 14(a), we explain the observed performance gain under *McD* using insights from a relevant performance counter for a randomly selected workload. The observations under all approaches are normalized against observations made under *McD* so that all of them can be shown concisely on the same graph. We observed that the NoC packets transmitted remain nearly the same under all approaches. Still, we observed that the packet delay due to NoC latency (*Queue Delay*) and delay due to NoC congestion (*Contention Delay*) is several times higher for *NonContig*. This significantly degrades the performance under *NonContig*. The number of instructions processed by *NonContig* is significantly higher because of the additional processing done by the tasks actively waiting longer for thread synchronizations to complete. This also results in higher CPU utilization under *NonContig*, but this increased utilization in practice is actually detrimental instead of beneficial for overall system performance. Reduced performance under *Contig* is mainly due to lower CPU utilization, as it keeps the tasks waiting longer in the queue. This results in lower congestion in NoC links, but

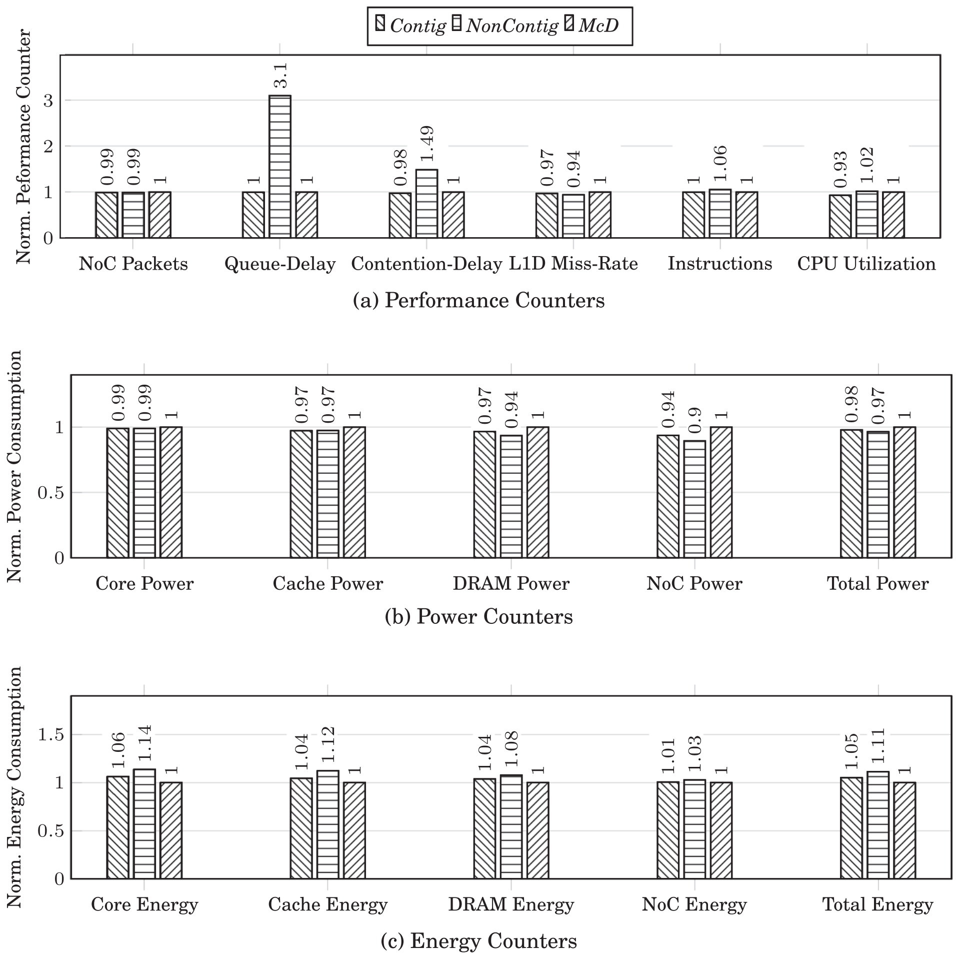


Fig. 14. Observed values (normalized against *McD*) for relevant counters for a given workload when executed under different approaches.

the reduced congestion still cannot compensate for the performance drop due to low CPU utilization. The L1 data cache miss rate is higher for *McD* than other approaches because of the involved defragmentation-related thread migrations.

Figure 14(b) and (c) show the selected workloads' normalized power and energy consumption, respectively. In comparison to baselines, *McD* pushes to execute more load in parallel, and as a result we see that all system components have higher power consumption. Still, executing more load in parallel allows it to also finish execution faster, resulting in lower energy consumption for all system components. Overall, *McD* results in a 2.06% increase in total power consumption while reducing the total energy consumption by 4.85%.

Figure 15 shows a performance comparison between *McD* and adapted versions of state-of-the-art heuristics designed to tackle fragmentation. *SHiC-like*, *CASqA-like*, and *DeFrag-like* symbolically represent the reimplemented versions of *SHiC*, *CASqA*, and *DeFrag*, respectively. We observe that *McD* can result in up to 12.54%, 8.35%, and 24.09% improved performance in comparison to *SHiC-like*, *CASqA-like*, and *DeFrag-like*, respectively. *SHiC-like* and *CASqA-like* perform worse for the same

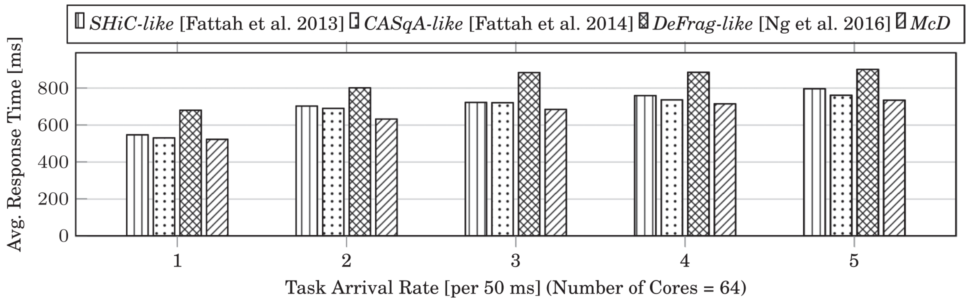


Fig. 15. Performance comparison between *McD* with *SHiC* [Fattah et al. 2013], *CASqA* [Fattah et al. 2014], and *DeFrag* [Ng et al. 2016] under different arrival rates when workloads are enforcing a power of two ESM constraint.

reasons as *Contig* and *NonContig*—a combination of suboptimal servicing and waiting time. Further, given their stochastic nature, their performance does not just vary with input but also in every execution based on the seed used for randomization. We also found that a hop by hop thread migration approach used by *DeFrag-like* is expensive, as it leads to heavy displacement of existing tasks and also does not preserve their contiguity, resulting in inferior performance. On the other hand, under ESM, *McD* always guarantees optimality irrespective of input and execution is always deterministic.

Performance under no constraints with scalable tasks. *McD* is ideally designed to perform optimally under the ESM constraint, which stipulates a core requirement of all tasks in powers of two. When this requirement is not enforced, for continued operations *McD* needs to buffer the number of cores allocated to a task to the next higher power of two. For example, if a task comes with a fixed requirement of seven cores, it must be allocated eight cores. This can lead to the problem of system underutilization due to intratask fragmentation or "internal" fragmentation if the buffered cores are not put to use by the tasks. *McD* is designed to minimize intertask fragmentation, as otherwise "external" fragmentation is not able to compensate for this internal defragmentation. The comparative baselines *Contig* and *NonContig* used in this work do not require any such buffering since they stipulate no such constraints.

To prevent system underutilization, *McD* allows tasks to spawn threads even on the buffered cores. This is permissible because our tasks made from PARSEC benchmarks are moldable. The number of threads spawn by a PARSEC benchmark is fixed once its main thread starts; however, before the execution begins, the maximum number of threads that it can invoke can be passed as a parameter to its main thread. Most PARSEC benchmarks support many different values of the maximum thread count parameters, which *McD* can exploit to prevent system underutilization.

Since the execution time of tasks is generally monotonically nondecreasing with the number of cores assigned (thread spawned) [Pathania et al. 2016], all tasks will execute faster, resulting in a more responsive system. Our evaluations in Figure 16 show that *McD* performs better than comparative baselines with scalable tasks even when the power of two core requirement is not enforced. Note that the randomized workload used in Figures 13 and 16 are different, and hence numbers are not directly comparable.

Performance under no constraints with nonscalable tasks. *McD* can be severely handicapped if the tasks are not able (or not allowed) to spawn additional threads on the buffered cores. Empty buffered cores can cause substantial system underutilization, and even our basic comparative baselines are capable of outperforming *McD*. Figure 17 shows the limitations of *McD* wherein even our basic comparative baseline like *NonContig* now outperforms *McD* by 4.35% because tasks (PARSEC benchmarks) are not

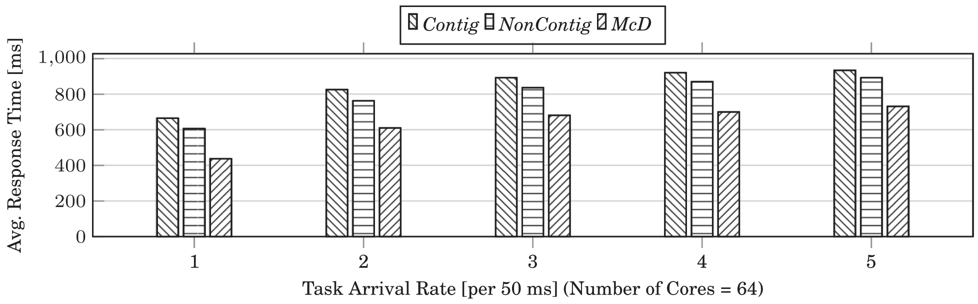


Fig. 16. Performance comparison between *McD* with *Contig* and *NonContig* under different arrival rates when workloads are not limited by any constraint and tasks are capable of spawning additional threads.

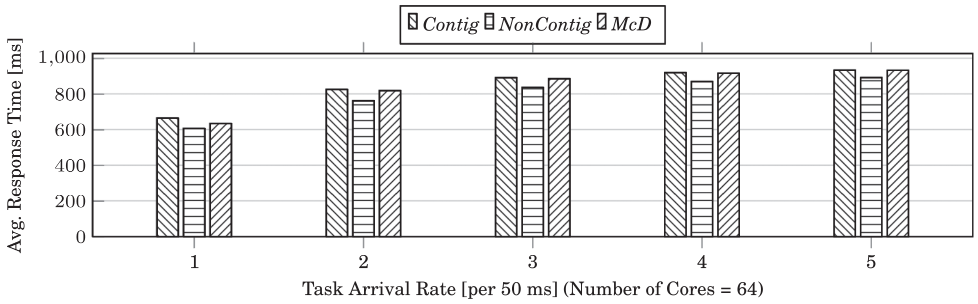


Fig. 17. Performance comparison between *McD* with *Contig* and *NonContig* under different arrival rates when workloads are not limited by any constraint and tasks are not allowed to spawn additional threads.

allowed to spawn additional threads. The performance gap will widen even further against the heuristic baselines. This brings forth the drawbacks of a constraint optimal like *McD* in general, where the price of maintaining optimality may be too high. Therefore, *McD* should not be used with tasks that are incapable of scaling up their thread count.

Scalability. The simulation time for a many-core is directly proportional to the number of parallel instruction executions simulated. To the best of our knowledge, no simulator, other than trace-based simulators, can simulate a 1,000-core many-core under heavy load in a reasonable amount of time. This makes it difficult to obtain overhead numbers directly from a multiprogram simulation of PARSEC workloads on a realistic simulator like Sniper to demonstrate scalability.

On the other hand, stand-alone execution of the *McD* algorithm for a large-size input is still feasible timewise. Hence, we execute *McD* with varisized worst-case representative inputs as an isolated distributed application over the simulated many-cores with 64 cores or more in Sniper and report *McD*'s problem-solving time. This execution time incorporates both the communication overhead of *McD* agent threads communicating through memory via NoC as well as their processing overheads. We believe that this is the closest we can get to obtaining real-world overhead of our approach on large-size many-cores. This overhead is directly comparable to the processing time of PARSEC workloads themselves.

Figure 18 shows the time it takes for *McD* to perform worse-case many-core defragmentation for 64-core to 512-core many-cores. It takes *McD* 1.115ms to solve the worst-case defragmentation problem on a 512-core many-core. For a scheduling epoch of 10ms used in this work, this results into a worst-case overhead of 11.15% on a

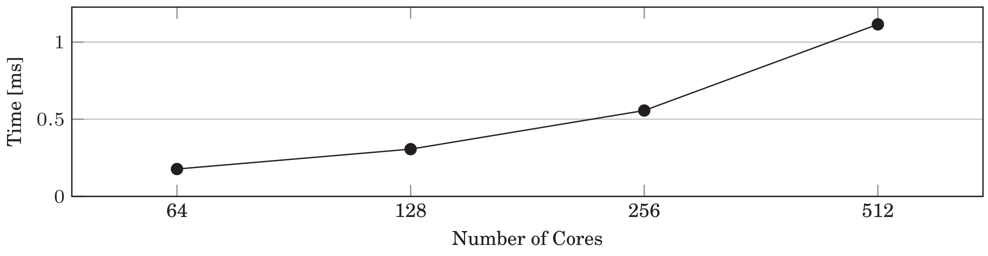


Fig. 18. Worst-case problem-solving time taken by *McD* on varisized many-cores.

512-core many-core. The worst-case defragmentation overhead on a 64-core many-core stands at an acceptable 1.77%.

6. CONCLUSION

In this article, we address the problem of many-core defragmentation, which is known to be NP-hard. To make the problem tractable, we simplify it to a problem that can be solved optimally in polynomial time by introducing the concept of ESM for many-cores. ESM puts constraints on tasks-to-cores mapping on a many-core, allowing for its optimal distributed defragmentation in polynomial time. We also introduced a defragmenter called *McD*, which demonstrates how ESM can be exploited. Our experiments show that defragmentation under *McD* increases performance and reduces energy consumption of a many-core. Since *McD* is proven optimal, it provides the maximum possible performance under ESM constraints, which cannot be surpassed by any other algorithm.

REFERENCES

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 2008 Conference on Parallel Architectures and Compilation Techniques (PACT'08)*.

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, et al. 2011. The gem5 simulator. *SIGARCH Computer Architecture News* 39, 2, 1–7.

Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'11)*. ACM, New York, NY, 52.

V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti. 2015. Noxim: An open, extensible and cycle-accurate network on chip simulator. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 162–163.

Sangyeun Cho and Lei Jin. 2006. Managing distributed, shared L2 caches through OS-level page allocation. In *Proceedings of the International Symposium on Microarchitecture (MICRO'06)*.

Erik D. Demaine and Martin L. Demaine. 2007. Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity. *Graphs and Combinatorics* 23, 1, 195–208.

Erik D. Demaine and Michael Hoffmann. 2001. Pushing blocks is NP-complete for noncrossing solution paths. In *Proceedings of the Canadian Conference on Computational Geometry*.

Pierre-François Dutot, Grégory Mounié, and Denis Trystram. 2004. Scheduling parallel tasks: Approximation algorithms. In *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Boca Raton, FL, 26-1.

T. Ebi, M. Faruque, and J. Henkel. 2009. TAPE: Thermal-aware agent-based power economy multi/many-core architectures. In *Proceedings of the 2009 International Conference on Computer Aided Design (ICCAD'09)*.

Al Faruque, Mohammad Abdullah, Rudolf Krist, and Jörg Henkel. 2008. ADAM: Run-time agent-based distributed application mapping for on-chip communication. In *Proceedings of the 2008 Design Automation Conference (DAC'08)*.

Mohammad Fattah, Masoud Daneshtalab, Pasi Liljeberg, and Juha Plosila. 2013. Smart hill climbing for agile dynamic mapping in many-core systems. In *Proceedings of the 2013 Design Automation Conference (DAC'13)*.

Mohammad Fattah, Pasi Liljeberg, Juha Plosila, and Hannu Tenhunen. 2014. Adjustable contiguity of run-time task allocation in networked many-core systems. In *Proceedings of the 2014 Asia and South Pacific Design Automation Conference (ASP-DAC'14)*.

Dror G. Feitelson and Larry Rudolph. 1998. Metrics and benchmarking for parallel job scheduling. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (IPPS/SPDP'98)*. 1–24.

Mo Guan and Minghai Gu. 2010. Design and implementation of an embedded Web server based on ARM. In *Proceedings of the 2010 International Conference on Software Engineering and Service Sciences (ICSESS'10)*.

Jörg Henkel, Andreas Herkersdorf, Lars Bauer, Thomas Wild, Michael Hübner, Ravi Kumar Pujari, Artjom Grudnitsky, et al. 2012. Invasive manycore architectures. In *Proceedings of the 2012 Asia and South Pacific Design Automation Conference (ASP-DAC'12)*.

Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. 2012. A case for exploiting subarray-level parallelism (SALP) in DRAM. In *Proceedings of the 2012 International Symposium on Computer Architecture (ISCA'12)*. IEEE, Los Alamitos, CA.

Kenneth C. Knowlton. 1965. A fast storage allocator. *Communications of the ACM* 8, 10, 623–624.

Keqin Li and Kam Hoi Cheng. 1990. A two dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system. In *Proceedings of the 1990 Annual Computer Science Conference (CSC'90)*.

Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. 2007. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the International Conference on Supercomputing (SC'07)*.

Jim Ng, Xiaohang Wang, Amit Kumar Singh, and Terrence Mak. 2016. Defragmentation for efficient runtime resource management in NoC-based many-core systems. *Transactions on Very Large Scale Integration (VLSI) Systems* 24, 11, 3359–3372.

Venkatesh Pallipadi and Alexey Starikovskiy. 2006. The ondemand governor. In *Proceedings of the 2006 Linux Symposium*.

Anuj Pathania, Vanchinathan Venkataramani, Muhammad Shafique, Tulika Mitra, and Jörg Henkel. 2016. Distributed scheduling for many-cores using cooperative game theory. In *Proceedings of the 2016 Design Automation Conference (DAC'16)*.

Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. 2004. PIN: A binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 International Symposium on Computer Architecture (ISCA'04)*.

Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. 2013. Mapping on multi/many-core systems: Survey of current and emerging trends. In *Proceedings of the 2013 Design Automation Conference (DAC'13)*.

Neil J. A. Sloane. 2003. The On-Line Encyclopedia of Integer Sequences. Retrieved February 14, 2017, from http://oeis.org.

Jonathan A. Winter, David H. Albonesi, and Christine A. Shoemaker. 2010. Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *Proceedings of the 2010 International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*.

Received June 2016; revised October 2016; accepted November 2016