Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems

Reetuparna Das* Rachata Ausavarungnirun† Onur Mutlu† Akhilesh Kumar‡ Mani Azimi‡
University of Michigan* Carnegie Mellon University† Intel Labs‡

Abstract

Future many-core processors are likely to concurrently execute a large number of diverse applications. How these applications are mapped to cores largely determines the interference between these applications in critical shared hardware resources. This paper proposes new application-to-core mapping policies to improve system performance by reducing inter-application interference in the on-chip network and memory controllers. The major new ideas of our policies are to: 1) map network-latency-sensitive applications to separate parts of the network from network-bandwidth-intensive applications such that the former can make fast progress without heavy interference from the latter, 2) map those applications that benefit more from being closer to the memory controllers close to these resources.

Our evaluations show that, averaged over 128 multiprogrammed workloads of 35 different benchmarks running on a 64-core system, our final application-to-core mapping policy improves system throughput by 16.7% over a state-of-the-art baseline, while also reducing system unfairness by 22.4% and average interconnect power consumption by 52.3%.

1. Introduction

One important use of multi-core systems is to concurrently run many diverse applications. Managing critical shared resources, such as the on-chip network (NoC), among co-scheduled applications is a fundamental challenge. In a large many-core processor, which core is selected to execute an application could have a significant impact on system performance because it affects contention and interference among applications. Performance of an application critically depends on how its network packets *interfere* with other applications' packets in the interconnect and memory, and how far away it is executing from shared resources such as memory controllers. Which core an application is mapped to in a NoC-based multi-core system significantly affects both the interference patterns between applications as well as applications' distance from the memory controllers. Hence, the application-to-core mapping policy can have a significant impact on both per-application performance and system performance, as we will empirically demonstrate in this paper.

While prior research (e.g., [21, 29, 33]) tackled the problem of how to map tasks/threads *within* an application, the interference behavior *between* applications in the NoC is less well understood. Current operating systems are unaware of the on-chip interconnect topology and application interference characteristics at any instant of time, and employ naive methods while mapping applications to cores. For instance, on a non-NUMA system that runs Linux 2.6.x [1], the system assigns a static numbering to cores and chooses the numerically-smallest core when allocating an idle core to an application.[1] This leads to an application-to-core mapping that is oblivious to application characteristics and inter-application interference, causing two major problems that we aim to solve in this paper.

First, overall performance degrades when applications that interfere significantly with each other in the shared resources get mapped to closeby cores. Second, an application may benefit significantly from being mapped to a core that is close to a shared resource (e.g., a memory controller), yet it can be mapped far away from that resource (while another application that does not benefit from being close to the resource is mapped closeby the resource), reducing system performance. To solve these two problems, in this work, we develop intelligent application-to-core mapping policies that are aware of application characteristics and on-chip interconnect topology.

Our new application-to-core mapping policies are built upon two major observations. First, we observe some applications are more *sensitive* to interference than others: when interfered with, network-sensitive applications slow down more significantly than others [12]. Thus, system performance can be improved by separating (i.e., mapping far away) network-sensitive applications from aggressive applications that have high demand for network bandwidth. To allow this separation of applications, we partition the cores into clusters such that the cores in a cluster predominantly access the memory controller(s) in the same cluster, develop heuristics to estimate each application's network sensitivity, and devise algorithms that use these estimates to distribute applications to clusters. While partitioning applications among clusters to reduce interference, our algorithms also try to balance the network load among clusters as much as possible.

Second, we observe that an application that is *both memory-intensive and network-sensitive* gains more performance from being close to a memory controller than one that does not have either of the properties (as the former needs fast, high-bandwidth memory access). Thus, system performance can be improved by mapping such applications to cores close to memory controllers. To this end, we develop heuristics to identify such applications dynamically and devise a new algorithm that maps applications to cores within each cluster based on each application's performance sensitivity to distance from the memory controller.

We make the following new contributions in this paper:

- We develop a new core assignment algorithm for applications in a NoC-based many-core system, which aims to minimize destructive inter-application network interference and thereby maximize overall system performance.

[1] The Linux kernels with NUMA extensions use a mapping similar to our CLUSTER+RND scheme that we discuss and compare to in Section 6.1.

978-1-4673-5587-2/13/$31.00 ©2013 IEEE

- We develop new insights on inter-application interference in NoC based systems, which form the foundation of our algorithm. In particular, we demonstrate that 1) mapping network-sensitive applications to parts of the network such that they do not receive significant interference from network-intensive applications and 2) mapping memory-intensive and network-sensitive applications close to the memory controller can significantly improve performance. We show that sometimes reducing network load balance to isolate network-sensitive applications can be beneficial.
- We demonstrate that intelligent application-to-core mappings can save network energy significantly. By separating applications into clusters and placing memory-intensive applications closer to the memory controllers, our techniques reduce the communication distance and hence communication energy of applications that are responsible for the highest fraction of overall network load.
- We extensively evaluate the proposed application-to-core mapping policies on a 60-core CMP with an 8x8 mesh NoC using a suite of 35 diverse applications. Averaged over 128 randomly generated multiprogrammed workloads, our final proposed policy improves system throughput by 16.7% over a state-of-the-art baseline, while also reducing application-level unfairness by 22.4% and NoC power consumption by 52.3%.

2. Motivation and Overview

A many-core processor with n cores can run n concurrent applications. Each of these applications can be mapped to any of n cores. Thus, there can be $n!$ possible mappings. From the interconnect perspective, an application-to-core mapping can determine the degree of interference of an application with other applications in the NoC as well as how well the overall network load is balanced. The application-to-core mapping also determines how the application's memory accesses are distributed among the memory controllers. These factors can lead to significant variation in performance.

For example, Figure 1 shows the system performance for 576 different application-to-core mappings for the same workload (detailed system configuration is given in Section 5). The workload consists of 10 copies each of applications `gcc`, `barnes`, `soplex`, `lbm`, `milc` and `leslie` running together and an application's memory accesses are mapped to the nearest memory controller. The Y-axis shows the system performance in terms of normalized weighted speedup of the different mappings (higher is better). Each data point represents a different mapping. It can be seen that the best possible mapping provides 24% higher system performance than the worst possible mapping.

To understand the behavior of these mappings, we systematically analyze the characteristics of the best performing mappings across a wide range of workloads. Figure 2 illustrates the four general categories of mappings on our baseline tiled manycore architecture. Figure 2(a) shows the logical layout of a manycore processor with cores organized in an 8x8 mesh on-chip network. The memory controllers (triangles) are located in corner tiles (*we also evaluate other memory controller placements in Section 6.7*). All other tiles consist of a core, L1 cache and L2 cache bank. Further, we divide the 8x8 mesh into four 4x4 clusters (marked by dotted lines). An application can be mapped to any of the 60 cores. We assume that integrated software and hardware techniques (discussed in Section 3.1) allow us to restrict most of an application's on-chip network traffic to the cluster where it is running.

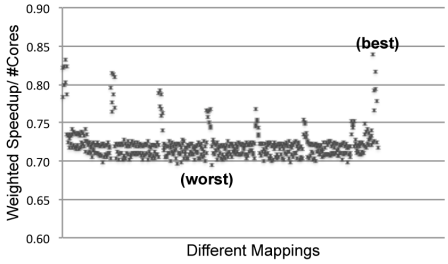


Figure 1: Performance of 576 different application to core mappings for one multiprogrammed workload (8x8 system)

Figure 2(b), (c), (d), (e) show four possible application to core mappings. Each core tile in the figure is shaded according to the network intensity of the application running on it; a darker tile corresponds to an application with a higher L1 Misses per Thousand Instructions (MPKI), running on the core in the tile.

Figure 2 (b) shows a possible random mapping of applications to clusters (called RND), which is the baseline mapping policy in existing general-purpose systems. Unfortunately, a random mapping does not take into account the balance of network and memory load across clusters, and as a result leads to high contention for available bandwidth and low system performance.

Figure 2 (c) shows an example where applications are completely imbalanced between clusters (called IMBL). The upper left cluster is heavily loaded, or filled with application with high MPKI, while the lower right cluster has very low load. An imbalanced inter-cluster mapping can severely degrade system performance because of poor utilization of aggregate available bandwidth in the NoC and in the off-chip memory channels. Such an undesirable imbalanced mapping is possible in existing systems. Hence we need to develop policies to prevent such scenarios.

Figure 2 (d) illustrates an example of a balanced mapping (called BL) which equally divides the load among different clusters. This mapping can achieve better performance due to effective utilization of network and memory channel bandwidth. However, we find that a balanced mapping is not the best performing mapping. We observe that some applications are more sensitive to interference in the network compared to other applications. In other words, these applications experience greater slowdowns than other applications from the same amount of interference in the shared network. Thus, system performance can be improved by separating (i.e., mapping far away) network-sensitive applications from aggressive applications that have high demand for network bandwidth.

Figure 2 (e) illustrates our balanced mapping with reduced interference (called BLRI). The BLRI mapping attempts to pro-

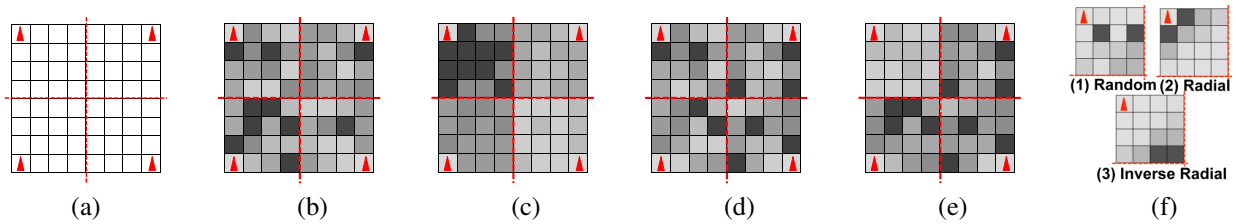


Figure 2: (a) A manycore processor organized in an 8x8 mesh network-on-chip divided into four clusters. Each cluster has a memory controller tile and core tiles. Memory controllers (triangle) are placed in corner tiles (alternative memory controller placement is explored in Section 6.7).The core tiles have a processor core, L1 cache and L2 cache. Different possible mappings: (b) Random (RND) mapping (c) Imbalanced (IMBL) (d) Balanced (BL) (e) Balanced with Reduce Interference (BLRI). Darker shades indicate applications that have higher network intensity (or injection rate). Three example mappings within a cluster are shown in (f): (1) Random (2) Radial (3) Inverse Radial.

tect interference-sensitive applications from other applications by assigning them to their own cluster (the top left cluster in Figure 2(e)) while trying to preserve load balance in the rest of the network as much as possible. BLRI can improve performance significantly by keeping the slowdowns of the most latency-sensitive applications under control.

After mapping applications to different clusters, a question remains: *which core within a cluster* should an application be mapped to? Figure 2 (f) shows three possible intra-cluster mappings for a single cluster. Figure 2 (f) (1) depicts a random intra-cluster mapping; this is not the best intra-cluster mapping as it is agnostic to application characteristics. The applications placed closer to memory controller enjoy higher throughput and lower latency, as they have to travel shorter distances. Thus, it is beneficial to place network-intensive applications (which benefit more from higher throughput) and network-sensitive application (which benefit from lower latency) close to memory controller. Figure 2 (f) (2) shows an example mapping within a cluster where applications are placed *radially* in concentric circles around the memory controller. Darker (inner and closer) tiles represent network-intensive and interference-sensitive applications and lighter (outer and farther) tiles represent lower intensity applications with low sensitivity. Figure 2 (f) (3) shows an example of an opposite policy which we will evaluate, referred to as *inverse-radial*.

3. Mapping Policies

In this section we develop policies for efficient application-to-core mappings. First, cores are clustered into subnetworks to reduce interference between applications mapped to different clusters (Section 3.1). Second, applications are distributed between clusters such that overall network load is balanced between clusters (Section 3.2.1), while separating the interference-sensitive applications from interference-insensitive ones (Section 3.2.2). Finally, after applications have been distributed between clusters, intra-cluster radial mapping (Figure 2 (f) (2)) is used to map applications to cores within a cluster such that applications that benefit most from being close to the memory controller are placed closest to the memory controller (Section 3.3).

3.1. Cluster Formation

What is a cluster? We define a cluster as a sub-network such that the majority of network traffic originating in the sub-network can be constrained within the sub-network. In our policies, cores are clustered into subnetworks to reduce interference between applications mapped to different clusters. Applications mapped to a cluster predominantly access the memory controller within that cluster (and share the L2 cache slices within that cluster). Clustering not only reduces interference between applications mapped to different clusters, but also 1) reduces overall congestion in the network, 2) reduces average distance packets traverse to get to the memory controllers or shared caches.

How to enforce clustering for memory accesses? Clustering can be achieved by mapping physical pages requested by cores to memory controllers in an appropriate manner. Typically, physical pages (or even cache blocks) are *interleaved* among memory controllers such that adjacent pages (or cache blocks) are mapped to different memory controllers [43, 36, 26, 25, 34]. To enable clustering, page allocation and replacement policies should be modified such that data requested by a core is opportunistically mapped to the home MC of the core. To achieve this, we slightly modify the commonly-used CLOCK [22] page allocation and replacement algorithm to what we call the *cluster-CLOCK* algorithm.

When a page fault occurs and free pages exist, the operating system gives preference to free pages belonging to the home MC of a requesting core when allocating the new page to the requesting core. If no free pages belonging to the home MC exist, a free page from another MC is allocated. When a page fault occurs and no free page exists, preference is given to a page belonging to the home MC, while finding the replacement page candidate. We look N pages beyond the default candidate found by CLOCK [22] algorithm to find a page that belongs to home MC.[2] If unsuccessful in finding a replacement candidate belonging to home MC when N pages beyond the default candidate, the algorithm simply selects the default candidate for replacement.

The above modifications ensure that the new page replacement policy does not significantly perturb the existing replacement order, and at the same time opportunistically achieves the effect of clustering. Note that these modifications to virtual memory management (for both page allocation and page replacement) *do not enforce a static partitioning of DRAM memory capacity*; they only *bias* the page replacement policy such that it likely allocates pages to a core from the core's home memory controller.

[2]We use $N = 512$ in our experiments, a value empirically determined to maximize the possibility of finding a page in home MC while minimizing the overhead of searching for one.

How to enforce clustering for cache accesses? Clustering can be enforced for cache accesses for shared cache architectures by slightly modifying state-of-the-art cooperative caching techniques [7, 38] or page coloring techniques [9] such that cache accesses of a core remain in the shared cache slices within the core's cluster. Such caching techniques have been shown to improve system performance by trading-off cache capacity for communication locality. For most of our evaluations, we use private caches that automatically enforce clustering by restricting the cache accesses from an application to the private cache co-located in the same tile as the core running the application. For completeness, we evaluate our techniques on a shared cache architecture in Section 6.7, in which clustering is achieved by implementing Dynamic Spill Receive (DSR) caches [38].

3.2. Mapping Policy between Clusters

In this subsection, we devise clustering algorithms that decide *to which cluster* an application should be mapped to. The key idea is to map the network-sensitive applications to a separate cluster such that they suffer minimal interference. While doing so, we also try to ensure that overall network load is balanced between clusters as much as possible.

3.2.1. Balanced Mapping (BL) The goal of the *Balanced (BL)* policy (an example of which is shown in Figure 2 (d)) is to balance load between clusters, such that there is better overall utilization of network and memory bandwidth. For BL mapping, we start with the list of applications *sorted with respect to their network intensity*. Network intensity is measured in terms of injection rate into the network, which is quantified by private L2 cache MPKI, collected periodically at run-time or provided statically before the application starts. The BL mapping is achieved by mapping consecutively-ranked applications to different clusters in a round robin fashion.

3.2.2. Balanced with Reduced Interference Mapping (BLRI) The goal of the *Balanced with Reduced Interference (BLRI)* policy (an example of which is shown in Figure 2 (e)) is to protect interference-sensitive applications from other applications by assigning them their own cluster (the top left cluster in the figure) while trying to balance load in the rest of the network as much as possible. To achieve this, our mechanism: 1) identifies sensitive applications 2) maps the sensitive applications to their own cluster *only if* there are enough such applications *and* the overall load in the network is high enough to cause congestion.

How to Identify Sensitive Applications We characterize applications to investigate their relative sensitivity. Our studies show that interference-sensitive applications have two main characteristics. First, they have low memory level parallelism (MLP) [16, 35]: such applications are in general more sensitive to interference because any delay for the application's network packet is likely cause extra stalls, as there is little or no overlap of packet latencies. Second, they inject enough load into the network for the network interference to make a difference in their execution time. In other words, applications with very low network intensity are not sensitive because their performance does not significantly change due to extra delay in the network.

We use two metrics to identify interference-sensitive applications. We find that *Stall Time per Miss* (STPM) metric correlates with MLP. STPM is the average number of cycles for which a core is stalled because it is waiting for a cache miss packet to return from the network. Relative STPM is an application's STPM value normalized to the minimum STPM among all applications to be mapped. Applications with high relative STPM are likely to have relatively low MLP. Such applications are classified as sensitive only if they inject enough load into network, i.e., if their L1 cache MPKI is greater than a threshold.

How to Decide Whether or Not to Allocate a Cluster for Sensitive Applications After identifying sensitive applications, our technique tests if a separate cluster should be allocated for them. This cluster is called the $RI_{cluster}$, which stands for *Reduced-Interference Cluster*. There are three conditions that need to be satisfied for this cluster to be formed:

- First, there have to be enough sensitive applications to fill majority of the cores in a cluster. This condition ensures that there are enough sensitive applications such that their separation from others actually reduces overall interference significantly.[3]
- Second, the entire workload should exert a large amount of pressure on the network. We found that allocating interference-sensitive applications to their own cluster makes sense only for workloads that have a mixture of interference-sensitive applications and network-intensive (high-MPKI) applications that can cause severe interference by pushing the network towards saturation. As a result, our algorithm considers forming an $RI_{cluster}$ if the aggregate bandwidth demand of the entire workload is higher than 1500 MPKI (determined empirically).
- Third, the aggregate MPKI of the $RI_{cluster}$ should be small enough so that interference-sensitive applications mapped to it do not significantly slow down each other. If separating applications to an $RI_{cluster}$ ends up causing too much interference within the $RI_{cluster}$, this would defeat the purpose of forming the $RI_{cluster}$ in the first place. To avoid this problem, our algorithm does not form an $RI_{cluster}$ if the aggregate MPKI of $RI_{cluster}$ exceeds the bandwidth capacity of any NoC channel.[4]

If any of these three criteria are not satisfied, Balanced Load (BL) algorithm is used to perform mapping in all clusters, without forming a separate $RI_{cluster}$.

How to Map Sensitive Applications to Their Own Cluster The goal of the Reduced Interference (RI) algorithm is to fill the $RI_{cluster}$ with as many sensitive applications as possible, as long as the aggregate MPKI of $RI_{cluster}$ does not exceed the capacity of any NoC channel. The problem of choosing p (p = number of cores in a cluster) sensitive applications that have an aggregate MPKI less than a upper bound, while maximiz-

[3] We empirically found that at least 75% of the cores in the cluster should run sensitive applications.

[4] For example. in our configuration, each NoC channel has a capacity of 32 GB/s. Assuming 64 byte cache lines and throughput demand of one memory access per cycle at core frequency of 2 GHz, 32 GB/s translates to 250 MPKI. Thus the aggregate MPKI of $RI_{cluster}$ should be less than 250 MPKI ($Thresh_{MPKIRI}$ = 250 MPKI).

ing aggregate sensitivity of $RI_{cluster}$ can be easily shown to be equivalent to the *0-1 knapsack problem* [10]. We use a simple solution described in [10] to the knapsack problem to choose p sensitive applications. In case there are fewer sensitive applications than p, we pack the empty cores in the $RI_{cluster}$ with the *insensitive applications that are the least network-intensive*.[5]

Forming More than One $RI_{cluster}$ If the number of sensitive applications is more than $2p$, then our technique forms two clusters.

Once the $RI_{cluster}$ has been formed, the rest of the applications are mapped to the remaining clusters using the BL algorithm. We call this final inter-cluster mapping algorithm, which **combines RI and BL algorithms**, as the **BLRI** algorithm.

3.3. Mapping Policy within a Cluster

Our intra-cluster algorithm decides *which core within a cluster* should an application be mapped to. We observed in Section 2 that memory-intensive and network-sensitive applications benefit more from placement closer to memory controllers. Thus our proposed mapping maps these applications radially around the memory controllers. Figure 2 (f) (2) shows an example the mapping achieved by our proposal. Our intra-cluster algorithm differentiates applications based on *both* their 1) *network/memory demand* (i.e. rate of injection of packets) measured as MPKI, and 2) *sensitivity* to network latency measured as STPM at the core. It then computes a metric, stall time per thousand instructions, $STPKI = MPKI * STPM$ for each application, and sorts applications based on the value of this metric. Applications with higher $STPKI$ are assigned to cores closer to the memory controller. To achieve this, the algorithm maps applications radially in concentric circles around the memory controller in sorted order, starting from the application with the highest $STPKI$.

3.4. Putting It All Together: Our Application-to-Core (A2C) Mapping Algorithm

Our final algorithm consists of three steps combining the above algorithms. First, cores are clustered into subnetworks using the cluster-CLOCK page mapping algorithm (Section 3.1). Second, the BLRI algorithm is invoked to map applications to clusters (Section 3.2.2). In other words, the algorithm attempts to allocate a separate cluster to interference-sensitive applications (Section 3.2.2), if possible, and distributes the applications to remaining clusters to balance load (Section 3.2.1). Third, after applications are assigned to clusters, they are mapped to cores within their clusters by invoking the intra-cluster radial mapping algorithm (Section 3.3). **We call this final algorithm, which combines clustering, BLRI and radial algorithms, as the *A2C mapping algorithm*.**

4. Implementation

4.1. Profiling

The proposed mapping policies assume knowledge of two metrics: a) network demand in terms of L2 MPKI and b) sensitivity in terms of STPM. These metrics can be either pre-computed for applications *a priori* or measured online during a profiling phase. We evaluate both scenarios, where metrics are known *a priori* (static A2C) and when metrics are measured online (dynamic A2C). For dynamic A2C, we profile the workload for 10 million instructions (profiling phase) and then compute mappings that are enforced for 300 million instructions (enforcement phase). The profiling phase and enforcement phases are repeated periodically. The profiling to determine MPKI requires two hardware counters in the core: 1) instruction counter and 2) L2 miss counter. The profiling to determine STPM requires one additional hardware counter at the core which is incremented every cycle the oldest instruction cannot be retired because it is waiting for an L2 miss. Note that the A2C technique requires only a relative ordering among applications and hence we find quantizing applications to classes based on the above metrics is sufficient.

4.2. Operating System and Firmware Support

Our proposal requires support from the operating system. First, the operating system page allocation and replacement routine is modified to enforce clustering, as described in Section 3.1. Second, the A2C algorithm can be integrated as part of the operating system task scheduler. If this is the case, the OS scheduler allocates cores to applications based on the optimized mapping computed by the A2C algorithm. The complexity of the algorithm is relatively modest and we found its time overhead is negligible since the algorithm needs to be invoked very infrequently (e.g., every OS time quantum). Alternatively, the A2C algorithm can be implemented as part of the firmware of a multi-core processor.

The spatial decisions of *where* to schedule applications made by the A2C algorithm can be used to augment the OS time-sharing scheduler's temporal decisions of *when* to schedule an application. When a context switch is needed, the OS can invoke A2C to determine if there is a need for migration and can decide whether or not to perform migrations based on a cost-benefit analysis.

4.3. Adapting to Dynamic Runtime Environment

The runtime environment of a manycore processor will be dynamic with continuous flow of incoming programs (process creation), outgoing programs (process completion), and context switches. Thus, it is hard to predict *a priori which* set of applications will run simultaneously as a workload on the manycore processor. Our application-to-core mapping techniques have the capability to adapt to such dynamic scenarios via two key elements. First, application characteristics can be determined during runtime (Section 4.1). Second, since application-to-core mapping of an application can change between different execution phases, we migrate applications between cores to enforce new mappings. We discuss the costs of application migration in the next subsection.

4.4. Migration Costs

A new application-to-core mapping may require migration of an application from one core to another. We can split the cost associated with application migration into four parts: 1) A constant cost due to operating system bookkeeping to facilitate migration. This cost is negligible because all cores are managed by a single unified operating system. For example, the file handling, memory management, IO, network socket state, etc are

[5] Note that this solution does not have high overhead as our algorithm is invoked at long time intervals at the granularity of OS time quantums.

Processor Pipeline	2 GHz processor, 128-entry instruction window
Fetch/Exec/Commit width	2 instructions per cycle in each core; only 1 can be a memory operation
Memory Management	4KB physical and virtual pages, 512 entry TLBs, CLOCK page allocation and replacement
L1 Caches	32KB per-core (private), 4-way set associative, 64B block size, 2-cycle latency, split I/D caches, 32 MSHRs
L2 Caches	256KB per core (private), 16-way set associative, 64B block size, 6-cycle bank latency, 32 MSHRs, directories are co-located with the memory controller
Main Memory	16GB DDR3-DRAM, up to 16 outstanding requests per-core, 160 cycle access, 4 on-chip Memory Controllers.
Network Router	2-stage wormhole switched, virtual channel flow control, 4 VC's per Port, 4 flit buffer depth, 4 flits per data packet, 1 flit per address packet.
Network Interface	16 FIFO buffer queues with 4 flit depth
Network Topology	8x8 mesh, 128 bit bi-directional links (32GB/s).

Table 1: Baseline Processor, Cache, Memory, and Network Configuration

Functional Simulations for Page Fault Rates: We run 500 million instructions per core (totally 30 billion instructions across 60 cores) from the simulation fast forward point obtained from [37] to evaluate cluster-CLOCK and CLOCK page replacement algorithms.
Performance Simulations for Static A2C mapping: To have tractable simulation time, we choose a smaller representative window of instructions, obtained by profiling each benchmark, from the representative execution phase obtained from [37]. All our experiments study multi-programmed workloads, where each core runs a separate benchmark. We simulate 10 million instructions per core, corresponding to at least 600 million instructions across 60 cores.
Performance Simulations for Dynamic A2C mapping: To evaluate the dynamic application-to-core mapping faithfully, we need longer simulations that model at least one dynamic profiling phase and one enforcement phase (as explained in Section 4.1). We simulate an entire profiling+enforcement phase (300 million instructions) per benchmark per core, corresponding to at least 18.6 billion instructions across 60 cores.

Table 2: Simulation Methodology

shared between the cores due to the single operating system image and need not be saved or restored; 2) A constant cost (in terms of bytes) for transferring the application's architectural context (including registers) to the new core; 3) A variable cache warmup cost due to cache misses incurred after transferring the application to a new core. We quantify this cost and show that averaged over the entire execution phase, this cost is negligibly small across all benchmarks (see Section 6.5); and 4) A variable cost due to potential reduction in *clustering factor*.[6] This cost is incurred only when we migrate applications between clusters and, after migration, the application continues to access pages mapped to its old cluster. We quantify this cost in our performance evaluation for all our workloads (see Section 6.5). Our evaluations faithfully account for all of these four types of migration costs.

5. Methodology

5.1. Experimental Setup

We evaluate our techniques using an instruction-trace-driven, cycle-level x86 CMP simulator. The functional frontend of the simulator is based on the Pin dynamic binary instrumentation tool [31], which is used to collect instruction traces from applications, which are then fed to the core models that model the execution and timing of each instruction.

Table 1 provides the configuration of our baseline, which consists of 60 cores and 4 memory controllers connected by a 2D, 8x8 mesh NoC. Each core is modeled as an out-of-order execution core with a limited instruction window and limited buffers. The memory hierarchy uses a two-level directory-based MESI cache coherence protocol. Each core has a private write-back L1 cache and private L2 cache. The memory controllers connect to off-chip DRAM via DDR channels and have sufficient buffering to hold 16 outstanding requests per core. The bandwidth of DDR channels is accurately modelled. The network connects the core tiles and memory controller tiles. The system we model is self-throttling as real systems are: if the miss buffers are full the core cannot inject more packets into the network. Each router uses a state-of-the-art two-stage microarchitecture. We use the deterministic X-Y routing algorithm, finite input buffering, wormhole switching, and virtual-channel flow control. We use the Orion power model to estimate the router power [42].

We also implemented a detailed functional model for virtual memory management to study page access and page fault behavior of our workloads. The baseline page allocation and replacement policy is CLOCK [22]. The modified page replacement and allocation policy, cluster-CLOCK, looks ahead 512 pages beyond the first replacement candidate to potentially find a replacement page belonging to home memory controller.

The parameters used for our A2C algorithm are: $Thresh_{MPILow}$ = 5 MPKI, and $Thresh_{MPIHigh}$ = 25 MPKI, $Thresh_{SensitivityRatio}$ = 5 and $Thresh_{MPKI-RI}$ = 250 MPKI. These parameters are empirically determined but not tuned. The constant cost for OS book keeping while migrating applications is assumed to be 50K cycles. The migrating applications write and read 128 bytes to/from the memory to save and restore their register contexts. These are modelled as extra memory accesses in our simulations.

5.2. Evaluation Metrics

Our evaluation uses several metrics. We measure **system performance** in terms of average **weighted speedup** [14], a commonly used multi-program performance metric, which is the average of the sum of speedups of each application compared to when it is run alone on the same system. We measure **system fairness** in terms of the maximum slowdown any application experiences in the system. We also report **IPC throughput**. IPC_{alone} is the IPC of the application when run alone on our baseline system.

$$(Average)\ Weighted\ Speedup = \frac{1}{NumThreads} \times \sum_{i=1}^{NumThreads} \frac{IPC_i^{shared}}{IPC_i^{alone}} \quad (1)$$

$$IPC\ Throughput = \sum_{i=1}^{NumThreads} IPC_i \quad (2)$$

$$UnfairnessIndex = \max_i \frac{IPC_i^{alone}}{IPC_i^{shared}} \quad (3)$$

5.3. Workloads and Simulation Methodology

Table 2 describes the three different types of simulations we run to evaluate our mechanisms. Due to simulation time limitations, we evaluate the execution time effects of our static

[6] Clustering factor is defined as the percentile of accesses that are constrained within the cluster.

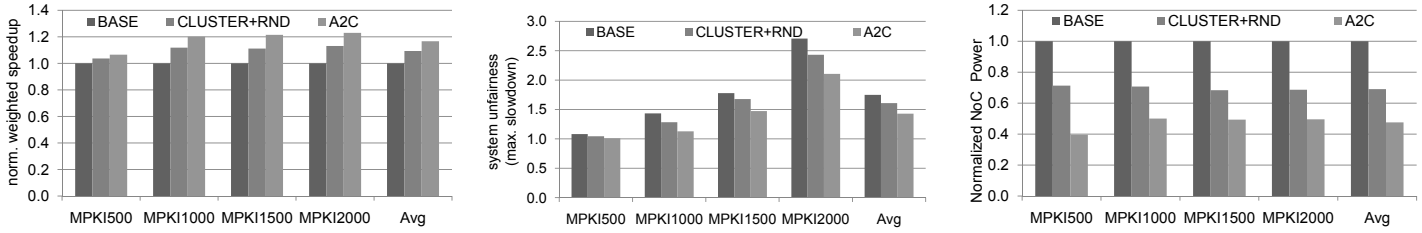
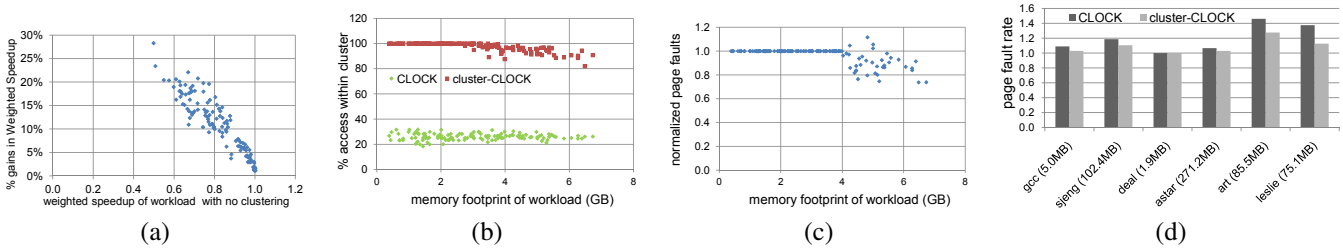


Figure 3: (a) System performance (b) system unfairness and (c) interconnect power of the A2C algorithm for 128 workloads



(a) (b) (c) (d)

Figure 4: Performance analysis of clustering across 128 workloads (a) Weighted speedup (b) Clustering Factor (c) Page Faults (d) A case study showing page faults per unique page accessed for different applications

and dynamic mechanisms without taking into account the effect on page faults. We evaluate the effect of our proposal on page fault rate separately using functional simulation, showing empirically that our proposal actually reduces page fault rate (Section 6.2).

We use a diverse set of multiprogrammed application workloads comprising scientific, commercial, and desktop applications. In total, we study 35 applications, including SPEC CPU2006 benchmarks, applications from SPLASH-2 and SpecOMP benchmark suites, and four commercial workloads traces (`sap`, `tpcw`,`sjbb`, `sjas`). We choose representative execution phases using PinPoints [37] for all our workloads except commercial traces, which were collected over Intel servers. Figure 6 (b) lists each application and includes results showing the MPKI of each application on our baseline system.

Multiprogrammed Workloads and Categories: All our results are across 128 randomly generated workloads mixes. Each workload mix consists of 10 copies each of 6 applications randomly picked from our suite of 35 applications. The 128 workloads are divided into four subcategories of 32 workloads each: 1) MPKI500: relatively less network-intensive workloads with aggregate MPKI less than 500, 2) MPKI1000: aggregate MPKI is between 500-1000, 3) MPKI1500: aggregate MPKI is between 1000-1500, 4) MPKI2000: relatively more network-intensive workloads with aggregate MPKI between 1500-2000.

6. Performance Evaluation

6.1. Overall Results for the A2C Algorithm

We first show the overall results of our final Application-to-Core Mapping algorithm (A2C). We evaluate three systems: 1) the baseline system with random mapping of applications to cores (BASE), which is representative of existing systems, 2) our enhanced system which uses our cluster-CLOCK algorithm (described in Section 3.1) but still uses random mapping of applications to cores (CLUSTER+RND), 3) our final system which uses our combined A2C algorithm (summarized in Section 3.4), which consists of clustering, inter-cluster BLRI mapping, and intra-cluster radial mapping algorithms (A2C).

Figure 3 (a) and (b) respectively show system performance (higher is better) and system unfairness (lower is better) of the three systems. Solely using clustering (CLUSTER+RND) improves weighted speedup by 9.3% over the baseline (BASE). A2C improves weighted speedup by 16.7% over the baseline, while reducing unfairness by 22%.[7] The improvement partly due to a 39% reduction in average network packet latency (not shown in graphs).

Interconnect Power Figure 3 (c) shows the average normalized interconnect power consumption (lower is better). Clustering only reduces power consumption by 31.2%; A2C mapping reduces power consumption by 52.3% over baseline (BASE). [8] The clustering of applications to memory controllers reduces the average hop count significantly, reducing the energy spent in moving data over the interconnect. Using inter-cluster and intra-cluster mapping further reduces hop count and power consumption by ensuring that network-intensive applications are mapped close to the memory controllers and network load is balanced across controllers after clustering.

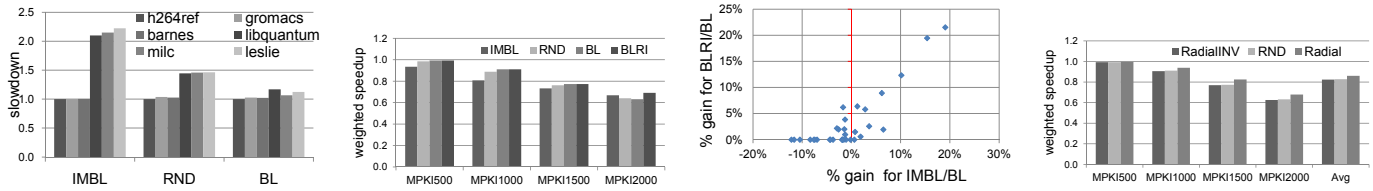
In the next three sections, we analyze the benefits and tradeoffs of each component of A2C.

6.2. Analysis of Clustering and Cluster-CLOCK

The goal of clustering is to reduce interference between applications mapped to different clusters. Averaged across 128 workloads, clustering improves system throughput by 9.3% in terms of weighted speedup and 8.0% in terms of IPC throughput. Figure 4 (a) plots the gains in weighted speedup due to clustering for each workload against the baseline weighted speedup of the workload. A lower baseline weighted speedup indicates that average slowdown of applications is higher and hence contention/interference is high between applications in the baseline. The figures show that performance gains due

[7] System performance improvement ranges from 0.1% to 22% while the unfairness reduction ranges from 6% to 72%. We do not show graphs for instruction throughput due to space limitations. Clustering alone (CLUSTER+RND) improves IPC throughput by 7.0% over the baseline, while A2C improves IPC by 14.0%.

[8] Power reduction ranges from 39% to 68%.



(a) (b) (c) (d)

Figure 5: (a) Case study analyzing BL mapping (b) Average results for different inter-cluster mappings across 128 workloads (c) Average results for BLRI mapping for the MPKI2000 workload category (d) Average results for radial mapping across 128 workloads

to clustering are significantly higher for workloads with lower weighted speedup (i.e., higher slowdowns due to interference). This is intuitive because the main goal of clustering is to reduce interference between applications.

Analysis of the cluster-CLOCK page replacement algorithm: To enforce clustering, we have developed the cluster-CLOCK algorithm (Section 3.1) which modifies the default page allocation and page replacement policies. The results in Figure 4 quantify the effect of cluster-CLOCK across 128 workloads. Figure 4 (b) plots the clustering factor with the baseline policy (CLOCK) and our new policy (cluster-CLOCK). Recall that the clustering factor is the percentage of all accesses that are serviced by the home memory controller. On average, cluster-CLOCK improves the clustering factor from 26.0% to 97.4%, thereby reducing interference among applications.

Figure 4 (c) shows the normalized page fault rate of cluster-CLOCK for each workload (Y-axis) versus the memory footprint of the workload (X-axis). A lower relative page fault rate indicates that cluster-CLOCK reduces the page fault rate compared to the baseline. We observe that cluster-CLOCK 1) does not affect the page fault rate for workloads with small memory footprint, 2) in general reduces the page fault rate for workloads with large memory footprint. On average, cluster-CLOCK reduces the page fault rate by 4.1% over 128 workloads. Cluster-CLOCK reduces page faults because it happens to make better page replacement decisions than CLOCK (i.e., replace pages that are less likely to be reused soon) by reducing the interference between applications in physical memory space: by biasing replacement decisions to be made within each memory controller as much as possible, applications mapped to different controllers interfere less with each other in the physical memory space. As a result, applications with lower page locality disturb applications with higher page locality less, improving page fault rate. Note that our execution time results do not include this effect of reduced page faults due to simulation speed limitations.

To illustrate this behavior, we focus on one workload as a case study in Figure 4 (d), which depicts the page fault rate in terms of page faults incurred per unique page accessed by each application with CLOCK and cluster-CLOCK. Applications `art` and `leslie` have higher page fault rate but we found that they have good locality in page access. On the other hand, `astar` also has high page but low locality in page access. When these applications run together using the CLOCK algorithm, `astar`'s pages contend with `art` and `leslie`'s pages in the entire physical memory space, causing those pages to be evicted from physical memory. On the other hand, if cluster-CLOCK is used, and `astar` is mapped to a different cluster from `art` and `leslie`, the likelihood that `astar`'s pages replace `art` and `leslie`'s pages reduces significantly because cluster-CLOCK attempts to replace a page from the home memory controller `astar` is assigned to instead of any page in the physical memory space. We conclude that cluster-CLOCK can reduce page fault rates by localizing page replacement and thereby limiting page-level interference among applications to pages assigned to a single memory controller.

6.3. Analysis of Inter-Cluster Mapping

We study the effect of inter-cluster load balancing algorithms described in Section 3.2. For these evaluations, all other parts of the A2C algorithm is kept the same: we employ clustering and radial intra-cluster mapping. We first show the benefits of balanced mapping (BL), then show when and why imbalanced mapping (IMBL) can be beneficial, and later evaluate our BLRI algorithm, which aims to achieve the benefits of both balance and imbalance.

BL Mapping Figure 5 (a) shows application slowdowns in an example workload consisting of 10 copies each of `h264ref`, `gromacs`, `barnes`, `libquantum`, `milc` and `leslie` applications running together. The former three are very network-insensitive and exert very low load. The latter three are both network-intensive and network-sensitive. An IMBL mapping (described in Section 3.2) severely slows down the latter three network-intensive and network-sensitive applications because they get mapped to the same clusters, causing significant interference to each other, whereas the former three applications do not utilize the bandwidth available in their clusters. A random (RND) mapping (which is our baseline) still slows down the same applications, albeit less, by providing better balance in load across the network. Our balanced (BL) mapping algorithm, which distributes load in a balanced way among all clusters provides the best speedup (19.7% over IMBL and 5.9% over RND) by reducing the slowdown of all applications because it does not overload any single cluster.

IMBL Mapping Figure 5 (b) shows average weighted speedups across 128 workloads, categorized by the overall intensity of the workloads. When the overall workload intensity is not too high (i.e., less than 1500 MPKI), BL provides significantly higher performance than IMBL and RND by reducing and balancing interference. However, when the workload intensity is very high (i.e., greater than 1500 MPKI), BL performs worse than IMBL mapping. The reason is that IMBL mapping isolates network-non-intensive (low MPKI) applications from network-intensive (high MPKI) ones by placing them into sep-

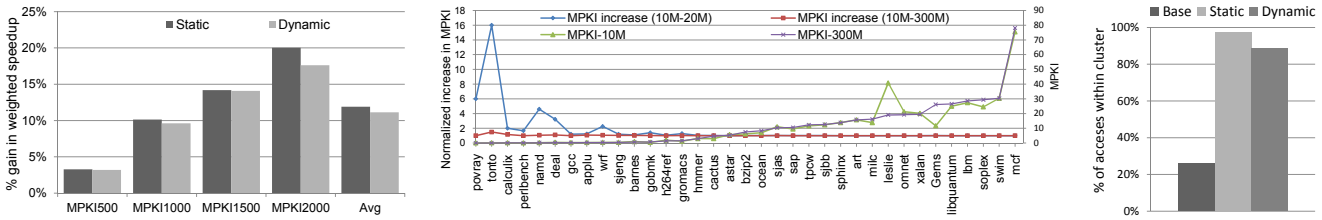


Figure 6: (a) Performance improvement of static and dynamic A2C for 8 workloads (b) Increase in MPKI due to migration for individual applications (c) Clustering factor for 128 workloads

arate clusters. When the network load is very high, such isolation significantly improves the performance of network-non-intensive applications without significantly degrading the performance of intensive ones by reducing interference between the two types of applications.

The main takeaway is that when network load is high, relatively less-intensive but network-sensitive applications' progress gets slowed down *too* significantly because other applications keep injecting interfering requests. As a result, if network load is very high, it is more beneficial to separate the accesses of non-intensive applications from others by placing them into separate clusters, thereby allowing their fast progress. However, such separation is not beneficial for non-intensive applications and harmful for performance if the network load is not high: it causes wasted bandwidth in some clusters and too much interference in others. This observation motivates our BLRI algorithm (which creates a separate cluster for non-intensive applications only when the network load is high), which we analyze next.

BLRI Mapping Figure 5 (c) shows the speedup achieved by BLRI mapping over BL for all 32 workloads in MPKI2000 category (recall that BLRI is not invoked for workloads with aggregate MPKI of less than 1500) on the Y-axis against the speedup achieved for the same workload by IMBL over BL. We make two observations. First, for workloads where imbalanced mapping (IMBL) improves performance over balanced mapping (BL), shown in the right half of the plot, BLRI also significantly improves performance over BL. Second, for workloads where imbalance reduces performance (left half of the plot), BLRI either improves performance or does not affect performance. We conclude that BLRI achieves the best of both worlds (load balance and imbalance) by isolating those applications that would most benefit from imbalance (i.e. network sensitive applications) and performing load balancing for the remaining ones.

6.4. Effect of Intra-Cluster Mapping

We analyze the effect of intra-cluster mapping policy, after applications are assigned to clusters using the BLRI inter-cluster policy. We examine three different intra-cluster mapping algorithms: 1) Radial: our proposed radial mapping described in Section 3.2, 2) RND: Cores in a cluster are assigned randomly to applications, 3) RadialINV: This is the inverse of our radial algorithm; those applications that would benefit least from being close to the memory controller (i.e., those with low STPKI) are mapped closest to the memory controller. Figure 5 (d) shows the average weighted speedup of 128 workloads with BLRI inter-cluster mapping and different intra-cluster mappings. The radial intra-cluster mapping provides 0.4%, 3.0%, 6.8%, 7.3% for MPKI500, MPKI1000, MPKI1500, MPKI2000 category workloads over RND intra-cluster mapping. Radial mapping is the best mapping for all workloads; RadialINV is the worst. We conclude that our metric and algorithm for identifying and deciding which workloads to map close to the memory controller is effective.

6.5. Effect of Dynamic A2C Mapping

Our evaluation assumed so far that a static mapping is formed with pre-runtime knowledge of application characteristics. We now evaluate the dynamic A2C scheme, described in Section 4.1. We use a profiling phase of 10 million instructions, after which the operating system forms a new application-to-core mapping and enforces it for the whole phase (300 million instructions). An application can migrate at the end of the dynamic profiling interval after each phase.

Figure 6 (a) compares the performance improvement achieved by static and dynamic A2C schemes for workloads from each MPKI category, over a baseline that employs clustering but uses random application-to-core mapping. The performance of dynamic A2C is close to that of static A2C (within 1% on average) for these eight workloads. However, static A2C performs better for the two MPKI2000 workloads. We found this is because the BLRI scheme (which determines sensitive applications online and forms a separate cluster for them) requires re-mapping at more fine-grained execution phases. Unfortunately, given the simulation time constraints we could not fully explore the best thresholds for the dynamic scheme. We conclude that the dynamic A2C scheme provides significant performance improvements, even with untuned parameters.

Migration Cost Analysis In Section 4, we qualitatively discussed the overheads of application migration. In this section, we first quantify the increases in cache misses when an application migrates from one core to another. We then quantify the reduction in clustering factor due to migrations. Overall, these results provide quantitative insight into why the dynamic A2C algorithm works.

Figure 6 (b) analyzes the MPKI of the 35 applications during the profiling phase (MPKI-10M) and the enforcement phase (MPKI-300M). It also analyzes the increase in the MPKI due to migration to another core during the 10M instruction interval right after the migration happens (MPKI increase (10-20M)) and during the entire enforcement phase (MPKI increase (300M)). The left Y-axis is the normalized MPKI of the application when it is migrated to another core compared to the MPKI when it is running alone (a value of 1 means the MPKI of the application does not change after migration).

Cache Size	256KB	512KB	1MB
Performance Gain	16.7%	13.6%	12.1%

(a)

Number of MCs	4 MC	8 MC
Performance Gain	16.7%	17.9%

(b)

Placement of MCs in a Cluster	Corner	Center
Performance Gain	16.7%	14.9%

(c)

Table 3: Sensitivity to (a) Last level cache size (per-core), (b) Number of memory controllers and (c) Placement of memory controllers within a cluster

Benchmarks on the X-axis are sorted from the lowest to highest baseline MPKI from left to right. We make several key observations:

• The MPKI in the profiling phase (MPKI-10M) correlates well with the MPKI in the enforcement phase (MPKI-300M), indicating why dynamic profiling can be effective.

• MPKI increase within 10M instructions after migration is negligible for high-intensity workloads, but significant for low-intensity workloads. However, since these benchmarks have very low MPKI to begin with, their execution time is not significantly affected.

• MPKI increase during the entire phase after migration is negligible for almost all workloads. This increase is 3% on average and again observed mainly in applications with low intensity. These results show that cache migration cost of migrating an application to another core is minimal over the enforcement phase of the new mapping.

The clustering factor (i.e., the ratio of memory accesses that are serviced by the home memory controller) is also affected by application migration. The clustering factor may potentially decrease, if, after migration, an application continues to access the pages mapped to its old cluster.

Figure 6 (c) shows the clustering factor for our 128 workloads with 1) baseline RND mapping, 2) static A2C mapping, and 3) dynamic A2C mapping. The clustering factor reduces from 97.4% with static A2C to 89.0% with dynamic A2C due to inter-cluster application migrations. However, dynamic A2C mapping still provides a large improvement in clustering factor compared to the baseline mapping. We conclude that both dynamic and static versions of A2C are effective: static A2C is desirable when application information is profiled before runtime, but dynamic A2C does not cause significant overhead and achieves similar performance.

6.6. A2C vs Application-Aware Prioritization

We compare the benefits of A2C mapping to application-aware prioritization in the network to show that our interference-aware mapping mechanisms are orthogonal to interference-aware packet scheduling in the NoC.

Das et al. [12] proposed an application-aware arbitration policy (called STC) to accelerate network-sensitive applications. The key idea is to rank applications at regular intervals based on their network intensity (outermost private cache MPKI), and prioritize packets of non-intensive applications over packets of intensive applications. We compare Application-to-Core mapping policies to application-aware prioritization policies because both techniques have similar goals: to effectively handle inter-application interference in NoC. *However, both techniques take different approaches towards the same goal*. STC tries to make the best decision when interference happens in the NoC through efficient packet scheduling in routers while A2C tries to reduce interference by mapping applications to separate clusters and controllers.

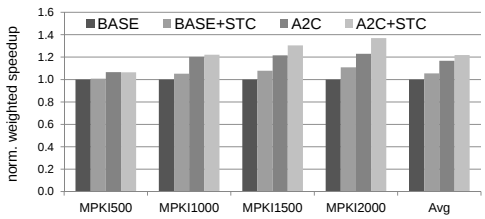


Figure 7: Performance comparison of A2C mapping and application-aware prioritization (STC) for 128 workloads.

Figure 7 shows that STC[9] is orthogonal to our proposed A2C technique and its benefits are additive to A2C. STC prioritization improves performance by 5.5% over the baseline whereas A2C mapping improves performance by 16.7% over the baseline. When STC and A2C are combined together, overall performance improvement is 21.9% over the baseline, greater than the improvement provided by either alone. STC improves performance when used with A2C mapping because it prioritizes via packet scheduling non-intensive applications (shorter jobs) within a cluster in a coordinated manner, ensuring all the routers act in tandem.The packet scheduling effect of STC is orthogonal to the benefits of A2C. Hence, we conclude that our mapping mechanisms interact synergistically with application-aware packet scheduling.

6.7. Sensitivity Studies

We evaluated the sensitivity of A2C to different system configurations. Table 3 shows A2C's performance improvement over the baseline on 128 multiprogrammed workloads.

As the **last-level cache size** increases, performance benefit of A2C decreases because NoC traffic and hence interference in the NoC reduces. However, A2C still provides 12.1% performance improvement even when total on-chip cache size is 60MB (i.e., 1MB/core).

As we vary the **number of memory controllers** from 4-8, A2C's performance benefit stays similar because more controllers 1) on the one hand enable more fine-grained clustering, improving communication locality and reducing interference, 2) on the other hand can reduce clustering factor when application behavior changes.

We also study the **placement of memory-controllers**, and change their locations from corner tiles to tiles in the center of each cluster. While this placement can reduce the contention at memory controllers, it also increases the effectiveness of intra-cluster radial mapping algorithms. Overall, A2C still provides 14.9% performance improvement with central placement of memory-controllers.

Finally, we study the effectiveness of our proposed techniques with a **shared cache**. An interleaved (striped) shared cache has poor locality because consecutive cache blocks are

[9]The default parameters used for STC [12] are: 1) ranking levels R = 8, 2) batching levels B = 8, 3) ranking interval = 350,000 cycles, 4) batching interval = 16,000 cycles, 5) BCIP packet sent every U = 4000 cycles.

Cache Configuration	Shared+DSR	Shared+DSR+A2C
Speedup over Shared+Striped	42.6%	59.5%

Table 4: Performance benefit over a shared cache

statically mapped to consecutive banks. To improve locality as well as cache utilization, we implement a shared cache using the Dynamic Spill Receive (DSR) mechanism [38]. DSR effectively balances locality with cache capacity. The key idea is to spill (i.e., have the ability to store) cache blocks of a core that benefits from more cache space than its local cache slice size to the closeby cache slices that are local to cores that do not benefit from using their entire local cache slice. In our implementation of DSR, a core's local cache slice can spill to cache slices within a 2-hop distance.

Table 4 shows the normalized performance gain with DSR and DSR combined with A2C over a striped cache mapping. As shown previously in [38], DSR improves the system performance over striped cache mapping because it allows applications that benefit more from shared cache to utilize the extra cache space effectively while also improving locality. Applying A2C on top of DSR also significantly improves system performance due to three major reasons. First, A2C reduces contention at the memory controllers and in the interconnect, which are still significant with a shared cache. Second, A2C reduces the distance that a network packet needs to travel, lowering network interference as well as packet latency. Third, A2C distributes applications with different network intensities across different clusters, thereby enabling applications that benefit from low latency (57% lower network latency) to make progress with less interference from applications that are intensive. Thus, A2C's benefits are significant in systems with shared caches.

We conclude that our proposed techniques provide consistent performance improvements when we vary major relevant system parameters.

7. Related Work

To our knowledge, this is the first work that tackles the problem of how to map competing applications to cores in a network-on-chip based memory system to minimize inter-application interference in multiprogrammed workloads. We briefly describe the closely related works in this section.

Thread-to-Core Mapping: Prior works have proposed thread-to-core mapping to improve locality of communication [29, 33] and shared cache management [8] in parallel applications by placing frequently communicating threads/tasks closer to each other. Their goal is to reduce inter-thread or inter-task communication. Our techniques solve a completely different problem: inter-application interference. As such, our goals are different: 1) reduce interference in the network between *different* independent applications and 2) reduce contention at the memory controllers. Previous thread-to-core mappings can potentially be combined with our techniques within an application when multiple applications are run in parallel, and this could be a promising future direction.

Reducing Interference in NoCs: Recent works propose quality-of-service [28, 18, 19], packet scheduling [12, 13] and routing techniques [32] to reduce interference in NoCs. These works are complementary to our mechanism.We already quantitatively showed that A2C is orthogonal to application-aware packet prioritization [12] (see Section 6.6).

Memory Controllers and Page Allocation: Awasthi et al. [4] explored page allocation and page migration in the context of multiple memory controllers in a multi-core processor with the goal of balancing load between memory controllers and improving DRAM row buffer locality. The problem we solve is different and our techniques are complementary to theirs. First, our goal is to reduce interference between applications in the interconnect and the memory controllers. Second, our mechanism addresses the problem of mapping applications to the cores while [4] balances the load at the memory controllers for a given appplication-to-core mapping. We believe the page migration techniques proposed in [4] can be employed to reduce the costs of migration in our dynamic application-to-core mapping policies. Kim et al. [25] propose Thread Cluster Memory scheduling (TCM), an application-aware memory access scheduling algorithm implemented within a memory controller. TCM does not consider the effects of core selection on contention between applications at the memory controllers, nor does it address the problem of interference in the interconnect. However, TCM can still be used as a baseline memory scheduling algorithm with our proposal.

Page allocation and migration have been explored extensively to improve locality of access within a single application executing on a NUMA multiprocessor (e.g., [6, 15]). These works do not aim to reduce interference between multiple competing applications sharing the memory system, and hence do not solve the problem we aim to solve. In addition, these works do not consider how to map applications to cores in a network-on-chip based system. Page coloring techniques have been employed in caches to reduce cache conflict misses [24, 5] and to improve cache locality in shared caches (e.g., [9]). These works solve an orthogonal problem and can be combined with our techniques.

Abts et al. [2] explore placement of memory controllers in a multi-core processor to minimize channel load. Memory controller placement is also complementary to our mechanisms, and as we showed in Section 6.7, our mechanism provides performance improvements with different memory controller placement configurations.

Thread Migration: Thread migration has been explored to manage power [39, 20], thermal hotspots [17, 11] or to exploit heterogeneity in multi-core processors [30, 23, 41, 3]. We use thread migration to enable dynamic application-to-core mappings to reduce interference in NoCs.

OS Task Scheduling: OS task schedulers [27, 44] have been proposed to reduce the contention in shared memory resources (the last level cache and memory) by co-scheduling tasks such that interference between tasks in these resources is minimized. Voltage Smoothing [40] has been proposed to co-schedule phases of different applications to mitigate voltage error recovery overheads in future resilient processor designs. In contrast to A2C, these techniques consider neither interference in the NoC nor the interference-sensitivity of different applications in the NoC when making core allocation decisions. They also

operate at a higher-level than A2C in that they select which applications concurrently share memory system resources. Our technique reduces interference between already co-scheduled applications, and hence 1) is complementary to these proposals, 2) can provide performance improvements even if a desirable co-schedule of tasks that reduces interference is not possible due to the mix of applications in the system.

8. Conclusion

We presented application-to-core mapping policies that largely reduce inter-application interference in network-on-chip based multi-core systems. We have shown that by intelligently mapping applications to cores in a manner that is aware of applications' characteristics, significant increases in system performance, fairness, and energy-efficiency can be achieved at the same time. Our proposed algorithm, A2C, achieves this by using two key principles: 1) mapping network-latency-sensitive applications to separate node clusters in the network from network-bandwidth-intensive applications such that the former makes fast progress without heavy interference from the latter, 2) mapping those applications that benefit more from being closer to the memory controllers close to these resources.

We have extensively evaluated our mechanism and compared it both qualitatively and quantitatively to different application mapping policies. Our main results show that averaged over 128 multiprogrammed workload mixes on a 60-core 8x8-mesh system, our proposed A2C improves system throughput by 16.7%, while also reducing system unfairness by 22.4% and interconnect power consumption by 52.3%. We conclude that the proposed approach can be an effective way of improving overall system throughput, fairness, and power-efficiency and therefore can be a promising way to exploit the non-uniform structure of network-on-chip-based multi-core systems.

Acknowledgments

We thank members of the SAFARI research group and the anonymous reviewers for their feedback. This research was partially supported by grants from NSF (CAREER Award CCF-0953246, CCF-1256203, CCF-1147397 and CCF-1212962), GSRC, SRC, Intel ISTC-CC, and the Intel URO Memory Hierarchy Program. We thank the generous support of our industrial sponsors, including Intel and Oracle. Rachata Ausavarungnirun is supported by the Royal Thai Government Scholarship.

References

[1] "Linux source code (version 2.6.39)," http://www.kernel.org.

[2] D. Abts *et al.*, "Achieving predictable performance through better memory controller placement in many-core CMPs." in *ISCA-36*, 2009.

[3] M. Annavaram *et al.*, "Mitigating Amdahl's Law through EPI throttling," in *ISCA-32*, 2005.

[4] M. Awasthi *et al.*, "Handling the problems and opportunities posed by multiple on-chip memory controllers," in *PACT-19*, 2010.

[5] B. N. Bershad *et al.*, "Avoiding conflict misses dynamically in large direct-mapped caches," in *ASPLOS-6*, 1994.

[6] R. Chandra *et al.*, "Scheduling and page migration for multiprocessor compute servers," in *ASPLOS-6*, 1994.

[7] J. Chang and G. S. Sohi, "Cooperative caching for chip multiprocessors," in *ISCA-33*, 2006.

[8] S. Chen *et al.*, "Scheduling threads for constructive cache sharing on CMPs," in *SPAA-19*, 2007.

[9] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-level page allocation," in *MICRO-39*, 2006.

[10] T. H. Cormen *et al.*, *Introduction to Algorithms*. MIT Press, 2009.

[11] A. K. Coskun *et al.*, "Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors," in *SIGMETRICS-11*, 2009.

[12] R. Das *et al.*, "Application-aware prioritization mechanisms for on-chip networks," in *MICRO-42*, 2009.

[13] R. Das *et al.*, "Aergia: Exploiting packet latency slack in on-chip networks," in *ISCA-37*, 2010.

[14] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, May-June 2008.

[15] B. Falsafi and D. A. Wood, "Reactive NUMA: A design for unifying s-COMA and cc-NUMA," in *ISCA-24*, 1997.

[16] A. Glew, "MLP Yes! ILP No! Memory level parallelism, or, why i no longer worry about IPC," in *ASPLOS WACI Session*, 1998.

[17] M. Gomaa *et al.*, "Heat-and-run: leveraging SMT and CMP to manage power density through the operating system," in *ASPLOS-11*, 2004.

[18] B. Grot *et al.*, "Preemptive virtual clock: A flexible, efficient, and cost-effective QoS scheme for networks-on-a-chip," in *MICRO-42*, 2009.

[19] B. Grot *et al.*, "Kilo-NOC: a heterogeneous network-on-chip architecture for scalability and service guarantees," in *ISCA-38*, 2011.

[20] S. Heo *et al.*, "Reducing power density through activity migration," in *ISLPED*, 2003.

[21] J. Hu and R. Marculescu, "Energy-aware mapping for tile-based NoC architectures under performance constraints," in *ASPDAC*, 2003.

[22] F. C. Jiang and X. Zhang, "CLOCK-Pro: an effective improvement of the CLOCK replacement," in *USENIX*, 2005.

[23] J. A. Joao *et al.*, "Bottleneck identification and scheduling in multithreaded applications," in *ASPLOS-17*, 2012.

[24] R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-indexed caches," *ACM Trans. on Com. Sys.*, 1992.

[25] Y. Kim *et al.*, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *MICRO-43*, 2010.

[26] Y. Kim *et al.*, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA-16*, 2010.

[27] R. Knauerhase *et al.*, "Using OS observations to improve performance in multicore systems," *IEEE Micro*, vol. 28, no. 3, pp. 54 –66, May-June 2008.

[28] J. W. Lee *et al.*, "Globally-synchronized frames for guaranteed quality-of-service in on-chip networks," in *ISCA-35*, 2008.

[29] T. Lei and S. Kumar, "A two-step genetic algorithm for mapping task graphs to a network on chip architecture," in *Euromicro Symposium on DSD*, 2003.

[30] T. Li *et al.*, "Operating system support for overlapping-ISA heterogeneous multi-core architectures," in *HPCA-16*, 2010.

[31] C. K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[32] S. Ma *et al.*, "DBAR: an efficient routing algorithm to support multiple concurrent applications in networks-on-chip," in *ISCA-38*, 2011.

[33] S. Murali and G. D. Micheli, "Bandwidth-constrained mapping of cores onto NoC architectures," in *DATE*, 2004.

[34] S. Muralidhara *et al.*, "Reducing memory interference in multi-core systems via application-aware memory channel partitioning," in *MICRO-44*, 2011.

[35] O. Mutlu *et al.*, "Efficient runahead execution: Power-efficient memory latency tolerance," *IEEE Micro*, 2006.

[36] C. Natarajan *et al.*, "A study of performance impact of memory controller features in multi-processor server environment," in *WMPI-3*, 2004.

[37] H. Patil *et al.*, "Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation," in *MICRO-37*, 2004.

[38] M. K. Qureshi, "Adaptive spill-receive for robust high-performance caching in CMPs," in *HPCA-15*, 2009.

[39] K. K. Rangan *et al.*, "Thread motion: fine-grained power management for multi-core systems," in *ISCA-36*, 2009.

[40] V. Reddi *et al.*, "Voltage smoothing: Characterizing and mitigating voltage noise in production processors via software-guided thread scheduling," in *MICRO-43*, 2010.

[41] M. A. Suleman *et al.*, "Accelerating critical section execution with asymmetric multi-core architectures," in *ASPLOS-14*, 2009.

[42] H.-S. Wang *et al.*, "Orion: A power-performance simulator for interconnection networks," in *MICRO-35*, 2002.

[43] Z. Zhang *et al.*, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *MICRO-33*, 2000.

[44] S. Zhuravlev *et al.*, "Addressing shared resource contention in multicore processors via scheduling," in *ASPLOS-15*, 2010.