Application-to-Core Mapping Policies to Reduce Interference in On-Chip Networks

Reetuparna Das[§] Onur Mutlu[†] Akhilesh Kumar[§] Mani Azimi[§]
[§]Intel Labs [†]Carnegie Mellon University

SAFARI Technical Report No. 2011-001

May 25, 2011

Abstract

As the industry moves toward many-core processors, Network-on-Chips (NoCs) will likely become the communication backbone of future microprocessor designs. The NoC is a critical shared resource *and its effective utilization is essential for improving overall system performance and fairness. In this paper, we propose application-to-core mapping policies to reduce the contention in network-on-chip and memory controller resources and hence improve overall system performance.*

First, we introduce the notion of clusters: cores are grouped into clusters, and a memory controller is assigned to each cluster. The memory controller assigned for a cluster is primarily responsible for servicing the data requested by the applications assigned to that cluster. We propose and evaluate page allocation *and* page replacement *policies that ensure that network traffic of a core is restricted to its cluster with high probability. Second, we develop algorithms that distribute applications between clusters. Our inter-cluster mapping algorithm separates interference-sensitive applications from aggressive ones by mapping them to different clusters to improve system performance, while maintaining a reasonable network load balance among different clusters. Contrary to the conventional wisdom of balancing network/memory load across clusters, we observe that it is also important to ensure that applications that are more sensitive to network latency experience little interference from applications that are network-intensive. Finally, we develop algorithms to map applications to cores within a cluster. The key idea of intra-cluster mapping is to map those applications that benefit more from being close to the memory controller, closer to the controller.*

We 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, the final proposed policy improves system throughput by 16.7% in terms of weighted speedup over a baseline manycore processor, while also reducing system unfairness by 22.4% and interconnect power consumption by 52.3%.

1 Introduction

In a large many-core processor, which processor core is selected to execute an application could have a significant impact on system performance. Performance of an application critically depends on how far away it is scheduled from shared resources such as memory controllers, and how its network packets interfere with other applicationss packets in the interconnect.Hence the core selection policy has to be aware of the spatial geometry of a many-core processor.

Unfortunately, current operating systems are oblivious to the network topology and its state at any instant of time, and employ naive methods while mapping applications to cores. For instance, the Linux 2.x assign a static numbering to cores and choose the numerically smallest numbered core while allocating a core to a application. By exposing the interconnect topology and network state to the OS, we can significantly reduce the destructive interference between concurrent applications and also schedule critical application close to the resources it needs. In this paper, our goal is to propose and evaluate hardware-software cooperative techniques to reduce interference in on-chip networks, thereby improving system performance.

Our proposal is to form intelligent Application-to-Core mapping policies to solve the problem of interference in the NoC. Our mapping policies attempt to answer the question: which core should an application be mapped to in a manycore processor? Our proposal takes a three-step approach. First, *we partition the network into clusters* by data placement techniques that opportunistically map the pages requested by a core to the memory controller within the same cluster as the core. The division of network into clusters ensures that applications that are mapped to different clusters minimally interfere with each other. In addition, clustering improves locality of communication and reduces overall congestion in the network by partitioning the network traffic to smaller sub-networks. To achieve clustering, we propose new page allocation and page replacement policies.

Second, we devise algorithms to *distribute applications between clusters*. In the baseline system, a application's memory accesses are distributed across all memory controllers. However, after clustering, a application's memory accesses are serviced mostly by the memory controller within its cluster. Which applications are mapped together in the same cluster largely affects performance and fairness. We observe that some applications are more *sensitive* to interference than others. We propose techniques to identify such sensitive applications and map them to separate clusters from those of aggressive applications, with the goal of improving performance. While performing inter-cluster mapping, our algorithms also ensure that network load is balanced among clusters to prevent any single memory controller from being a bottleneck.

Finally, we tackle the issue of how to map applications to cores within a cluster. To this end, we develop an intra-cluster mapping policy which maps closest to the memory controller those applications that benefit most from being close to the memory controller. We find that memory-intensive and network-sensitive applications benefit more from being mapped close to the memory controller and develop a new metric to identify such applications. Our algorithm forms a ranking of applications using this metric, and maps applications to cores in concentric circles around the memory controller in rank order, a policy we call *radial mapping*.

To summarize, the main contributions of this paper are the following:

- We observe that how applications are mapped to cores and how the data accessed by them is mapped to memory controllers has significant impact on system performance, fairness, and energy efficiency. We develop systematic policies and algorithms that take into account application characteristics to form efficient application-to-core mappings for many core processors.
- We propose novel application-to-core mapping policies to improve system throughput, system fairness, and network energy. Our key idea consists of three new contributions. First, to cluster cores into sub-networks to reduce interference between applications mapped to different clusters. Second, to distribute applications between clusters such that interference-sensitive applications are separated from others and network load is balanced across clusters. Third, to map memory-intensive and interference-sensitive applications closest to the memory controller within a cluster. We develop insights into and metrics for identifying applications that would benefit from being mapped to a separate cluster from others or close to a memory controller within a cluster.

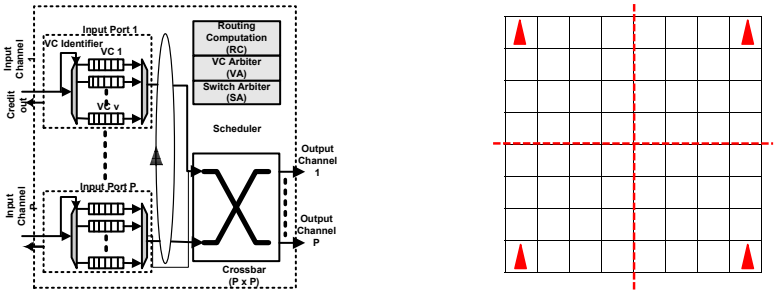


Figure 1: Generic NoC router and topology: (a) Router (b) CMP Layout for a 8x8 Mesh with 4 memory controllers.

- We 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. We compare the proposed mapping scheme to the baseline many-core processor with no clustering and random application-to-core mapping as well as worst case scenarios. Averaged over 128 randomly generated multiprogrammed workloads, the final proposed policy improves system throughput by 16.7% in terms of weighted speedup over baseline manycore processor, while also reducing application-level unfairness by 22.4% and NoC power consumption by 52.3%. We also show that our policies can be used as a complementary technique with application-aware prioritization techniques [11] to further improve performance and fairness.

2 Background

In this section, we provide a brief background on NoC architectures followed by a motivating discussion on the effect of application-to-core mapping policies. For an in-depth introduction to NoCs, we refer the reader to [10].

Router: A generic NoC router architecture is illustrated in Figure 1(a). The router has P input and P output channels/ports; typically $P = 5$ for a 2D mesh, one from each direction and one from the network interface (NI). The Routing Computation unit, RC, is responsible for determining the next router and the virtual channel within the next router for each packet. The Virtual channel Arbitration unit (VA) arbitrates amongst all packets requesting access to the same VCs and decides on winners. The Switch Arbitration unit (SA) arbitrates amongst all VCs requesting access to the crossbar and grants permission to the winning packets/flits. The winners are then able to traverse the crossbar and are placed on the output links.

Network Transactions: In the many-core processor architecture we study in this paper, the NoC connects the core nodes (a CPU core with its private cache's) and the on-chip memory controllers. Figure 1(b) shows the layout of the many core processor with a 8x8 mesh. All tiles have a router. The memory controllers (triangles) are placed in the corner tiles. All other tiles have a CPU core, private L1 and private L2 cache. The core nodes send request packets to on-chip memory controller nodes via the NoC and receive response data packets from the memory controller, once data has been fetched from off-chip DRAMs.

Interference in NoC: Each packet spends at least 2-4 cycles at each router depending on the number of stages in the router pipeline. In addition to the router pipeline stages, a packet can spend *many cycles* waiting in a router, competing for buffers or switch with other packets. Thus, an application's packet may be blocked in the network due to interference with other applications' packets. While its packets are buffered in remote routers, the application (running on the core node) stalls waiting for its packets to return.

3 Motivation

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, a 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 different factors can lead to variation in performance. For example, Figure 2 shows the system performance for 576 different application-to-core mappings for the same workload (detailed system configuration is given in Section 6). The workload consists of 10 copies each of applications `gcc`, `barnes`, `soplex`, `lbm`, `milc` and `leslie` running together. The Y-axis shows the system performance in terms of weighted speedup of the different mappings (higher is better). Each blue dot represents a different mapping. It can be seen that the best possible mapping provides 1.6X higher system performance than the worst possible mapping. **Our goal** is to devise new policies that can find a application-to-core mapping to maximize system performance.

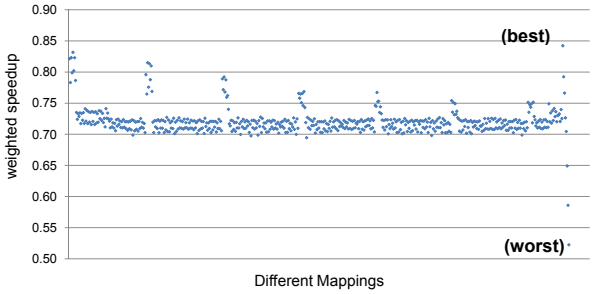


Figure 2: Performance of 576 different application to core mappings for one multiprogrammed workload

4 Application-to-Core Mapping Policies

Our goal is to improve system throughput by efficient Application-to-Core mapping policies that reduce interference in on-chip networks. To enable this, we develop three techniques: 1) clustering, 2) algorithms to distribute applications between clusters, and 3) algorithms to map applications to cores within a cluster. First, cores are clustered into subnetworks to reduce interference between applications mapped to different clusters (Section 4.1). Second, applications are distributed between clusters such that overall network load is balanced between clusters (Section 4.2.1), while separating the interference-sensitive (or, network-sensitive) applications from interference-insensitive ones (Section 4.2.2). Finally, after applications have been distributed between clusters, intra-cluster radial mapping 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 4.3).

4.1 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. *Clustering factor* is defined as the percentile of accesses that can be constrained within the cluster. As a first step, we form clusters to partition network traffic between clusters. Forming clusters ensures that applications mapped to different clusters interfere

minimally with each other. Figure 1 (b) shows a many core processor with cores organized in an 8x8 mesh on-chip network. We partition the tiles of the on-chip network such that each cluster has one memory controller (the *home memory controller* of that cluster). The dotted lines show the division of the on-chip network into four clusters. In addition to reducing interference among applications, clustering also improves communication locality since the applications in the cluster communicate with the memory controller in the cluster. This has two positive effects: 1) it reduces overall congestion in the network, 2) it reduces the average distance packets need to traverse (to get to the memory controller), thereby reducing packet latency and network energy consumed per packet.

How to enforce clustering? 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 [30, 25, 20]. To enable clustering, page allocation and replacement policies should be modified such that data requested by a core is opportunistically mapped to the *home memory controller (home MC)* of the core. The required modifications for the page allocation and replacement policies are relatively modest, as we describe in detail below.

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 pages exist, *preference* is given to a page belonging to the *home MC*, while finding the replacement page candidate. To achieve this, we modify the commonly-used CLOCK [18] page replacement policy. The CLOCK policy keeps a circular list of pages in memory, with the "hand" (iterator) pointing to the oldest allocated page in the list. There is a referenced ("R") bit associated with each page; when a page is referenced, the R bit is set to 1. R bits are cleared periodically by the operating system. When a page fault occurs and no free pages exist, then the R bit is inspected at the hand's location. If R is 0, the given page is used as a replacement candidate. If R is 1, the CLOCK hand is incremented and the process is repeated until a replacement candidate is found. The goal is to replace the first page that is not referenced (i.e., R bit set to 0). To achieve clustering, the CLOCK algorithm looks ahead N pages beyond the first replacement candidate in the list, so that it can potentially find a page that belongs to the *home MC* with R bit set to 0. If the algorithm does not find a page belonging to the *home MC* with R bit set to 0, within N pages beyond the default replacement candidate, the default replacement candidate (which belongs to another MC) is selected.[1] We call this modified CLOCK algorithm the *cluster-CLOCK* algorithm.

The above modifications ensure that the new page replacement policy does not perturb significantly the existing replacement order, and at the same time achieves the effect of clustering opportunistically. 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.

4.2 Mapping Policy between Clusters

In this subsection, we devise clustering algorithms that decide *to which cluster* a application should be mapped to. Figure 3 shows several different possible inter-cluster mappings. Each core tile is shaded according to network intensity of the application running on it; a darker tile corresponds to an application

[1] 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.

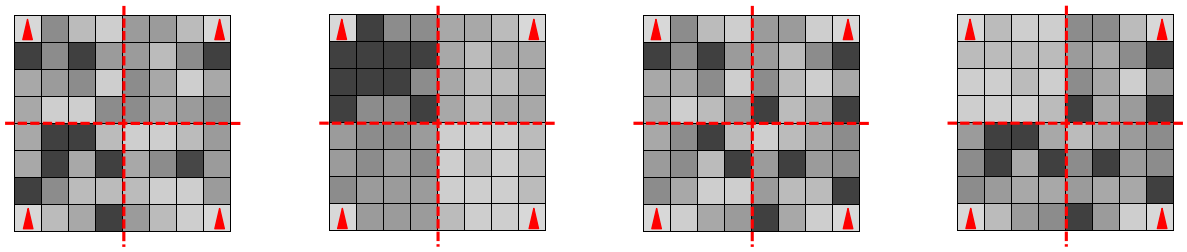


Figure 3: Inter-Cluster Application-to-Core Mapping Examples: (a) Random (RND) (b) Imbalanced (IMBL) (c) Balanced (BL) (d) Balanced with Reduce Interference (BLRI)

with a higher private cache miss rate, i.e. Misses per Thousand Instructions (MPKI), running on the core in the tile. Figure 3 (a) shows a possible random mapping of applications to clusters (called RND), which is the baseline mapping policy in existing general-purpose systems. Figure 3 (b) shows a mapping where applications are completely imbalanced between clusters (called IMBL). The upper left cluster is heavily loaded while the lower right cluster has very low load. We achieve an imbalanced mapping by ranking applications based on their MPKIs and mapping consecutively ranked applications to the same cluster until the cluster is full. 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. We investigate this policy only for comparison purposes.

We develop two inter-cluster mapping policies. The goal of the *Balanced (BL)* policy, an example of which is shown in Figure 3 (c), is to balance load between clusters. The goal of the *Balanced with Reduced Interference (BLRI)* policy, and example of which is shown in Figure 3 (d), 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 keep balance in the network as much as possible.

4.2.1 Balanced Mapping (BL): Mapping to Balance Load between Clusters

The first goal of our inter-cluster application to core mapping policy is to balance the load among clusters such that there is better overall utilization of network and memory bandwidth. To form a application-to-core mapping singularly optimized to balance load, we start with the list of applications *sorted with respect to their network intensity*: $\{A_0, A_1, \ldots, A_{n-1}\}$ where A_i is i^{th} highest intensity application and n is number of cores in the chip. Network intensity is measured in terms of injection rate into the network, which can be quantified by private cache Misses per Thousand Instructions (MPKI), periodically at run-time or provided statically before the application starts. The basic idea is to map consecutively ranked applications to consecutive clusters in a round robin fashion. Hence, applications are mapped to clusters as follows: $C_i = \{A_i, A_{i+k}, A_{i+2k}, A_{i+3k}, \ldots\}$ where C_i is the set of applications mapped to the i^{th} cluster, and k is the number of cores in each cluster. Figure 3 (c) shows an example mapping after applying the inter-cluster BL mapping algorithm to a workload mix.

4.2.2 Reduced Interference Mapping (RI): Mapping to Reduce Interference for Sensitive Applications

We observe that some applications are more sensitive to interference in the network compared to other applications. In other words, the relative slowdown these applications experience due to interference in the shared NoC is higher than that other applications experience. Our insight is that if we can separate such

Algorithm 1: Balanced Mapping (BL) Algorithm to Map Applications to Clusters

Input: $Applications = \{A_0, A_1, \ldots, A_{n-1}\}$ such that
$\forall\, i\ \ NetworkDemand(A_i) \geq NetworkDemand(A_{i+1})$
p is the number of clusters

```
1:  begin
2:      for j ← 0 to p − 1 do
3:          C_j ⟵ ∅            // C_j is the set of applications mapped to j^th cluster
4:      end
5:      j ⟵ 0
6:      for i ← 0 to n − 1 do
7:          C_j ⟵ C_j ∪ A_i
8:          j ⟵ j ⊕_p 1          // modulo p increment of j for round robin cluster
            assignment
9:      end
10: end
```

sensitive applications from other network-intensive ones by mapping them into their own cluster, we can improve performance significantly because the slowdowns of the most sensitive applications can be kept under control.[2] To achieve this, we first present an algorithm to identify sensitive applications in a given workload and then propose an algorithm to map them into a separate cluster, if enough sensitive applications exist in a workload.

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): such applications are in general more sensitive to interference because any delay for the application's network packet likely results in 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 Cycles Per Miss (STPM) metric correlates with memory level parallelism (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 private cache misses per thousand instructions (MPKI) is greater than a threshold. Algorithm 2 formally summarizes how our technique categorizes applications as sensitive.

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 at least R% of the cores in a cluster. This condition ensures that there are enough sensitive applications such that their separation from others actually reduces interference significantly. We empirically found $R = 75$ is a good threshold.

[2] In a similar spirit, researchers proposed prioritizing latency-sensitive applications over others for memory scheduling [24, 20] or packet scheduling [11] to improve system throughput. Our goal is to reduce interference by cluster assignment rather than scheduling.

Algorithm 2: Algorithm to identify sensitive applications

Input: $Applications = \{A_0, A_1, \ldots, A_{n-1}\}$ such that
$\forall\, i\ \ STPM(A_i) \geq STPM(A_{i+1})$
$Thresh_{MPILow}, Thresh_{MPIHigh}, Thresh_{SensitivityRatio}$

```
1:  begin
2:      SensitiveApplications ⟵ ∅
        // The STPM of all applications (A_0 to A_{n-1}) is normalized to MinSTPM
3:      MinSTPM ⟵ minimumofSTPM(Applications)
4:      for i ← 0 to n − 1 do
5:          if NetworkDemand(A_i) > Thresh_MPILow and NetworkDemand(A_i) < Thresh_MPIHigh and
6:          STPM(A_i)/MinSTPM ≥ Thresh_SensitivityRatio then
7:              SensitiveApplications ⟵ SensitiveApplications ∪ A_i
8:          end
9:      end
10: end
```

Second, the entire workload should exert a large amount of pressure on the network. We found that allocating an $RI_{cluster}$ 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. *Therefore, we consider forming a separate cluster only for very intensive workloads that are likely to saturate the network*. As a result, our algorithm considers allocating an $RI_{cluster}$ if *the aggregate aggregate bandwidth demand* of the entire workload is higher than 1500 MPKI. [3]

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 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 Algorithm 3 (Reduced Interference) illustrates how to form a separate cluster ($RI_{cluster}$) for sensitive applications. 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 maximizing aggregate sensitivity of $RI_{cluster}$ can be easily shown to be equivalent to the *0-1 knapsack problem* [8]. We use a simple dynamic programming solution [8] to the knapsack problem to choose p sensitive applications. In case there are fewer sensitive applications than p, we pack the empty

[3] Each memory controller feeds the cores with two outgoing NoC channels (each channel @32 GB/s, total 64GB/s). The off-chip memory channels are matched to the network channel capacity (4 channels @16 GB/s, total 64GB/s). The total bandwidth demand of 64 GB/s translates to an MPKI of 500 MPKI (assuming 64 byte cache lines, throughput demand of one memory access per cycle and core frequency of 2 GHz). We consider forming an RI cluster if the bandwidth demand is R=0.75 times the cluster capacity of 64 GB/s (or 375 MPKI). With 4 clusters, the total demand of the workload should be 4*375=1500 MPKI. We empirically validated this threshold.

[4] 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 ($Thresh_{MPIRI}$ = 250 MPKI).

Algorithm 3: Reduced-Interference (RI) Mapping: Algorithm to form the $RI_{cluster}$ for Sensitive Applications

Input: $SensitiveApplications = \{S_0, S_1, \ldots, S_{k-1}\}$
$FillerApplications = \{F_0, F_1, \ldots, F_{l-1}\}$ such that
$\forall\, i\ NetworkDemand(F_i) \leq NetworkDemand(F_{i+1})$ **and** $F_i \notin SensitiveApplications$
$Thresh_{MPKI-RI}(= 250MPKI), p = number\ of\ cores\ in\ a\ cluster$

```
begin
    RI_cluster ⟵ ∅ MaxWeight ⟵ Thresh_MPKI−RI
    for i ← 0 to k − 1 do
        Weight_i ⟵ MPKI(S_i)
        Value_i ⟵ STPM(S_i)/MinSTPM
    end
    RI_cluster ⟵ KnapSack(Weight, Value, k, MaxWeight)
    for i ← 0 to p − |RI_cluster| − 1 do
        RI_cluster ⟵ RI_cluster ∪ F_i
    end
end
```

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 identified by Algorithm 2 is more than $2p$, then we try to form two $RI_{cluster}$. We found this happened negligibly in our experiments with 128 workloads.

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.** Figure 3 (d) shows an example mapping after applying the inter-cluster BLRI mapping algorithm for a workload mix.

4.3 Mapping Policy within a Cluster

After mapping each application to a cluster, a question remains: *which core within a cluster* should a application be mapped to? Figure 4 shows different possible intra-cluster mappings for a single cluster. Figure 4 (a) depicts a random intra-cluster mapping, but this is not the best intra-cluster mapping as it is agnostic to application characteristics. We observe that 1) a network-intensive application benefits more from being placed closer to memory controller than other applications (See Figure 5) and 2) an interference-sensitive application benefits more for being placed closer to memory controller (especially if its network intensive) than other applications. Figure 5 illustrates the former by showing that applications with higher MPKI (shown in graphs to the left) gain more in terms of IPC performance when mapped closer to the memory controller than applications with lower MPKI (shown in graphs to the right).

Our intra-cluster mapping exploits these insights. It differentiates applications based on *both* their 1) *network demand* (i.e. rate of injection of packets) measured as MPKI, and 2) *sensitivity* to network latency measured as *Stall Time per Miss (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*

[5]Note that the dynamic programming solution is feasible since our algorithm is invoked at long time intervals at the granularity of OS time quantums.

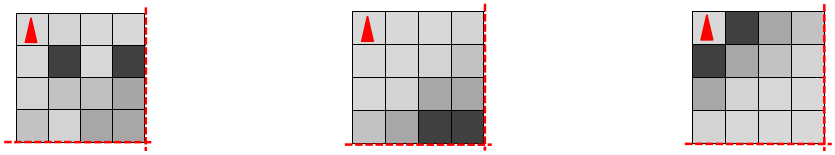


Figure 4: Intra-Cluster Application-to-Core Mapping Examples: (a) Random (b) Inverse Radial (c) Radial

in sorted order, starting from the application with the highest $STPKI$, Algorithm 4 shows this process. Figure 4 (c) shows an example resulting mapping within a cluster. Darker (inner and closer) tiles represent heavy and sensitive applications and lighter (outer and farther) tiles represent lower intensity applications with low sensitivity.

Algorithm 4: Intra-Cluster Mapping: Radial Mapping Algorithm

Input: $Applications = \{A_0, A_1, \ldots, A_{c-1}\}$ such that
$\forall\, i\ \ MPKI(A_i) * STPM(A_i) \geq MPKI(A_{i+1}) * STPM(A_{i+1})$
$Cores = \{N_0, N_1, \ldots, N_{c-1}\}$ such that
$\forall\, i\ \ Distance(N_i, HomeMemoryController) \leq Distance(N_{i+1}, HomeMemoryController)$

1: **for** $i \leftarrow 0$ **to** $c - 1$ **do**
2: $\quad ApplicationMap(N_i) \longleftarrow A_i$
3: **end**

4.4 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 (Section 4.1). Second, the BLRI algorithm is invoked to map applications to clusters (Section 4.2.2). In other words, the algorithm attempts to allocate a separate cluster to interference-sensitive applications (Section 4.2.2), if possible, and distributes the applications to remaining clusters to balance load (Section 4.2.1). Third, after applications are assigned to clusters, the applications are mapped to cores within the clusters by invoking the intra-cluster radial mapping algorithm (Section 4.3). **We call this final algorithm, which combines clustering, BLRI and radial algorithms, as the A2C mapping.**

5 Enforcement of Application-to-Core Mapping

5.1 Profiling

The proposed mapping policies assume knowledge of two metrics: a) network demand in terms of MPKI and b) sensitivity in terms of stall cycles per miss (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 additional 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

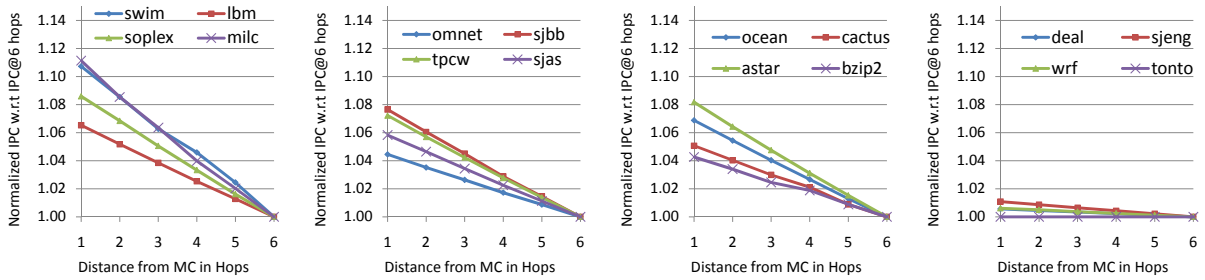


Figure 5: Application performance with distance from memory controller (MC) when running *alone*. Each figure has four benchmarks each with (a) MPKI range from 25.4 to 29.6 (b) MPKI range from 8.5 to 10.6 (c) MPKI range from 3.2 to 7.2 (d) MPKI range from 0.26 to 0.40. The figures show 1) speedups of higher MPKI applications are higher with closer placement to MC and 2) some applications with similar or lower MPKI show high speedups because they are more sensitive to network latency.

retired because it is waiting for an L2 miss. Note that the A2C technique requires only a relative ordering among applications and hence quantizing applications to classes based on the above metrics is sufficient.

5.2 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 4.1. 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. There is already thrust towards firmware support for multi-core processors to manage power and thermal issues, support application migration, and provide fault tolerance. If integrated into the firmware, our techniques can be either exposed to the operating system or completely managed by the hardware.

5.3 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, the ability to determine *online the application characteristics* (Section 5.1). Second, since application-to-core mapping of a application can change between different execution phases, we migrate the applications between cores to enforce new mappings. We discuss the costs of application migration in the next subsection.

The application-to-core mappings evolve naturally and synergistically over time such that the changes between an old and new mapping are minimal. Consider the scenario when their are no applications running, the operating system schedules applications one-by-one. The mapping produced will reach a stable state over time, requiring minimum application migrations between old and new mappings.

5.4 Migration Costs

A new application-to-core mapping may require migration of a 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 both the cores are managed by a single unified operating system. Thus, unlike process migration in MP systems where a process is migrated between processors managed by different operating systems, in this case, minimal system state needs to be accounted for. For example, the file handling, memory management, IO, network socket state, etc are shared between the cores due to the single operating system image and need not be saved or restored; 2) A constant cost of 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 entire execution phase this cost is negligibly small across all benchmarks (see Section 7.6); and 4) A variable cost due to potential reduction in *clustering factor*. This cost is only incurred if we migrate applications between clusters and, after migration, the application continues to access pages mapped to its old cluster. We quantify this cost as well in our performance evaluation for all our workloads (see Section 7.6). Our evaluations faithfully account for all of these four types of migration costs.

6 Methodology

6.1 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 Pin dynamic binary instrumentation tool [23], 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 network connects the core tiles and memory controller tiles. The system we model is self-throttling as real CMP 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 for estimating the router power [29].

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 [18]. 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, $Thresh_{SensitivyRatio}$ = 5 and $Thresh_{MPKI-RI}$ = 250 MPKI. These parameters are empirically determined. The constant cost for OS book keeping while migrating applications is assumed to be 50,000 cycles. The migrating applications write and read 128 bytes to/from the memory to save and restore their register contexts.

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, write-back, 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
Main Memory	4GB DRAM,up to 16 outstanding requests per-core, 160 cycle access, 4 DDR Channels at 16GB/s 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

6.2 Evaluation Metrics

Our evaluation uses several metrics. We measure **system performance** in terms of average **weighted speedup** [13], a commonly used multi-program performance metric, which is the average of the sum of slowdowns of each application compared to when it is run alone on the same system. We measure **system fairness** in terms of the maximum slowdown observed in the system.

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

IPC_{alone} is the IPC of the application when run alone on our baseline system. We also report **IPC throughput**.

$$IPC\ Throughput = \sum IPC_i \quad (2)$$

6.3 Workloads and Simulation Methodology

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 [26] for all our workloads except commercial traces, which were collected over Intel servers. Figure 11 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. Each workload 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.

7 Performance Evaluation

7.1 Overall Results across 128 Multiprogrammed Workloads: A2C Mapping 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 repre-

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 [26] 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 [26]. 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 5.1). 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

sentative of existing systems, 2) our enhanced system which uses our clustering and modified CLOCK algorithm (described in Section 4.1) but still uses random mapping of applications to cores (CLUSTER+RND), 3) our final system which uses our combined A2C algorithm (summarized in Section 4.4), which consists of clustering, inter-cluster BLRI mapping, and intra-cluster radial mapping algorithms (A2C).

Figure 6 (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). Our final A2C algorithm improves IPC throughput by 13.5% (not in figure) and weighted speedup by 16.7% over the baseline (BASE), while reducing unfairness by 22%.[6]

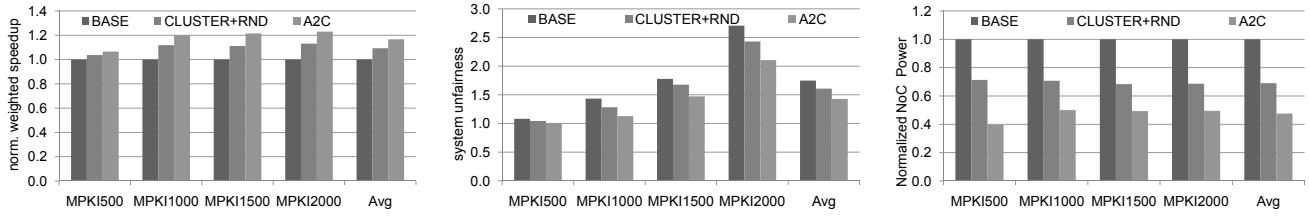


Figure 6: (a) System performance (b) system unfairness and (c) interconnect power of the A2C algorithm for 128 workloads

Interconnect Power Figure 6 (c) shows the 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). 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 get mapped close to the memory controllers and network load is balanced across controllers after clustering.

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

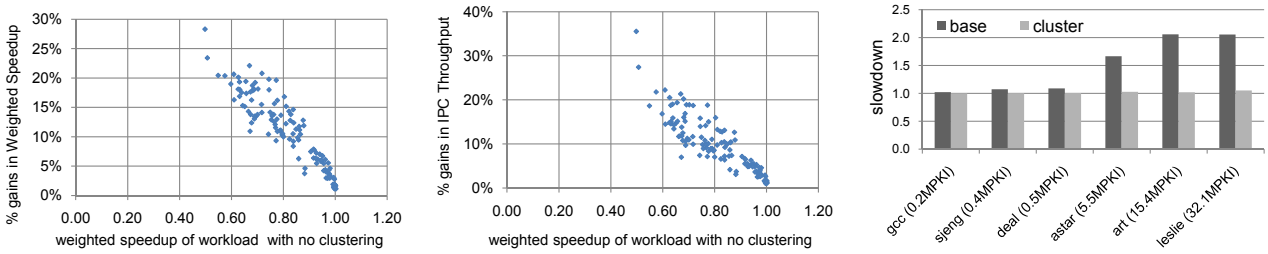
7.2 Effect of Clustering

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 7 (a) and (b) plot the gains in weighted and relative speedup due to clustering for each workload against the baseline weighted speedup of the workload. A lower baseline weighted speedup indicates that

[6]We do not show graphs for instruction throughput due to space limitations. Clustering alone (CLUSTER+RND) improves IPC throughput by 8.0% over the baseline.

average slowdown of applications are higher and hence contention/interference is high between applications in the baseline. The figures show that performance gains due to clustering are significantly higher for workloads with lower weighted speedup (i.e., higher interference). This is intuitive because the main goal of clustering is to reduce interference between applications. We conclude that the benefits of clustering are higher when interference is higher in the network.



(a) (b) (c)

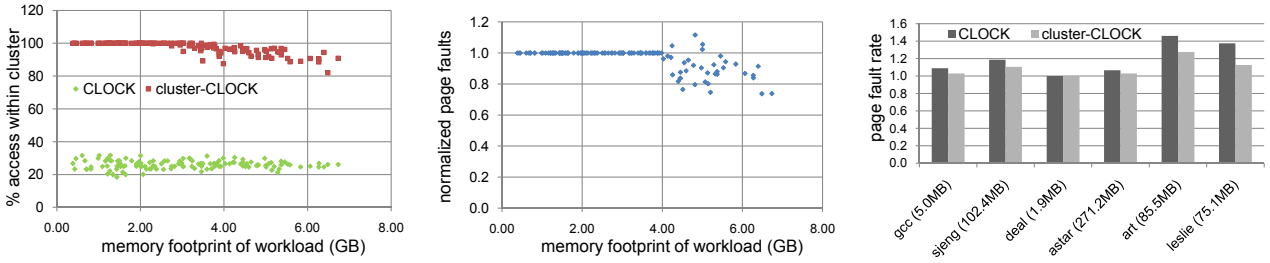
Figure 7: Performance gain of clustering across 128 workloads (a) Weighted speedup (b) IPC throughput (c) A Case Study

To understand the details, we zoom in on one workload as a case study in Figure 7 (c). This workload consists of 10 copies each of applications `gcc`, `sjeng`, `deal`, `astar`, `art`, and `leslie` running together on the many-core processor. The Y-axis measures the slowdowns (lower is better) of individual applications compared to when run alone. We observe that clustering 1) reduces slowdown of all applications because it reduces interference and reduces their latency of access to the memory controller, 2) provides larger benefits to more network-intensive (higher MPKI) applications because they have higher slowdown in the baseline system.

Analysis of the cluster-CLOCK page replacement algorithm: To enforce clustering, we have developed the cluster-CLOCK algorithm 4.1 which modifies the default page allocation and page replacement policies. The results in Figure 8 quantify the effectiveness of cluster-CLOCK across 128 workloads. Figure 8 (a) 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 8 (b) 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. This is a side effect of cluster-CLOCK since the algorithm is not designed to reduce page faults. Yet, it 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.

To illustrate this behavior, we focus on one workload as a case study in Figure 8 (c), which depicts the page fault rate in terms of page faults incurred per unique page accessed by each application with CLOCK



(a) (b) (c)

Figure 8: Page access behavior across 128 multiprogrammed workloads (a) Clustering factor (b) Page Faults (c) A Case Study

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 footprint 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. Hence, cluster-CLOCK can reduce page fault rates by likely localizing page replacement and thereby limiting as much as possible page-level interference among applications to pages assigned to a single memory controller.

7.3 Effect of Inter-Cluster Mapping

We study the effect of inter-cluster load balancing (BL) and reduced interference (BLRI) mapping algorithms.

BL Mapping Figure 9 (a) shows application slowdowns in an example workload consisting of 10 copies each of `h264ref`, `gromacs`, `barnes`, `libquantum`, `milc` and `leslie` applications running together. A completely imbalanced (IMBL) inter-cluster mapping (described in Section 4.2) severely slows down the network-intensive and network-sensitive applications. A random (RND) mapping (which is our baseline) still slows down the same applications, albeit less, by providing better balance in interference. Our balanced (BL) mapping algorithm, which distributes load equally among all cluster provides the best speedup (19.7% over IMBL and 5.9% over RND) by reducing the slowdown of all applications.

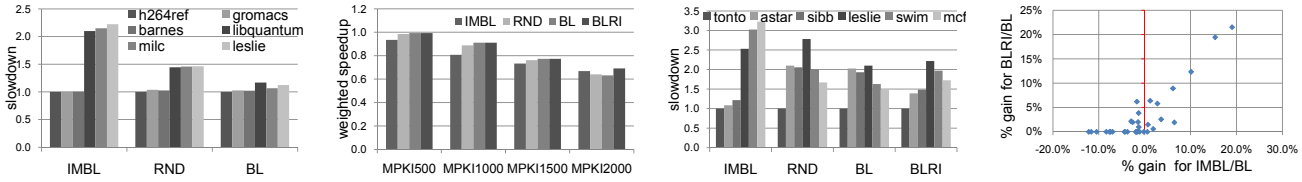


Figure 9: (a) Case study analyzing BL mapping (b) Average results for different inter-cluster mappings across 128 workloads (c) Case study analyzing BLRI mapping (d) Average results for BLRI mapping for the MPI2000 workload category

Figure 9 (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), balanced mapping (BL) provides significantly higher performance than IMBL and RND, as expected, 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-intensive (low MPKI) applications from network-non-intensive (high MPKI) ones by placing them into separate 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. We conclude that a completely balanced mapping is not desirable when network load is very high. This observation motivated our BLRI algorithm, which we analyze next.

BLRI Mapping We first analyze the impact of balanced and reduce interference (BLRI) mapping algorithm with a case study shown in Figure 9 (c) for a workload consisting of 10 copies each of applications `tonto`, `astar`, `sjbb`, `leslie`, `swim` and `mcf` running together. We make the following observations:

• Baseline random mapping (RND) slows down all applications except the very network-insensitive ones. Both sensitive applications (`astar` (5.7 MPKI) and `sjbb` (12.3 MPKI)) and network-intensive applications (`swim` (30.8 MPKI) and `mcf` (75.5 MPKI)) suffer large slowdowns.

• Our balanced mapping (BL) technique improves over RND mapping (6.5% speedup over RND), but it does not reduce the slowdown of network-sensitive applications (`astar`, `sjbb`). The BL algorithm reduces slowdown of all applications equally.

• Imbalanced mapping (IMBL) improves slowdowns of sensitive applications by separating them from network-intensive ones. However, this severely slows down intensive applications because they compete for bandwidth in the same controllers. Although the overall performance of IMBL is 1.3% higher than BL in this workload, IMBL largely increases unfairness (Maximum Slowdown) by 1.53X.

• RI part of BLRI mapping correctly identifies `astar`, `sjbb` as sensitive applications and maps them to their own cluster. On the other hand, the BL part of BLRI effectively balances the load among the other clusters. As a result, BLRI mapping improves performance 6.4% over BL, 13.2% over RND, and 5.1% over IMBL.

Figure 9 (d) 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 MPI 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 in 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 and performing load balancing for the remaining ones.*

7.4 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. Figure 10 (a) shows the slowdowns in a workload consisting of 10 copies each of `gcc` (MPKI 0.2), `calculix` (MPKI 0.14), `cactus` (MPKI 2.9), `tpcw` (MPKI 11.6), `lbm`(MPKI 27.4) and `mcf` (MPKI 75.5) applications running together. We examine three different intra-cluster mapping algorithms: 1) Radial: our proposed radial mapping described in Section 4.2, 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.

The applications are ranked as follows in terms of decreasing STPKI order by our radial mapping policy: `mcf`, `tpcw`, `lbm`, `cactus`, `calculix` and `gcc`. In Figure 10, is clear that all applications placed closer to memory controller benefit strongly, while applications `gcc` and `calculix` are not affected by farther placement from the memory controller since they are not sensitive. As a result, radial mapping provides speedup of 13.2% over RadialINV and 11.0% over RND for this workload.

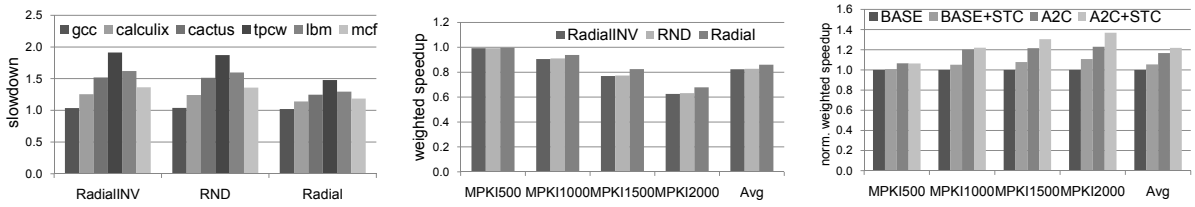


Figure 10: (a) A case study analyzing radial mapping (b) Average results for radial mapping across 128 workloads (c) Performance comparison of A2C mapping and application-aware prioritization (STC) for 128 workloads

Figure 10 (b) 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. RadialINV mapping is the worst mapping for all workloads. We conclude that our metric and algorithm for identifying and deciding which workloads to map close to the memory controller is effective.

7.5 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. [11] 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 a non-intensive applications over packets of non-intensive applications. We compare Application-to-Core mapping policies to application-aware prioritization policies (STC) because both techniques have similar goals: to reduce interference in NoC. *However, both techniques take different approaches towards the same goal*. STC tries to reduce interference in the NoC by efficient packet scheduling in routers while A2C tries to reduce interference by mapping applications to separate clusters and controllers.

Figure 10 (c) shows that STC[7] 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 is because it prioritizes non-intensive applications (shorter jobs) within a cluster in a coordinated manner, ensuring all the routers act in tandem. In contrast, the baseline round-robin arbitration policy is uncoordinated: it causes one application to be prioritized in one router only to be delayed in the next, resulting in a slowdown for all applications. This coordinated packet scheduling effect of STC is orthogonal to the benefits of A2C. Hence, we conclude that

[7]The default parameters used for STC [11] 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.

our mapping mechanisms interact synergistically with application-aware packet scheduling.

7.6 Effect of Dynamic A2C Mapping

Our evaluation assumed so far static mapping is formed with pre-runtime knowledge of application characteristics. We now evaluate the dynamic A2C scheme, described in Section 5.1. 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). A application can migrate at the end of the dynamic profiling interval after each phase. In section 5 we qualitatively discussed the overheads of application migration. In this section, we first quantify the increases in cache misses when a application migrates from one core to another. We then quantify the reduction in clustering factor due to migrations.

Figure 11 analyzes the MPKI of the 35 applications during the profiling phase (MPKI-10M) and the enforcement phase (MPKI-300M). It also analyzes the the increase in the MPKI due to migration to another core right 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). Benchmarks on the X axis are sorted from lightest (lowest baseline MPKI) to heaviest (highest baseline MPKI) from left to right. We make several key observations:

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

2. 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.

3. 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 a application to another core is minimal over the enforcement phase of the new mapping.

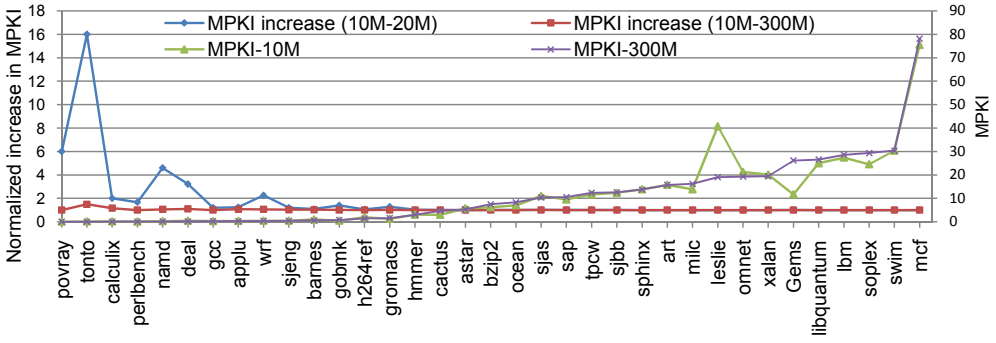


Figure 11: Increase in MPKI due to migration for individual applications

The clustering factor (i.e., the ratio of memory accesses that are serviced by the home memory controller) is also affected by application migrations. The clustering factor may potentially decrease, if, after migration, a application continues to access the pages mapped to its old cluster. Our dynamic algorithm minimizes the number of inter-cluster application migrations by placing applications with similar MPKI in equivalence classes for A2C mapping computation (we omit the detailed explanation of this optimization due to space limitations).

Figure 12 (a) 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 2C to 89.0% with dynamic A2C due to inter-cluster application migrations. However, dynamic A2C mapping still provides a very large improvement in clustering factor compared to the baseline mapping.

Figure 12 (b) compares the performance improvement achieved by static and dynamic A2C schemes for eight workloads, which consist of two workloads from each MPKI category, over a baseline that employs clustering but uses random application-to-core mapping.[8] The performance 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 MPI2000 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.

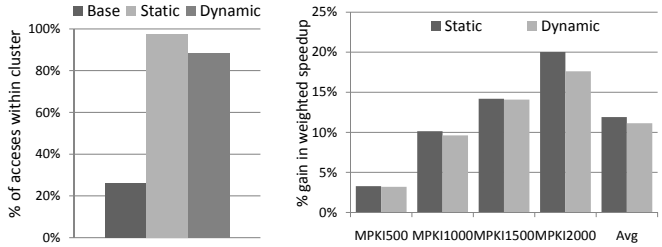


Figure 12: (a) Clustering factor for 128 workloads (b) performance of static and dynamic A2C for 8 workloads

8 Related Work

We briefly describe the most closely related previous work in this section.

Thread Migration and Task Scheduling: Thread migration has been explored to manage power [27, 17], thermal hotspots [15, 9] or to exploit heterogeneity in multi-core processors [22]. We use thread migration to enable dynamic application-to-core mappings to reduce interference in NoCs. Prior works have proposed task/thread scheduling to improve locality of communication by placing frequently communicating threads closer to each other [6, 7] in parallel applications and for shared cache management [5]. Our techniques are different from the above because our goals are to 1) reduce interference in the network between different applications and 2) reduce contention at the memory controllers.

Memory Controllers and Page Allocation: Awasthi et al. [2] have 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. Our work is different and complementary in three major aspects. First, our goal is to reduce interference between applications in the interconnect and the memory controllers. Second, our mechanism solves the problem of mapping applications to the cores whereas Awasthi et al. do not tackle the issue of how applications are mapped to cores. Third, our results show that balancing load is not desirable to reduce interference in all workloads; some-

[8] We show only eight workloads because we were severely limited by simulation time for evaluation and analysis of dynamic scenarios. The performance simulations for dynamic mapping took us fifteen to twenty days for different workloads because the minimum length of cycle-level simulation had to be a whole phase of 300 million instructions per application (totally 18 billion instructions).

times it is better to create imbalance to isolate interference-sensitive applications from others. We believe the page migration techniques proposed in [2] can be employed to reduce the costs of migration in our dynamic application-to-core mapping policies.

Page coloring techniques have been employed in caches to reduce cache conflict misses [19, 3]. Page allocation and migration has been explored to improving locality in NUMA MPs [4, 28, 14]. These works do not aim to reduce interference between applications, and hence do not solve the problem we are aiming to solve. In addition, these works do not consider application and page mapping in a multi-core NoC system.

Abts et al. [1] recently explore placement of memory controllers in a multi-core processor to minimize channel load. This is orthogonal to our mechanisms.

Reducing Interference in NoCs: Recent works [21, 16, 11, 12] propose prioritization and packet scheduling policies to provide quality of service or improve application-level throughput in the NoC. These works are orthogonal to our mechanism since they perform packet scheduling. We have already shown that our proposal works synergistically with application-aware prioritization [11].

9 Conclusion

In the many-core era, the network-on-chip is a critical shared resource. Reducing interference of applications in the NoC is an important problem for enhancing both system performance and fairness. To this end, we propose application-to-core mapping algorithms to improve the performance, fairness, and energy efficiency of NoC-based many-core systems. Our paper proposes a new approach to reduce interference in the NoC and memory controllers by 1) clustering the cores into subnetworks with at least one memory controller, 2) balancing the load across the clusters while separating interference-sensitive applications, and 3) mapping the applications that will benefit most from high bandwidth and low latency to memory closest to the memory controllers within a cluster. Averaged over 128 randomly-generated multiprogrammed workload mixes on a 60-core 8x8-mesh CMP, our proposed application-to-core mapping policy (A2C) improves overall system throughput by 16.7%, while also reducing system unfairness by 22.4% and interconnect power consumption by 52.3%. Our results also show that A2C outperforms state-of-the-art application-aware prioritization techniques in the NoC and can be easily combined with them. We conclude that the proposed approach is 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 NoC-based multi-core systems.

References

[1] D. Abts, N. D. E. Jerger, J. Kim, D. Gibson, and M. H. Lipasti, "Achieving predictable performance through better memory controller placement in many-core cmps." in *ISCA-36*, 2009.

[2] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the problems and opportunities posed by multiple on-chip memory controllers," in *PACT-19*, 2010.

[3] B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen, "Avoiding conflict misses dynamically in large direct-mapped caches," in *ASPLOS-VI*, 1994.

[4] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum, "Scheduling and page migration for multiprocessor compute servers," in *ASPLOS-VI*, 1994.

[5] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson, "Scheduling threads for constructive cache sharing on cmps," in *SPAA-19*, 2007.

[6] C.-L. Chou and R. Marculescu, "Contention-aware application mapping for network-on-chip communication architectures," in *ICCD*, 2008.

[7] ——, "User-aware dynamic task allocation in networks-on-chip," in *DATE*, 2008.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2009.

[9] A. K. Coskun, R. Strong, D. M. Tullsen, and T. Simunic Rosing, "Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors," in *SIGMETRICS-11*, 2009.

[10] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2003.

[11] R. Das, O. Mutlu, T. Moscibroda, and C. Das, "Application-Aware Prioritization Mechanisms for On-Chip Networks," in *MICRO-42*, 2009.

[12] ——, "Aergia: Exploiting Packet Latency Slack in On-Chip Networks," in *ISCA-37*, 2010.

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

[14] B. Falsafi and D. A. Wood, "Reactive numa: a design for unifying s-coma and cc-numa," in *ISCA-24*, 1997.

[15] M. Gomaa, M. D. Powell, and T. N. Vijaykumar, "Heat-and-run: leveraging smt and cmp to manage power density through the operating system," in *ASPLOS-XI*, 2004.

[16] B. Grot, S. W. Keckler, and O. Mutlu, "Preemptive Virtual Clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-a-Chip," in *MICRO-42*, 2009.

[17] S. Heo, K. Barr, and K. Asanović, "Reducing power density through activity migration," in *ISLPED*, 2003.

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

[19] R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-indexed caches," *ACM Transactions on Computer Systems*, 1992.

[20] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA-16*, 2010.

[21] J. W. Lee, M. C. Ng, and K. Asanovic, "Globally-Synchronized Frames for Guaranteed Quality-of-Service in On-Chip Networks," in *ISCA-35*, 2008.

[22] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn, "Operating system support for overlapping-isa heterogeneous multi-core architectures," in *HPCA-16*, 2010.

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

[24] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," in *ISCA-35*, 2008.

[25] C. Natarajan, B. Christenson, and F. Briggs, " A study of performance impact of memory controller features in multi-processor server environment," in *WMPI-3*, 2004.

[26] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel itanium programs with dynamic instrumentation," in *MICRO-37*, 2004.

[27] K. K. Rangan, G.-Y. Wei, and D. Brooks, "Thread motion: fine-grained power management for multi-core systems," in *ISCA-36*, 2009.

[28] B. Verghese, S. Devine, A. Gupta, M. Rosenblum, B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Os support for improving data locality on cc-numa compute servers," 1996.

[29] H.-S. W. Xinping, H. sheng Wang, X. Zhu, L. shiuan Peh, and S. Malik, "Orion: A power-performance simulator for interconnection networks," in *MICRO-35*, 2002.

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