Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 205852, 22 pages
doi:10.1155/2010/205852

Research Article

A Workload-Adaptive and Reconfigurable Bus Architecture for Multicore Processors

Shoaib Akram, Alexandros Papakonstantinou, Rakesh Kumar, and Deming Chen

Department of Electrical and Computer Engineering, University of Illinois at Urbana Champaign, 1308 West Main Street Urbana, IL 61801, USA

Correspondence should be addressed to Deming Chen, dchen@uiuc.edu

Received 3 October 2009; Revised 8 March 2010; Accepted 24 May 2010

Academic Editor: Marco Platzner

Interconnection networks for multicore processors are traditionally designed to serve a diversity of workloads. However, different workloads or even different execution phases of the same workload may benefit from different interconnect configurations. In this paper, we first motivate the need for workload-adaptive interconnection networks. Subsequently, we describe an interconnection network framework based on reconfigurable switches for use in medium-scale (up to 32 cores) shared memory multicore processors. Our cost-effective reconfigurable interconnection network is implemented on a traditional shared bus interconnect with snoopy-based coherence, and it enables improved multicore performance. The proposed interconnect architecture distributes the cores of the processor into clusters with reconfigurable logic between clusters to support workload-adaptive policies for inter-cluster communication. Our interconnection scheme is complemented by interconnect-aware scheduling and additional interconnect optimizations which help boost the performance of multiprogramming and multithreaded workloads. We provide experimental results that show that the overall throughput of multiprogramming workloads (consisting of two and four programs) can be improved by up to 60% with our configurable bus architecture. Similar gains can be achieved also for multithreaded applications as shown by further experiments. Finally, we present the performance sensitivity of the proposed interconnect architecture on shared memory bandwidth availability.

1. Introduction and Motivation

Designing an efficient interconnection network for a chip multiprocessor (CMP) is a challenging problem. On one hand, as gate delay reduces with shrinking process technologies, the relative delay of global wires increases [1], thus increasing the latency of the interconnect compared to the compute logic of the system. On the other hand, increasing number of cores in a CMP places a corresponding increasing demand on the bandwidth requirements of an interconnection network. Both these problems are depicted in Figure 1 which shows that the increasing delay of wires and the increasing number of cores that are utilized on a CMP result in more conflicting requests for a shared bus (Conflicting requests result when a request has to wait in a queue because the bus is currently not available. This data was collected for a multicore processor with a separate request bus and response bus. Conflicts for bus were measured only for request bus. The workload in both cases consisted of two and eight applications respectively from the SPEC benchmark suite. The detailed parameters of the multicore processor modeled are described in Section 5). This large increase in conflicts at the interconnect increases the resolution time for memory reference instructions and is one barrier to the high performance and throughput of multicore processors.

One limiting factor to the efficiency of any interconnection network is that it is designed to serve a diversity of workloads. Therefore, a particular network topology may serve efficiently only a small subset of potential workloads. The adaptability of network topology to dynamic traffic patterns is therefore a useful property of any interconnection network.

In this paper, we propose a reconfigurable approach to the design of interconnection networks so that the interconnect could be configured on-demand based on

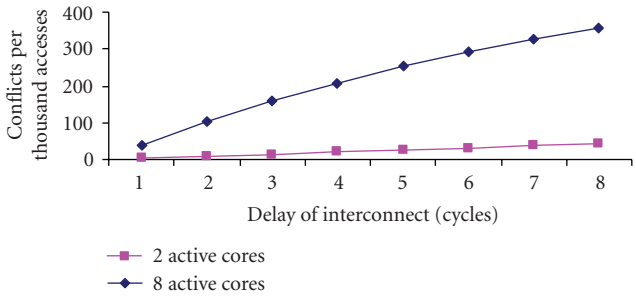


Figure 1: Conflicts per thousand accesses for a shared-bus interconnect as more cores are made active and the delay of interconnect increases for a workload consisting of random SPEC applications.

workload characteristics. Since the interconnection network serves as a communication mechanism between the different memory modules in the system, the interaction of the workload with the memory system has direct implications on the performance of the interconnection network. If this interaction is known *a priori* or could be determined at run time, the interconnection network could be adapted to serve each workload efficiently. Our proposal consists of clustering the cores of a CMP into many groups to localize the traffic within a cluster and then augmenting clustering by inserting reconfigurable logic between clusters. Configurable logic placed in this manner is used to maintain coherence between clusters if required. As a result, we can either isolate the clusters and localize traffic or provide different policies for communication between clusters.

The outline of this paper is as follows. In this section, we will first introduce some requirements of future interconnection networks for multicore processors. Since workload adaptability of the network is the main theme of this paper, we will then describe several scenarios where the workload characteristics interact closely with the memory subsystem and the interconnection network. Then, we will provide some background on reconfigurable interconnects and how they can be useful for interconnection networks for multicore processors. In Section 2, we will introduce the baseline interconnect on top of which we will add support for reconfiguration. In Section 3, we will describe a framework at the hardware level to support reconfigurable interconnects. In the same section, we will also discuss the support required at the system level and mechanisms for supporting configuration of interconnect. In Section 4, we will discuss further optimizations possible at the hardware and the software layer. In Section 5, we will present the experimental methodology used to evaluate different interconnection networks and the data collected for different benchmark suites. Finally, we will discuss some conclusions of the research presented in this paper.

1.1. Requirements of Future Interconnection Networks. In this section, we will outline two requirements of future interconnection networks for multicore processors. The interconnection network framework proposed in this paper addresses these two requirements described below, that is, workload adaptability and efficient broadcasting.

1.1.1. Workload Adaptability. Different workloads stress the interconnection network in different manners. For a single application, the variation in performance demands from different resources in a uniprocessor is a well-studied problem [4]. For a multicore processor, the workloads are more diverse. This includes a mix of multiprogramming workloads and multithreaded applications with different communication patterns. These different workloads put varying performance demands on the interconnection network.

1.1.2. Efficient Broadcasting. Efficient broadcasting is extremely important for multicore processors with several cores. Broadcasting is necessary for maintaining cache coherence among the cores that execute different threads of an application. Coherence messages are broadcast to ensure correct handling of shared data among the cores. The overhead of coherence messages can be reduced with selective broadcasting, that is, selective dispatching of such messages only to cores that indeed share application data. In the case of shared-bus interconnect, appropriately configuring circuit-switched connections between cores can enable effective communication through selective broadcasting.

1.2. Interaction between Workloads and Interconnection Networks. In this section, we will motivate the design of workload-adaptive interconnection networks by providing some examples that demonstrate the strong interaction between workload, memory system, and interconnection networks.

As a first example, we will show how the interaction of workload with memory system provides an opportunity to optimize the interconnect latency incurred by a particular type of traffic. In multiprocessors, requests for reading data generated by individual processors can be categorized as those ultimately satisfied by the underlying shared memory (also called memory-to-cache transfers) and those satisfied by the private memory of one of the other processors (also called cache-to-cache transfers). In the former case, it does not matter if other processors are aware of the request or not. However, in the latter case, the interconnect needs to provide a mechanism to other processors so that other processors are able to probe the requests. Figure 2(a) shows a large increase in cache-to-cache transfers for some Splash benchmarks resulting from an increase in the size of the private L2 cache of cores. With smaller L2 caches, the number of cache-to-cache transfers is small because the probability that some processor has any given line in dirty state is small. When we increase the cache size, a bigger fraction of the working set resides in the caches, which results in increased communication through cache-to-cache transfers. Thus, if we know this information beforehand, the interconnect could be configured to speed up cache-to-cache-transfers for

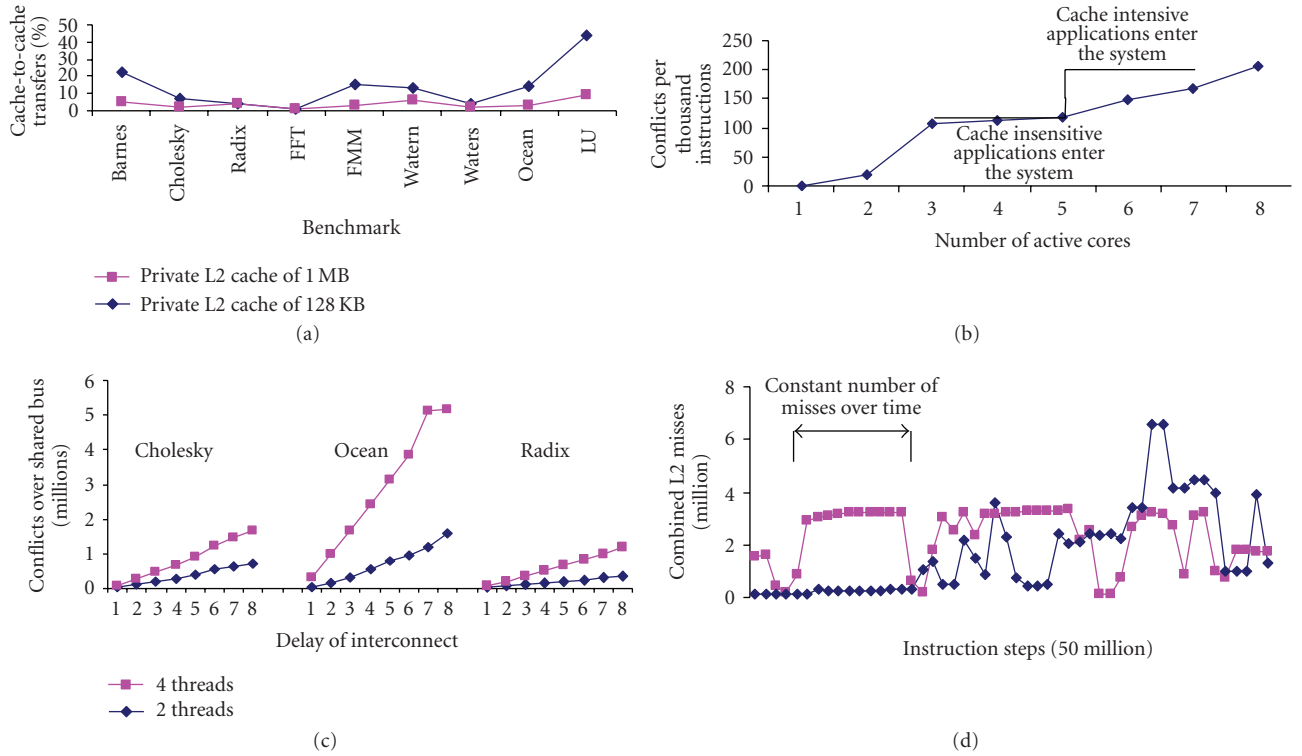


FIGURE 2: Examples of workload interaction with cache memories and interconnection network; (a) The Impact of cache size on the amount of cache-to-cache transfers; (b) the impact of the number of active cores on the interconnect conflicts; (c) The impact of the number of utilized threads on interconnect conflicts; (d) L2 misses of two concurrently running applications.

a given workload; for instance, by setting up circuit-switched paths between frequently communicating cores.

Figure 2(b) shows that once an additional core is made active to run a new application, the resulting conflicts for access to the interconnect from different cores increase in a predictable manner depending upon the cache behavior of applications. There are two distinct areas shown in the figure. The first one corresponds to a very low increase in conflicts for shared-bus. This is because applications that enter the processor do not send a large number of requests over the interconnect. The second adjacent area corresponds to the arrival of applications that send a large number of requests over the interconnect. If there is a hardware mechanism to detect the gradient of curve shown in Figure 2(b), the interconnect can be configured to accommodate the increase or decrease in bandwidth requirements from the current set of active cores. For instance, the interconnection network could be divided at runtime into different regions with each region handling traffic from a subset of cores. Initially, we begin with a monolithic network. As we reach the cache-intensive region in Figure 2(b), we divide the network into different regions to separate the traffic being generated from different cores so as to reduce conflicts over the interconnect.

Figure 2(c) is an example of how for the same problem size (three Splash benchmarks), the use of additional threads to solve the same problem results in varying increases in conflicts on the interconnect. In case of Cholesky and Radix, there is a small increase in conflicts when we move from two threads to four threads. However in case of Ocean, the increase in conflicts is significant. If this interaction is known, the interconnect can be tailored specifically depending upon the number of active threads used to solve the problem. For instance, consider a clustered multicore architecture in which each cluster of cores is connected by a shared-bus. The shared-bus segments of different clusters are further connected to each other by glue logic. In the case of Cholesky and Radix in Figure 2(c), the four threads can be scheduled on a cluster with four cores. However, in the case of Ocean, the four threads can be divided among two clusters with each cluster consisting of two cores. Such a segmented interconnection network architecture in tandem with an appropriate mapping policy for threads could handle the increase in conflicts more gracefully.

Also, Figure 2(d) shows, for different workloads (consisting of two SPEC applications), the variation in combined misses in L2 caches for a modeled multicore architecture. In this experiment, we calculate the number of combined L2 misses from two active cores that result in requests being sent over the interconnect every 50 million instructions. It could be seen that there is a considerable variation in the way an application loads the interconnect over different program phases. Also, we have labeled the area in the figure where there is a constant number of misses over a long sequence of instructions. This behavior could be utilized for an efficient sharing of bandwidth provided by the interconnect.

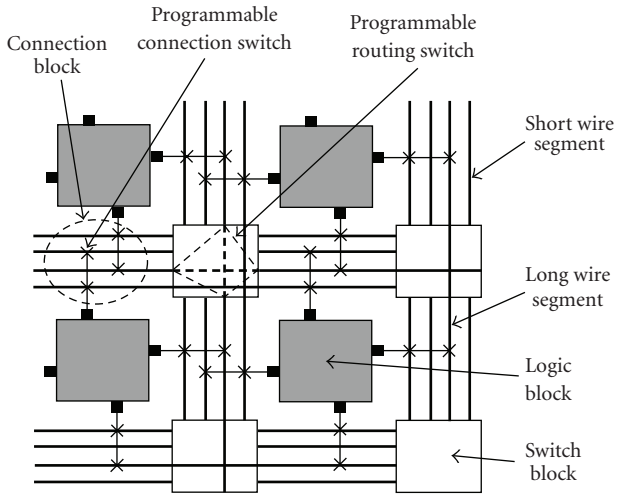


Figure 3: The architecture of island-style field programmable gate array.

In all of the examples above, an adaptive, workload-aware interconnect will be able to respond to different workloads and changing program phases to support the current demands of bandwidth and latency efficiently.

1.3. Reconfigurable Interconnects. Traditional wisdom suggests taking a general approach to the design of an interconnection network to achieve independence from application characteristics [5, 6]. This is necessary for traditional multiprocessors with off-chip interconnects as it is not convenient to provide run-time configurable application-specific paths. Application-specific paths in the interconnect normally involve extra traffic due to various control signals and use up extra pins (a limited resource) in multichip modules. However, in the case of on-chip interconnects, this strategy needs to be reconsidered. In Section 1.2, we motivated the need for workload adaptability in on-chip interconnects. Tight integration of cores and interconnection network on the same chip provides an opportunity to take an application-specific approach for the design of interconnection networks. Reconfigurable interconnects can be utilized to introduce application-specific characteristics in on-chip interconnects.

Reconfigurable interconnects were made popular by their usage as a fine-grained switching fabric for routing of signals among logic blocks in field programmable gate arrays (FPGAs) [7]. The architecture of an island-style FPGA model is shown in Figure 3 [8]. As shown, the resources on chip are divided among logic blocks which are used to implement arbitrary Boolean functions and switch blocks that allow connections among the logic blocks. The logic and switch resources can be mapped and programmed using computer aided design (CAD) tools. Figure 3 depicts the details of the switch block (or switch box). As shown, the switch box routes an incoming signal to one of the three outgoing terminals. The selection of one of the three outgoing directions is made by the configuration bits associated with the switch box.

In this paper, we propose the use of switch blocks, as shown in Figure 3, for interconnection networks of multicore processors. These reconfigurable switch blocks are used to route messages to one or more of the possible destinations. In particular, the shared-bus fabric of current multicore processors is a monolithic bus. We propose to segment the bus into different sections and insert the switch blocks among these segments. We will discuss the details in a subsequent section. However, a motivational example is as follows.

In shared-bus architectures, a transaction consists of many subtransactions. For instance, in order to load data from memory, a processor issues a request on the bus. The completion of this request consists of making all other processors connected to the bus aware of this request, collecting and analyzing the responses and finally receiving data through the bus. Each of these sub-transactions takes place on a different segment of the bus. These sub-transactions can occur in parallel. Thus, a monolithic shared-bus consists of many horizontal wire segments each handling a different sub-transaction as shown in Figure 4(a). The horizontal segments are connected to each other, the cores and the memory modules. We propose to split the horizontal segments into vertical segments as shown in Figure 4(b).

Normally in the monolithic shared-bus shown in Figure 4(a), each request issued by a core will have to traverse the entire length of horizontal wire segment thus incurring a very high latency. By creating a vertical split, each vertical segment can accommodate a subset of cores with direct communication, thus reducing the communication latency. This can be useful in cases where intensive communication occurs only between a subset of cores. The switch box can isolate the segments as it consists internally of tri-state buffers. Furthermore, if configured properly, it can send the request from one vertical segment to the other vertical segment. Depending upon the fan-out of the switch box, a message arriving from one vertical segment can be routed to any of the horizontal segments of the other vertical segments.

2. Shared-Bus Interconnects

This paper aims at making the shared-bus interconnects found in current chip multiprocessors workload-adaptive. Although point-to-point meshes (e.g., network-on-chip (NoC)) architectures have been proposed as a scalable solution for multicore processors [9], the use of shared-bus interconnect is likely to remain popular in the near future for chips with a moderate number of cores. This is because the design of snoopy-based interconnection networks is simple and entails a lower area overhead compared to directory-based schemes (i.e., directory memories and directory-management logic). Moreover, our proposed scheme for reconfigurable clustering of the processor cores can potentially allow the use of a larger number of cores while mitigating the penalty of a highly contended shared-bus.

2.1. Baseline Interconnect. In a multicore processor with private caches, the transactions involve more than just address

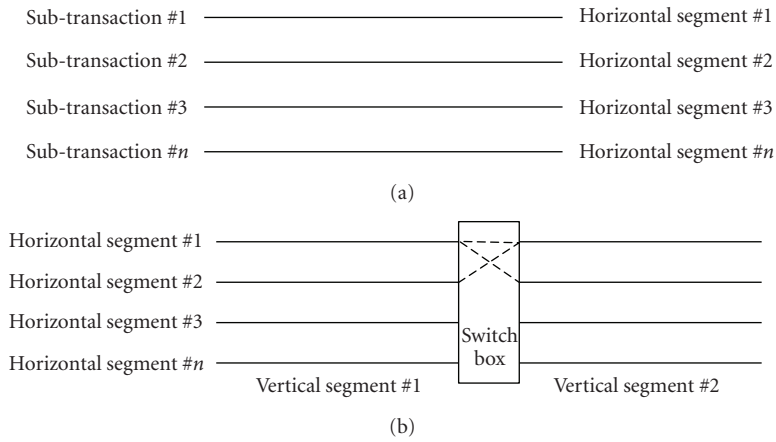


Figure 4: (a) Monolithic shared-bus with horizontal segments for sub-transactions; (b) shared-bus with horizontal segments and a vertical split with a switch box inserted among the vertical segments.

and data transfers. This is due to maintaining coherency of the caches. Additionally, a given request could ultimately be served by multiple sources. Furthermore, a transaction might change the state of data in other modules. A shared-bus interconnect that serves all these purposes is shown in Figure 5 [10]. The bus-based interconnect shown in the figure consists of multiple buses, a centralized arbitration unit, queues, and other logic.

The last-level private cache of each core in the system is connected to the address bus through (request) queues. All components that could possibly serve the request are connected to the snoop bus. The request travels to the end of the address bus where it is stored in another queue. From there, the request travels across the snoop bus. All components connected to the snoop bus read the request and check to see if they could serve the request. Access to the address bus is provided by the arbiter. The address bus and snoop bus are the only broadcast buses in the interconnect. Bandwidth limitation of the address bus can be overcome by aggressive pipelining of the bus in order to allow each cache to send the request simultaneously. Arbitration logic becomes area-intensive with increasing levels of pipelining.

Each component connected to the snoop bus receives the request, processes it and dispatches a response on the response bus. Processing involves looking up the private cache tags to check for the requested data. All responses travel to the end of the response bus where a piece of logic (bookkeeping logic) collects them and generates a combined response which travels back along the response bus. The combined response includes information regarding action that each component needs to take. Actions include sending data on the data bus or changing the coherence status.

The shared L3 cache can snoop requests in two ways. The first involves waiting for the responses to be collected by the bookkeeping logic from the response bus and then sending the request to the cache controller below if none of the private caches of other cores could service the request. The second approach is to snoop from somewhere along the snoop bus. The two approaches have a delay-power tradeoff. In this paper, we assume that the L3 cache snoops from the middle of the snoop bus. That is, the L3 cache could read the coherence messages from the snoop bus similarly to the L2 caches and provide a reply concurrently with the higher level caches.

2.2. Ordering of Requests and Latency Overhead. In Figure 5, the request queue at the end of the address bus is a common ordering point for all requests. Each request is broadcast onto the snoop bus from the request queue. The inherent ordering point in shared-bus interconnects can simplify the implementation of sequential consistency. On the other hand, it can result in communication overhead. As can be deduced from Figure 5, the overhead of a single transaction involves traversing the point-to-point links on different buses as well as the logic overhead of arbiters for gaining access to the bus. Furthermore, some cores may not need to take part in the coherence mechanism described above. For example, an application which is part of a multiprogramming workload could send a request to the L3 controller once an L2 miss is detected. Similarly, requests generated by independent multithreaded applications do not need to pay the penalty of coherence communication imposed on traditional interconnects like the one depicted in Figure 5. Our proposed technique aims to reduce the overhead of redundant serialization which is inherent in bus-based interconnect, without limiting the broadcast capability.

2.3. Reconfigurable Shared-Bus Interconnect. In this section, we will briefly introduce how reconfigurable interconnects can be used to improve the performance of the shared-bus fabric described in Section 2.1. Figure 6(a) depicts a high-level view of the baseline-shared-bus interconnect. For simplicity, only the address bus and snoop bus are shown. As can be seen, requests from all cores arrive at a queue (RQ) which is the global ordering point. From RQ all requests get serialized. The address bus is granted to any of the cores by an arbiter (not shown).

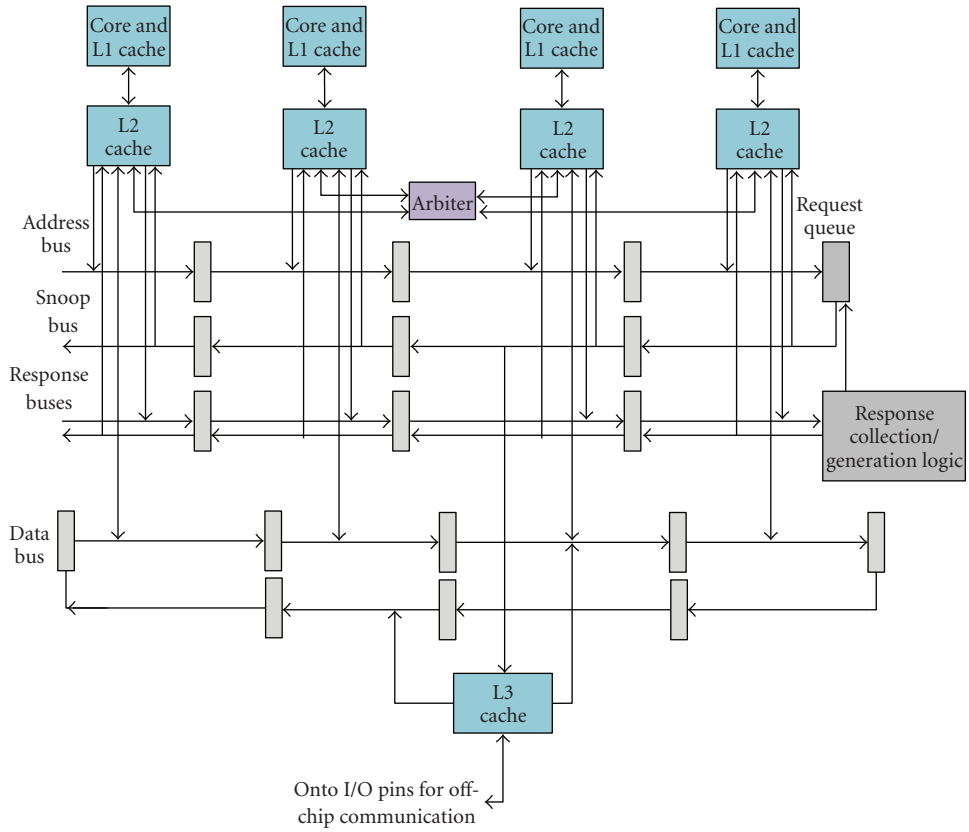


Figure 5: Detailed model of a shared-bus interconnect for CMPs.

The high-level view of the proposed reconfigurable shared-bus interconnects is shown in Figure 6(b). The available cores on the chip are now grouped into clusters. Each cluster has two cores, independent queues, and arbiters in this case.

The detailed framework and hardware resources utilized will be explained in next section. The box labeled *switch box* in Figure 6(b) can be abstracted at this point to serve the following purposes.

(i) It isolates the incoming signals from traveling across to the neighboring cluster.

(ii) It can send the incoming signal to any of its outputs.

(iii) The behavior of either isolating incoming signals or forwarding them across any of the outputs is selected by configuring bits.

Note that in Figure 6(b), two clusters are shown only as a motivational example. We can group the cores into more than two clusters (each cluster can also have different number of cores) depending upon the number of cores on chip and the target applications.

2.4. Configurations of Proposed Interconnect. The three configurable modes of the interconnect that make it more efficient than the baseline interconnect are explained below.

2.4.1. Region Isolation. The shared-bus interconnect in Figure 6(a) is now divided among different regions. These regions can now be isolated from each other as shown in Figure 6(b). This implies that there is no single global ordering point for coherence requests, which results in a different perceived order of requests by cores in different clusters. Independent multithreaded applications can use the cores within single isolated regions for mapping their threads. This configuration is also useful for multiprogramming workloads. Each region still has a local ordering point, and therefore sequential consistency is implemented within each cluster core set. Region isolation is depicted in Figure 6(c) in which the two regions are shown, labeled 1 and 2. The bold lines show the ordering of requests within region 1, and the thin lines show the ordering of requests within region 2. The switch box prohibits signals from either region passing to the neighboring region.

2.4.2. Region Fusion. Two or more regions in Figure 6(b) can be combined to act as one monolithic region. This is useful when a multithreaded application has more threads than the number of cores in a region. In this case, the cores from two or more regions need to have a common ordering point. Interconnect fusion helps to achieve it by combining two regions and making them act as a monolithic region. When two regions are fused together, the entire region has a single ordering point.

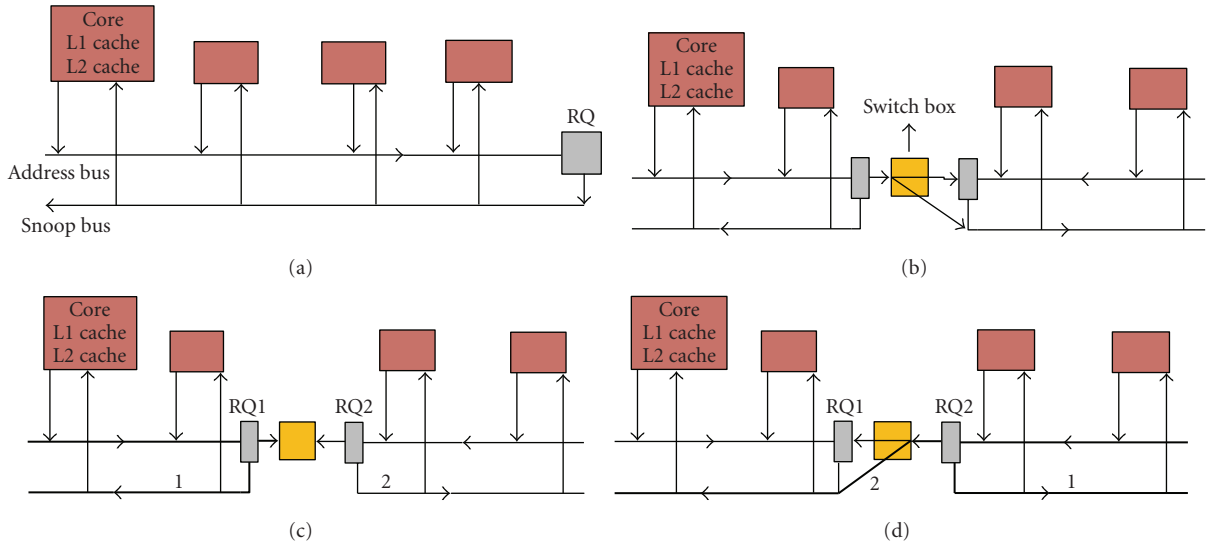


Figure 6: Examples of proposed interconnects; (a) High-level view of baseline shared-bus interconnect; (b) High-level view of proposed reconfigurable shared-bus interconnect; (c) Shared-bus interconnect depicting region isolation; (d) Shared-bus interconnects depicting interconnect fusion.

As an example of region fusion, consider Figure 6(d) in which we show four cores divided into two clusters. Each cluster has two cores and a local ordering point (RQ1 and RQ2). Also, each cluster has a local arbiter. In the case of region fusion, only one request from either of the two clusters is granted access at any point in time. In Figure 6(d), a request from the right cluster has been granted access to the bus. The request follows the path labeled 1 and is made visible to the right cluster. Also, after arriving at the switch box, the request takes the path labeled 2 and is made visible to the left cluster. In Figure 6(d), the paths marked with bold lines are used to make the requests visible to all cores and to maintain coherence.

The configuration depicted in Figure 6(d) results in an obvious degradation in performance compared to the configuration in Figure 6(b). This is because at any given point in time, only one request from either RQ1 or RQ2 is selected to be propagated to both regions labeled 1 and 2 in order to preserve global ordering. In the next section, we describe a configuration of our proposed interconnect that helps to improve its performance for a single multithreaded application whose threads are scheduled across clusters.

2.4.3. Region Confederation. In this configuration, the switch box shown in Figure 6(b) sends selective incoming traffic to other regions. This property is helpful for single multithreaded applications whose threads are scheduled across clusters. The decision to send the incoming traffic to a particular region can be made either statically or dynamically based upon the information in incoming requests. In both cases, the hardware bits are set according to the requirements. As for static decision making, we will describe the complete procedure in Section 3. The mechanisms described in that section involve extracting the knowledge of expected communication pattern of a workload beforehand. For dynamic configurations, there are two possibilities. One possibility is to use sharing pattern predictors [11] and the other is to categorize the traffic and use some header bits to configure the paths during runtime. The latter approach is used in this paper and will be described in Section 3.

2.5. Memory Consistency Model. In this section, we present the memory consistency models [2] supported by the baseline interconnect in Figure 6(a) and in the various configurations of the proposed interconnect. However, let us first discuss the association of memory consistency and interconnection network design.

The architecture of the interconnection network may not impose the type of consistency model, but it does affect the efficiency of its implementation. Thus, shared-bus interconnects are better suited to stricter consistency models, such as sequential consistency, due to the inherent sequentialization of requests at a common ordering point. On the other hand, sequential consistency in point-to-point interconnects can cause a significant delay overhead due to the requirement of ensuring a common ordering point for requests from all processors. Weaker consistency models allow more freedom in the perceived order of memory references by the different cores in the multicore processor and thus are usually preferred in systems with point-to-point interconnects. Shared-bus systems can also benefit from weaker consistency models, especially when the cores are clustered as in the proposed interconnect scheme. In the following, we will discuss which consistency models can be efficiently supported by our proposed techniques.

Starting from the baseline interconnect depicted in Figures 5 and 6(a), we can easily support sequential consistency due to the inherent ordering of requests in shared-bus interconnects. The same applies when running independent applications across clusters such as in the case depicted

in Figure 6(c), where region isolation is the implemented intercluster communication policy. As we mentioned above, sequential consistency can be used regardless of the properties of the interconnect. Thus, the interconnect fusion policy shown in Figure 6(d) also easily supports sequential consistency by ensuring that at any given point in time only one request from either RQ1 or RQ2 is selected to be passed through both regions labeled 1 and 2. Until a request has traversed both paths (bold lines in Figure 6(d)), a new request is not sent off on the interconnect. It should be noted that implementing sequential consistency on a clustered shared-bus system may incur some extra overhead compared to the baseline shared-bus system due to the extra levels of arbitration and the overhead of the intercluster switch logic.

Additionally, we also support weaker consistency models that allow a more flexible ordering of requests and can also eliminate some of the intercluster coherence traffic. In the interconnect configuration described in Section 2.4.3 and implemented in Figure 6(b), requests are not perceived in the same order by all cores. Furthermore, some requests may not be communicated beyond cluster switches if they can be satisfied within the home cluster. This type of weaker consistency, apart from allowing a more flexible ordering of requests, enables a special type of selective broadcasting within the cluster region and eliminates the overhead of intercluster traffic.

In order to support weaker consistency, hardware or software support is required for specially handling synchronization operations. We used the M5 simulator [12] for evaluating the interconnection networks described in this paper. M5 uses the Load-Link/Store-Conditional operations [13] for supporting synchronization among different threads of the same application. It should be noted that weaker consistency models do not place any limitations to the programmer compared to sequential consistency, whereas they are usually easier to implement in terms of hardware complexity. In fact, most multiprocessor systems are designed to support weaker consistency models.

3. A Framework to Support Reconfigurable Interconnects

In this section, we will describe an interconnection network architecture that utilizes reconfigurable interconnects for improved performance over the baseline interconnection network described in Section 2. The proposed chip architecture allows one to relax the serialization constraint on bus-based interconnects by clustering the cores into groups and localizing the traffic generated by an independent workload within a cluster.

3.1. Chip Overview and Clustering. The high-level view of the baseline multicore processor for eight cores is shown in Figure 7(a). In the baseline architecture, all cores are connected through a monolithic shared-bus. There could be multiple L3 controllers simultaneously serving different requests destined for different banks. The proposed architecture for the multicore chip is shown in Figure 7(b). In Figure 7(b), the shared-bus is split among clusters and there is no global ordering point. Each cluster has a local arbiter, and therefore requests within a cluster are ordered. If threads share data across clusters, intercluster logic is set up to provide communicating paths among clusters. For the proposed architecture, there could be an L3 controller per cluster or multiple clusters could share a single L3 controller.

The number of independent clusters available on the chip depends upon the granularity of the reconfigurable logic provided on the chip. For instance, for a CMP with sixteen cores, providing four reconfigurable points will make it possible to have four independent clusters with each cluster having four cores. For this architecture, we assume a MOESI snoopy-based cache coherence protocol [14] within a cluster and a weaker consistency model for access to shared data across clusters. In particular, writes to data shared across clusters are protected by locks.

3.2. Communication Policies and Reconfigurable Interconnection Logic. The reconfigurable logic between clusters has three goals. First, it provides on-demand isolation between clusters, thereby not having to send coherence traffic across clusters. This will be particularly helpful for multiprogramming workloads as they could be scheduled within one cluster. Secondly, for multithreaded workloads, based upon the expected communication patterns, it supports two different policies for communication among clusters.

(i) *As Soon As Possible Policy (ASAP).* A request from one cluster is sent immediately to another cluster to be serviced without waiting for final response generation by bookkeeping logic. If the probability of finding data in the other cluster is high, this policy reduces latency for the requesting cluster.

(ii) *If Necessary Policy (IN).* A request from one cluster is sent to the other cluster after first being assured that the requested data does not reside locally. This policy is useful for coarse-grained multithreaded applications that communicate rarely.

It should be noted that if the architecture supports the above two policies, we can implement the interconnect models described in Section 2.3. However, one additional mechanism that needs to be supported for the interconnect model in Section 2.4.2 is the setting up of communication between arbiters so as to maintain sequential ordering of requests in the resulting fused interconnect. This could be done by having additional levels of arbitration.

In subsequent sections, we will discuss a mechanism to select between the two policies. For now, it suffices to say that the selection between two policies depends upon the confidence of the programmer regarding the expected communication pattern among different threads. Figure 8 illustrates the rationale behind the two policies supported by intercluster logic.

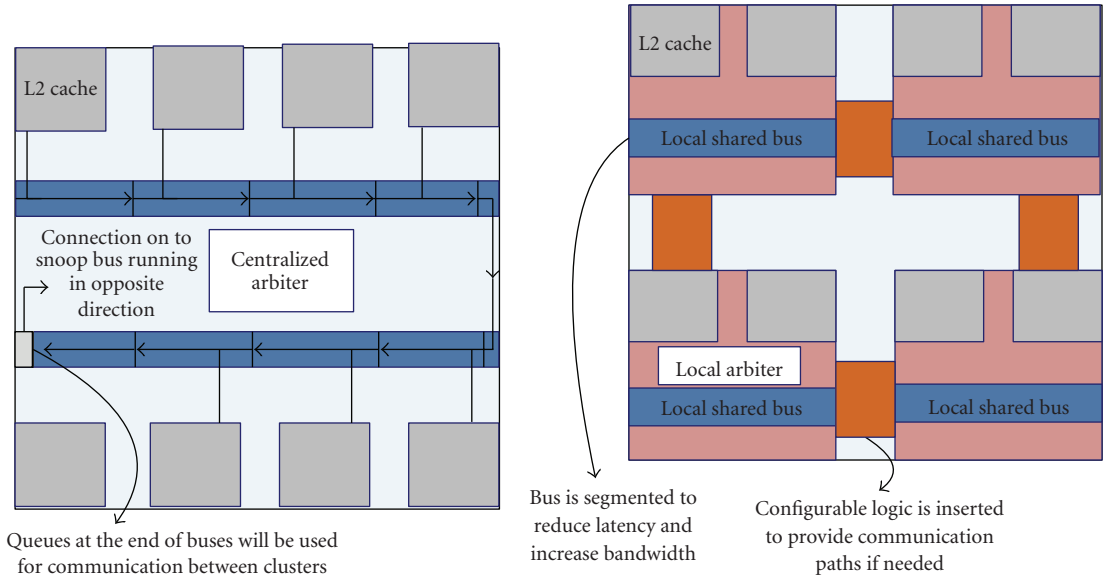


Figure 7: (a) Baseline multicore architecture versus; (b) Clustered multicore architecture.

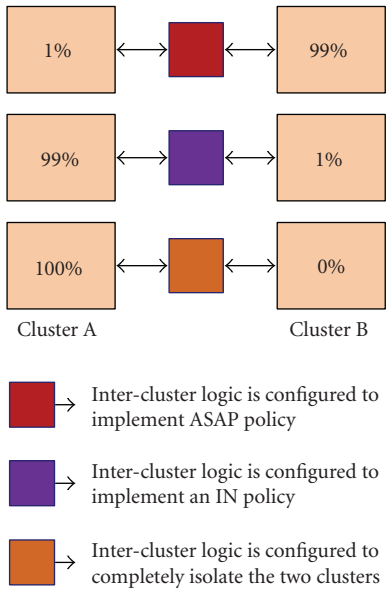


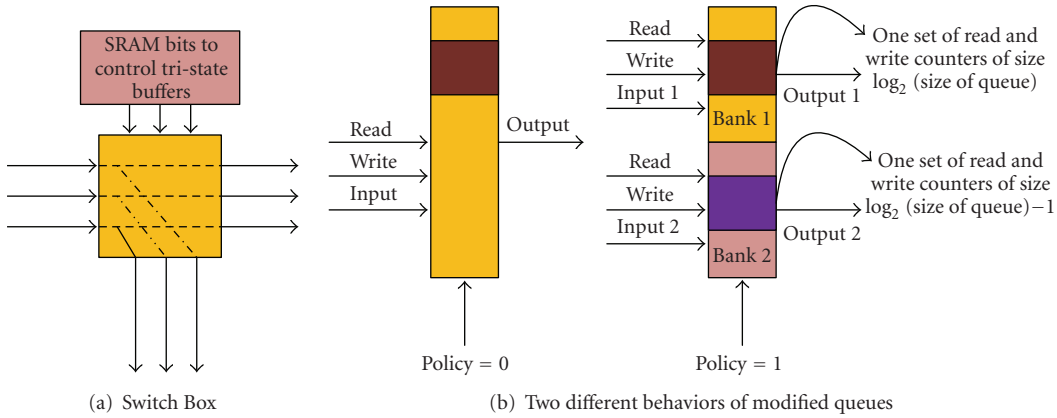
Figure 8: Example Scenarios for Using the Two Policies for Communication between Clusters and Isolation Property of Clusters.

The next two sections provide implementation details of the reconfigurable logic between clusters with the goals described in this section.

3.3. Reconfigurable Interconnect Overhead Cost. In this section, we will describe the additional logic components that will be utilized. We will also provide the area and timing overhead of the additional components for 65 nm technology (TSMC) using Synopsys Design Vision. Note that in this section, the additional logic components are described with respect to a clustered multicore architecture consisting of two clusters. The projection to multiple clusters is straightforward.

3.3.1. Switch Boxes. The switch box fabric used for routing purposes in field programmable gate arrays [15] is an important structure to be utilized in on-chip reconfigurable interconnects. A Switch box, as shown in Figure 9(a), provides different routing paths for the incoming signals. The number of outgoing routing paths is called the Flexibility, Fs, of a switch box. For our design, the required Fs is two. The number of switches required for our switch box is therefore $Fs * W$ (for each direction), where W is the width of the bus. We assume the presence of tri-state buffers as switches inside the switch block. Tri-state buffers are used to either isolate the clusters or to setup a unidirectional communication path if required. The area overhead of a 64-bit bus switch box at 65 nm was found to be 430 μm^2.

3.3.2. Modified Queues. Queues are the primary mode of communication in our design as shown in Figure 7. We modified the queues as follows. If the cluster is isolated from the neighboring cluster, the queue can buffer Q requests thus only serving the local cluster. However, if the clusters share data, the queue is configured to be partitioned into two banks. One bank of size $Q/2$ takes requests from the local cluster, and another bank of size $Q/2$ buffers requests from the neighboring cluster. The two behaviors are depicted in Figure 9(b). The queue is provided with two read ports and two write ports to communicate with other clusters. The counters within the queue are managed to reduce the area overhead while supporting both cases as shown in Figure 9(b). A configuration bit selects the required behavior of the queue. The area overhead of the modified queue over



(a) Switch Box (b) Two different behaviors of modified queues

FIGURE 9: Two major components used for reconfigurable logic between clusters.

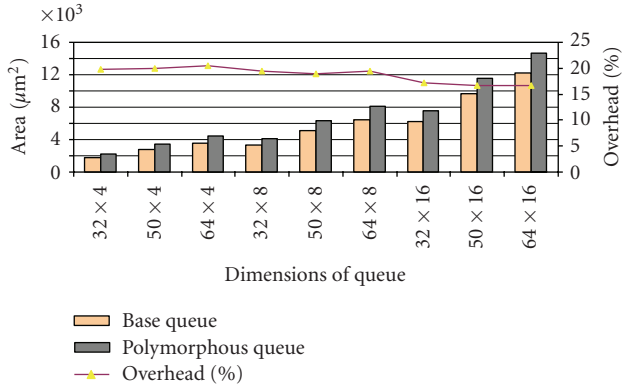


FIGURE 10: Area overhead of modified queues over base queues.

the base queue for different capacities and bus widths at 65 nm is shown in Figure 10.

3.3.3. Multiplexors. We made use of multiplexors in our design to support the ASAP policy when clusters are sharing data and for the interconnect optimization proposed for single applications (discussed later). The area overhead of the multiplexor was found to be 200 μm^2 for a bus width of 64 bits at 65 nm.

The area overhead of the above logic components required for intercluster logic is not prohibitive, thus providing the designer with freedom to provide many configurable points on chip. In order to configure the above components, we assume the availability of hardware bits that are set at run time by the operating system. For 65 nm, we chose a clock cycle of 1.5 GHz for simulation results in Section 5. All components passed the one-cycle delay constraint for bus frequency of 1.5 GHz.

3.4. Complete Design of Intercluster Logic for Two Clusters. Figure 11 shows two clusters connected through the intercluster logic. Additional logic (consisting of a switch box and a multiplexor) is only shown from left cluster to right cluster. We need to provide two different paths for communication between clusters and a way to isolate the clusters. The electrical properties of tri-state buffers serve to provide the isolation between clusters. The first path corresponds to the ASAP policy among clusters. In this case, we want the request in one cluster (named as local cluster) to be sent immediately to the neighboring cluster. For this, as the request passes through the switch box, it is routed on to the snoop bus of the neighboring cluster through the path labeled ASAP in Figure 11, where a multiplexor drives it on the snoop bus. Every time a request is buffered in the local queue, the neighboring queue is prohibited from issuing any further requests on the snoop bus. The multiplexor is simultaneously configured to select the request coming from the switch box. For the IN policy, the switch box is configured to send the incoming request along the path labeled IN in Figure 11, and the neighboring queue is signaled to store the request. The bookkeeping logic (BKL) of the local cluster signals the neighboring queue to issue the request on its snoop bus only after collecting responses and finding out that the request cannot be serviced locally. The queues in this case are multibanked as was discussed in Figure 9(b).

3.5. Software Support. In this section, we will discuss the system-level support required to use our proposed interconnection network.

3.5.1. Programmer's Support. We propose to use the programmer's knowledge of the expected communication pattern of the workload to be run on the system. The information regarding the expected communication pattern is assumed to be provided by the programmer through annotations in the code. The following annotations are applicable to the architecture described in Section 3.1.

(i) *Single Threaded.* This is a single-threaded application.

(ii) *Coarse-Grained Sharing.* This group of threads is part of a multithreaded application, and the threads have coarse-grained sharing among them.

(iii) *Fine-Grained Sharing.* This group of threads is part of a multithreaded application, and the threads share data at a finer granularity.

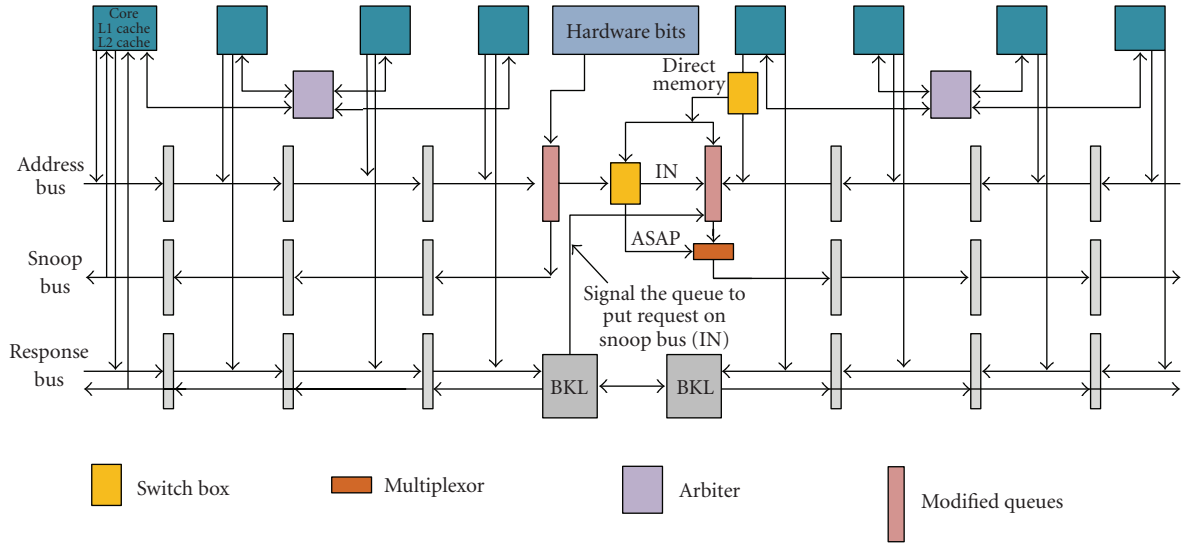


Figure 11: Intercluster logic between two clusters.

The use of annotations to configure the underlying parameters through hardware bits is the same as in [16]. The above annotations are used to select between the two policies described in Section 3.2. If the threads scheduled on two clusters share data at a fine granularity, we propose to use the ASAP policy. However, if the data shared is at a coarser granularity, we propose to use the IN policy for communication. In the next section, we will give an example of selection between these policies by using profiling data from realistic workloads.

3.5.2. Operating System Support. The Operating System (OS) support can be used in two ways. First, the annotations can be used by the compiler to generate instructions for the operating system (OS) to set hardware bits and configure the switches, queues, and multiplexors as discussed in Section 3.3. Second, modern operating systems have affinity masks to schedule threads to cores. If we can provide the operating system with knowledge of the expected communication pattern among threads in our application, the OS can make use of interconnect-aware scheduling to improve performance of applications.

The embedded annotations can be used by the OS to determine scheduling and mapping of applications on the available cores and also configure the switches according to one of the communication policies described in Section 3.2. Let us use the example depicted in Figure 12 which is based on a 4-thread application (T1,T2,T3,T4) running on a 4-core CMP. As Figure 12(a) shows, there are six possible communicating thread pairs, whereas Figure 12(b) shows the relative distribution of communication within those thread pairs for the SPLASH benchmarks. *Sxy* denotes the portion of total cache-to-cache messages that were sent between cores x and y. Annotations indicating the sharing pattern between threads can be applied, as illustrated in Figure 12(b), to guide the mapping of threads onto cores and the intercluster communication policies. Figure 12(c) shows the scheduling and communication policy decisions made by the OS for the RADIX benchmark run on a 4-core CMP with 2 clusters. These decisions were guided by the values of *Sxy* for different pairs of x and y cores running RADIX threads. Threads T1 and T2 are mapped to one cluster since the communication among them is high. Same applies for threads T3 and T4. The logic between clusters is configured to use ASAP policy since T1 and T4 communicate very often. The frequent communication between T1 and T4 is also what guides the OS to schedule these threads on cores P2 and P3, respectively.

3.6. Dynamic Configuration of Hardware Bits. We also require a mechanism to dynamically configure the hardware bits. The need for dynamic configuration is based on the different sharing patterns for different types of coherence messages. In the example of Figure 12, we examined the aggregate cache-to-cache transfers regardless of coherence message types. However, our profile results show that the degree of sharing might vary considerably for different types of messages. Thus, it is important to have a mechanism to support dynamic configuration of hardware bits.

Support of dynamic configuration can be done by encoding additional information in the request. This additional information describes the type of request sent over the interconnect. Since we are using the five state MOESI protocol for maintaining coherence in the proposed interconnects, the majority of requests that traverse the interconnect can be one of the following.

(i) *ReadReq and ReadResp.* This request corresponds to a simple read request for cache block and the corresponding response with data.

(ii) *ReadExReq and ReadExResp.* This request corresponds to a request for a cache block with exclusive access to the cache block. The data is not currently present in the cache. Another cache in local or any other cluster might have a copy of data. The response ensures that all other copies of this block in the system have been invalidated.

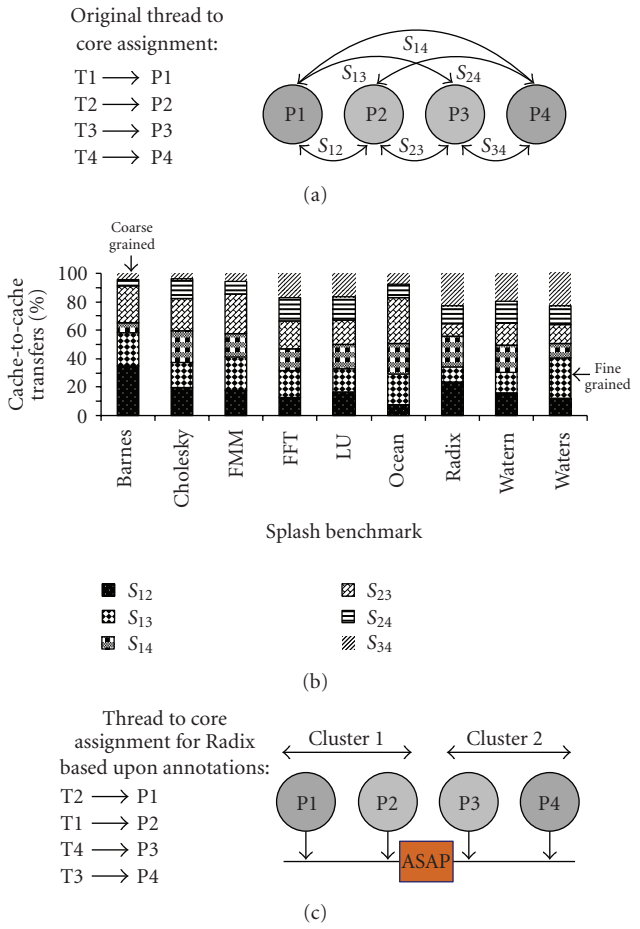


FIGURE 12: (a) Four cores and possible communication pairs; (b) example of annotations in splash benchmarks based upon expected communication patterns among thread pairs; (c) resulting scheduling decisions and policy selection between clusters.

(iii) *UpgradeReq and UpgradeResp.* There is a valid copy of block present in cache. However, the cache requires exclusive access to data in order to update it. The response does not contain data but ensures that all other copies have been invalidated. The response is sent by the current owner of block.

(iv) *Writeback Req.* Another frequent request on the interconnect is Writeback request. Writeback requests appear on the bus when a block in dirty state in L2 cache is replaced. The dirty block is written back to L3 cache after passing through the interconnect. There is no response corresponding to this type of request.

Note that the above bus transactions are typical of an interconnect that supports snoopy coherence protocols. Further details can be found in [17].

Each of these requests can be encoded by bit patterns that determine the opening or closing of paths in the switch box during runtime. Additionally, one bit in the request can inform the logic circuitry at the request queue whether the hardware bits are statically set or are to be set dynamically. If the bits are statically set, the request queue can just forward the request to the switch. Otherwise, a logic circuitry is used to read the bit pattern in the request and configure the switch accordingly. This corresponds to using the header flits to set up circuits in traditional circuit-switched networks [6].

We will use Figure 11 to elaborate the datapath for supporting dynamic configuration of switch boxes. If the request is residing in RQ1 and needs access to the ASAP path, RQ1 communicates with RQ2 to signal RQ2 to stop sending more requests from its local cluster on snoop bus. In the mean time, it can also configure the switch box to send the request along the ASAP path.

4. Additional Optimizations

In this section, we will describe two additional optimizations that can help to improve the performance of the baseline interconnect described in Section 2. The first optimization is performed at the hardware level while the second is performed at the operating system level.

4.1. Direct Memory Connection. For single-threaded applications running on a CMP, the requests for memory accesses do not need to go through the shared-bus interconnect at all, and a direct connection to underlying memory will enhance the performance of single-threaded applications. Therefore, a connection between the request queues at the output port of the L2 cache and the input port of the L3 cache controller is useful in Figure 5.

Since all transactions begin when a request is placed on the address bus, a switch box is placed at the output of the request queues connecting the L2 cache to the address bus. One output of the switch box is routed to the address bus and the second output is routed directly to the L3 controller. A multiplexor at the input of L3 controller selects between the request coming from path labeled *direct memory* in Figure 11 and the regular path for memory accesses (somewhere along the snoop bus). In this case, the L2 cache controller is inhibited from sending requests to the arbiter. We evaluated the performance improvement for single-threaded applications resulting from the direct memory connection in Section 5.

The direct memory connections can be provided to a small subset of cores to reduce the associated cost of additional hardware. Also, if multiple direct memory connections are provided, arbitration is needed to send request to L3 cache. In the results provided in this paper, we will assume the presence of one core provided with a direct memory connection.

It should be mentioned that a direct memory connection is also useful for sending writebacks straight to memory instead of routing them through the interconnect. The MOESI protocol ensures that writebacks only take place if a cache block is in dirty state. As only one cache in the processor can have the block in dirty state, no other cache in the processor needs to snoop on writebacks going to L3 cache.

4.2. Interconnect-Aware Scheduling. There is also a possibility of improving the performance of a multiprogramming

workload by intelligent scheduling within a cluster. This is because, as could be seen in Figure 5, latency of access to many resources (arbiters, queues) is location-dependent. If an application communicates with the arbiter very often, it scheduling closer to the arbiter will increase its performance. The selection policy of the arbiter and the dynamic traffic pattern of a particular workload are also the factors that could be taken into account for scheduling of processes to cores.

Single-threaded applications forming part of the multiprogramming workload could be mapped to a single cluster. This will give the operating system a larger decision space when a multithreaded application enters the system.

In Section 5, we evaluate the performance improvement possible for multiprogramming workloads by using interconnect-aware scheduling. The applications were chosen at random from SPEC benchmark suite. All possible scheduling permutations of selected workloads were evaluated for performance on a cluster with four cores. The permutations were selected such that not to repeat a workload twice. For instance, 2 of the possible 24 permutations of four workloads $\{A, B, C, D\}$ are $\{A, D, C, B\}$ and $\{D, B, C, A\}$.

5. Methodology and Results

The experimental results and methodology are divided into two sections. In the first half, we will describe the methodology and results for multiprogramming workloads. Multiprogramming workloads constitute an important spectrum of workloads for future multicore processors. While new parallel applications will emerge, traditional single-threaded applications will persist. Therefore, it is important that multiprogramming workloads do not suffer from the overhead of coherence management that results for correct execution of multithreaded applications. In the second half of this section, we will discuss the methodology and experimental setup for evaluating multithreaded benchmarks.

5.1. Methodology for Multiprogramming Workloads. We evaluated our proposed architectural techniques using the M5 simulator [12]. We modified the M5 shared-bus interconnect to implement separate address, snoop, response, and data bus as shown in Figure 5. All buses are pipelined. Caches are modeled such that requests arriving at time X and Y incur a latency of X + Latency and Y + Latency regardless of $X - Y$. The data bus is modeled as a bidirectional bus. Each request has a unique TAG associated with it. The TAG has both an ID of the request itself and an ID of core that generated the request. Based upon the TAG in the request, the L3 cache controller places the data on the appropriate direction along the data bus (see Figure 5) after arbitrating for it. For this set of experiments, the L3 controller always snoops the request from the middle of the snoop bus and is later inhibited from servicing the request if the request is found in the private cache of some other core.

We performed the experiments across four technology generations. The scaling of frequency of cores is taken from the ITRS roadmap. The frequency of shared-buses is assumed to be half of the core frequency. This assumption is consistent with existing CMP architectures [18]. The different technology nodes, clock frequency of cores and of bus fabric is shown in Table 1. The chip area is assumed to be constant at 400 mm^2 due to yield constraints. When we scale down from a higher technology node to a lower technology node, we assume that twice the number of cores (along with associated private caches) is available. The parameters of a single core and caches are shown in Table 2. Our methodology to model wire delay is as follows. If there are n cores connected to the address bus, we pipeline the wire n-way with n latches. The rationale behind this methodology is to allow each cache connected to the bus to send a request every cycle. The delay of the link between two latches is always one cycle. The length of a link calculated in this manner is used as latch spacing for the remaining wires on the chip. The logic delay of the arbiter is not modeled. However, behavior of the arbiter is modeled such that no two requests conflict for any segment of the pipelined address bus. We considered applications from the SPEC benchmark suite for the evaluation of our proposed architectures. Simulations were run in a detailed mode for 200 million instructions after fast-forwarding the initial phase for 2 billion instructions. Since the evaluation of our proposed ideas depends heavily on the underlying cache miss rates of workloads, the miss rates of considered SPEC benchmarks are shown in Figure 13.

Table 1: Node parameters.

Node	Number of Cores	Core Frequency	Bus Frequency
90 nm	4	2 GHz	1 GHz
65 nm	8	3 GHz	1.5 GHz
45 nm	16	4 GHz	2 GHz
32 nm	32	6 GHz	3 GHz

In the following set of results, the M5 simulator was run in Syscall emulation mode. In this mode, the simulator does not boot an actual operating system. The system calls are emulated by the host-operating system.

5.2. Results for Multiprogramming Workloads. In this section, we show the results for the improvement in performance of using the proposed multicore architecture with various levels of clustering over the baseline processor. Indirectly, we show the reduction in the impact of global wire delays in shared-bus chip-multiprocessors for different workloads. We use the following terminology for our modeled architectures.

(i) *DM.* One core has been provided with direct connection to memory as shown in Figure 11.

(ii) *CX.* The cores are divided into X clusters. For instance, for 32 nm C2 means that the processor has two clusters with sixteen cores per cluster.

Delay of switching components and additional wires is included in simulations wherever applicable.

Table 2: Core Parameters.

Parameter	Value
Processor cores	Alpha 21264 2-issue
L1 D-Cache	32 KB 2-way set associative, 1-cycle hit latency 64-byte cache lines, 10 MSHRs
L1 I-Cache	64 KB 2-way set associative, 1-cycle hit latency 64-byte cache lines, 10 MSHRs
L2 Cache	1 MB 8-way set associative 5-cycle latency 64-byte cache lines, 20 MSHRs
Shared L3 Cache	36 MB 16-way set associative 40-cycle latency 64-byte cache lines, 60 MSHRs
Physical memory	512 MB 200-cycle latency

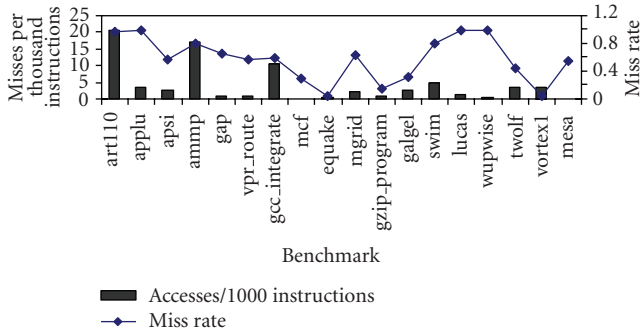


Figure 13: Miss rates and accesses per thousand instructions for considered SPEC benchmarks.

5.2.1. Performance Analysis for Single Applications. Figure 14(a) shows the average latency incurred by L2 cache misses for various sizes of clusters running a single application. It also shows the average latency of L2 misses when the application is running on a core which has direct connection to the shared L3 cache below. In the results shown in Figure 14(a) where one core has been provided with a direct connection to the shared L3 cache, we do not assume a clustered architecture. Although Art and Ammp have high miss rates and large number of misses, their average miss latency is smaller because these benchmarks have very high L3 cache hit rates. It can be seen in Figure 14(a) that the direct memory approach can not compete with the 2-core cluster configuration (C16). This is because, by providing a direct memory connection, we only get rid of the latency incurred by the address and snoop buses. The latency of the monolithic data bus is still visible to the L2 miss. This motivates us to consider an architecture that combines the effect of clustering (cluster with two cores) and direct memory connection described in Section 4.1. Figure 14(b) shows the results of performance improvement with this combined effect over the base case for all technology nodes. Performance improvements are shown in terms of cycles per instruction (CPI). As we scale down, the CPI of the baseline processor increases due to more point-to-point connections, but the improvement of the enhanced interconnect does not increase significantly. Therefore, the overall performance in terms of CPI improves for clustered multicore architecture with direct memory connection. Performance gained is very high as we scale down the technology node, and applications have high miss rates.

5.2.2. Performance Analysis for Multiprogramming Workloads. We evaluated the performance of workloads running on independent clusters using multiprogramming workloads. We created multiprogramming workloads consisting of two and four applications by selecting the applications among the benchmarks shown in Figure 13. The workloads were created to have a varied representation of L2 cache miss rates. In the following experiments, we initially run the workload on a baseline processor. Subsequently, we run the workloads on clusters of finer granularity (less number of cores per cluster) and note the reduction in latency. The results indicate that as we adapt the architecture to best serve the workload, overall performance is always improved. It should be mentioned that the entire workload is scheduled in a single cluster and the performance boost comes from the shrinking of the bus interconnect to a cluster-wide bus.

Figure 14(c) shows that for a multicore processor with four cores modeled after 90 nm technology and running two applications, there is a moderate performance gain with clustering. The floorplan considered for this experiment is four cores connected in a straight line with the arbiter in the middle. Applications were mapped to the first two cores along the straight line. This architecture is depicted in Figure 15.

From the collected data, we found a high correlation between prospects of performance improvement and $(A1 + A2)$ where $A1$ and $A2$ equals the misses by the L2 cache of core 1 and core 2, respectively. The greater the value of $A1 + A2$, the greater the opportunity to optimize the portion of IPC related to traffic due to L2 cache misses.

Figure 14(d) shows the results for different levels of clustering for a CMP with sixteen cores modeled after 45 nm. Performance gains increase as fabrication technology scales down, because the relative delay of global wires increases, and clustering has a greater advantage.

From the data collected for baseline processors modeled after 65 nm and 45 nm running four applications, the increase in CPI when scaling down from 65 nm to 45 nm is shown in Table 3. For the first workload in Table 3, the increase in CPI due to delay of global wires is 30%. This offsets the advantage of a 25% increase in core frequency as we scale down. Our proposed architecture reduces this impact of wire delays considerably.

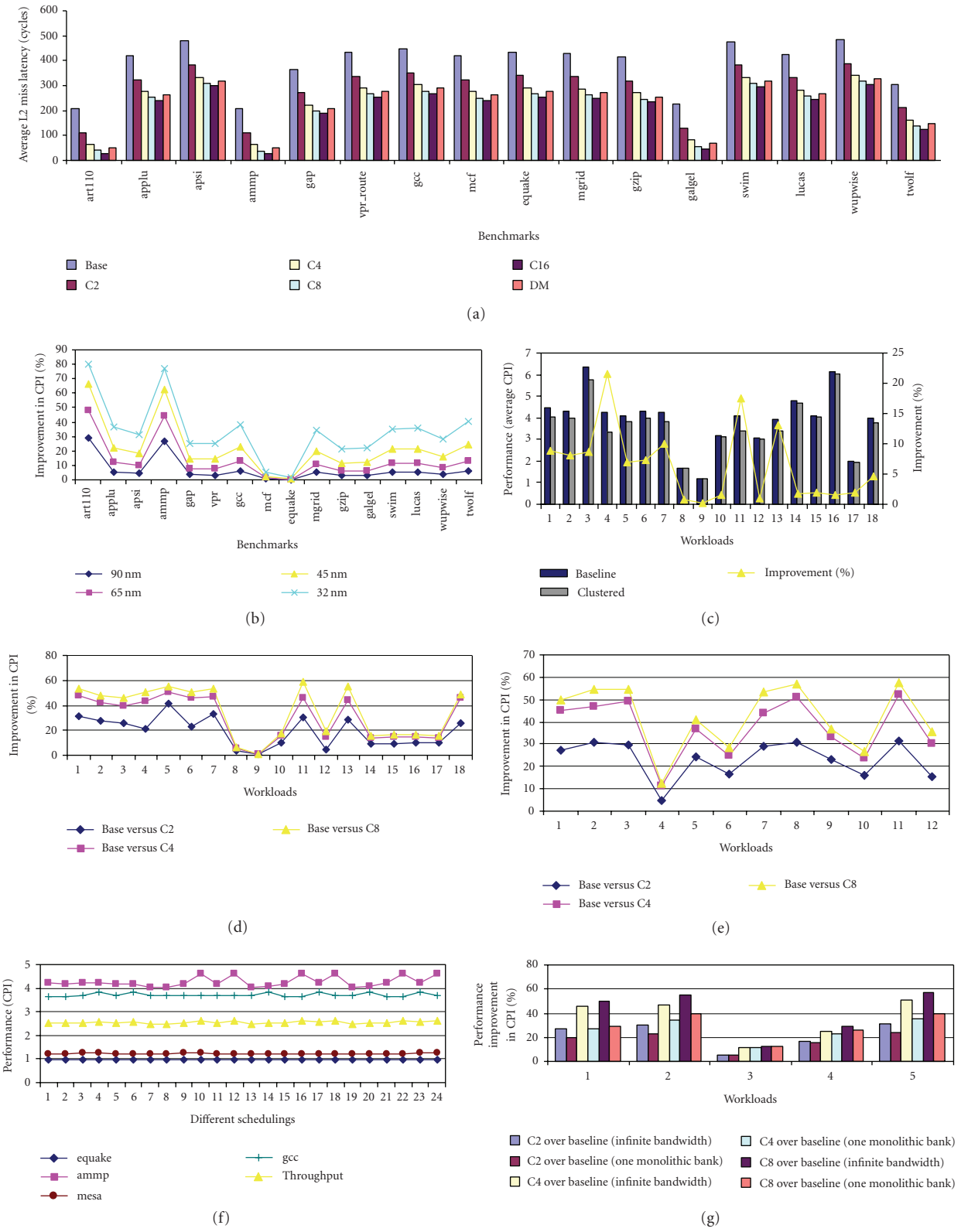


Figure 14: Performance improvement for multiprogramming workloads; (a) Average L2 miss latency for different architectures running single application (32 nm); (b) Performance improvement for single application using a cluster of 2 cores and direct memory connection to the shared L3 cache; (c) Performance improvement for different architectures running 2 applications (90 nm); (d) Performance improvement for different architectures running two applications (45 nm); (e) Performance improvement for different architecture running four applications (32 nm); (f) Impact of scheduling on performance within a single cluster; (g) Impact of L3 bandwidth limitation on performance gained with clustering.

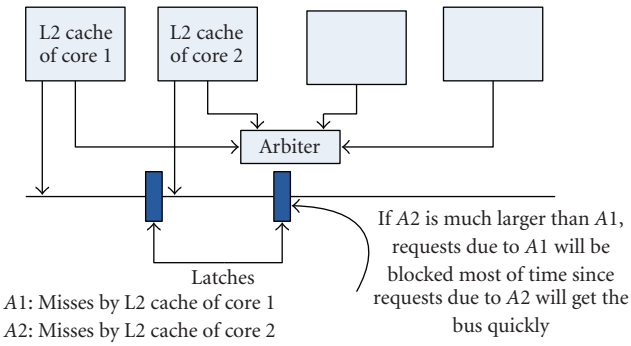


Figure 15: Factors behind Performance Improvement Shown in Figure 14(c).

Table 3: Comparison of CPI between baseline processors modeled after different technology nodes.

Workload	CPI of Processor		Increase in CPI
	65 nm	45 nm	
ammp, swim, gcc, vpr	4.8	6.361	24.5%
ammp, apsi, applu, mgrid	4.29	5.92	30%
galgel, apsi, applu, swim	3.86	4.092	5.66%
galgel, lucas, ammp, wupwise	3.68	5.18	28.95%
mcf, equake, apsi, applu	2.14	2.56	16%

The performance results for four applications running on a 32 core multicore processor are shown in Figure 14(e). For 32 nm, we observed that the performance gains for some workloads are as high as 60%.

Note that in Figure 14(d), the performance improvement in C8 compared to C4 is not significant. This is related to the fact that in C8 all the cluster cores are busy since the number of applications matches the number of cores in the cluster. The benefit of reduced delay is now offset by increased contention for the arbiter and shared-bus since there are more requests per unit of time (as the cluster shrinks each request is served faster and thus more requests are submitted per unit of time). This can be overcome by mapping multiprogramming workloads across clusters. This effect is also observed in Figure 14(e).

5.2.3. Performance Analysis of Scheduling within a Cluster. Figure 14(f) shows the impact of scheduling on the performance of a workload running on a single cluster with four cores. The x-axis shows 24 different possible schedules for a workload consisting of four applications given that no application is repeated twice in any schedule. Different curves in Figure 14(f) show the performance of individual applications and the overall throughput of the complete workload. It is depicted in the figure that there is a 6% variation in overall throughput and 12% variation in CPI of the most cache-intensive benchmark (Ammp), by using interconnect-aware scheduling. We analyzed many results for schedules using different multiprogramming workloads. The variation in performance is a function of many parameters such as positioning of cores relative to arbiters, arbitration policy, size of queues, and dynamic traffic pattern generated by the benchmark.

The improvement in performance through scheduling relies on the programmer to provide reasonable estimates regarding the expected interaction of application with memory. The OS scheduler can then assign incoming applications in a manner such that the most appropriate core is reserved (closest to arbiter, etc.) for the most memory-intensive application.

5.2.4. Analysis of Bandwidth Requirements of Shared L3 Cache for Clustering. Any improvement in the latency and bandwidth of interconnect for chip multiprocessors will stress the underlying memory system respectively. While we considered the availability of a large number of banks in our results shown in Figure 14, in this section we will do some analysis of the dependence of our proposed techniques on the bandwidth of the shared L3 cache. For this, we chose to run simulations using a chip multiprocessor with 32 cores running four applications with different levels of clustering. For comparison with results in above section, we modeled the L3 cache as one monolithic bank of 36 MB. Hits are queued up while misses are not affected due to the presence of large number of miss handling status registers (MSHRs) [19]. Figure 14(g) shows the performance gained both with one bank and with a large number of banks in L3 cache.

Our analysis indicates that for workloads that have low L2 miss rates and thus low performance gain potential, L3 bandwidth does not have a big impact on performance. However, workloads with high L2 miss rates are significantly affected by a limited L3 bandwidth. We also analyzed the impact of performance loss due to the presence of no MSHRs in our L3 caches. Not surprisingly, as we make clusters of finer granularity, the performance loss increases. We observe that the interconnect optimizations proposed should be complemented by aggressive techniques to increase shared memory bandwidth to lower memory hierarchies. However, a moderate degree of banking could still offer a reasonable performance improvement.

5.3. Results for Multithreaded Workloads. In this section, we will describe the modeled architecture and experimental methodology for multithreaded workloads. The benchmarks used for evaluation in this section are taken from the Splash benchmark suite [20].

5.3.1. System Model. For multithreaded workloads, we ran the simulator in full-system (FS) mode. In FS mode, the simulator boots an actual operating system. We utilized a recent copy of Linux kernel as the operating system. The Linux kernel scheduler is used for scheduling threads in FS mode. Also, the complete I/O system is simulated. Figure 16 depicts the high-level view of the complete (baseline) system simulated in FS mode.

In Figure 16, the direct memory accesses are handled through the IO bus and IO bridge. The IO cache is used to maintain coherence between the IO subsystem and the cores in the system. The IO cache snoops the requests in a similar

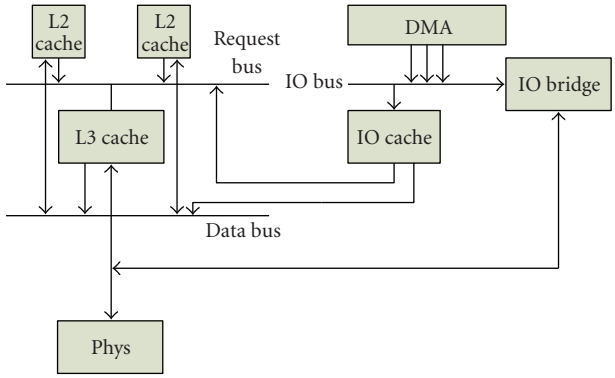


Figure 16: The system architecture used for full-system mode simulation.

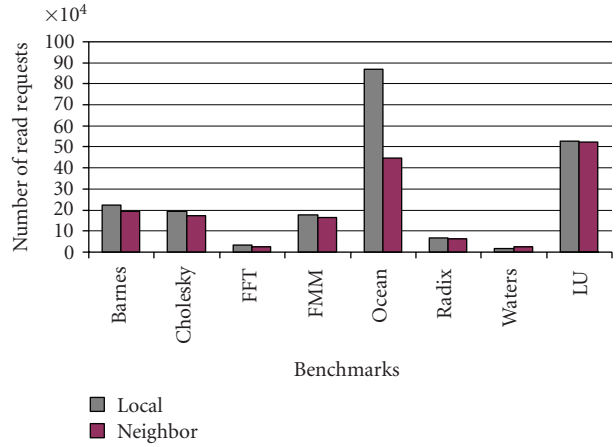


Figure 17: Profiling results for read requests.

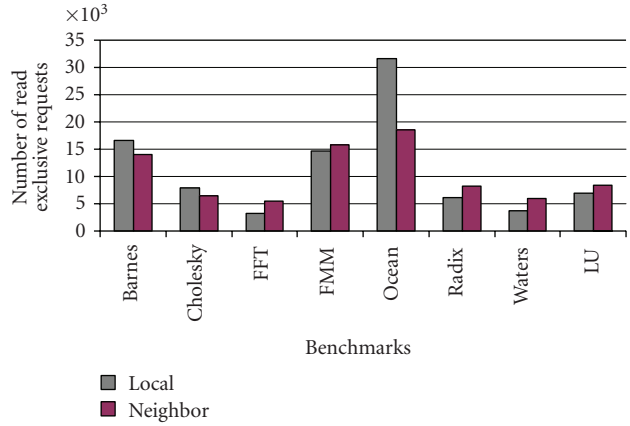


Figure 18: Profiling results for read exclusive requests.

manner as the rest of L2 caches in the system. Note that in Figure 16, the address bus and snoop bus are encapsulated into the request bus for simplicity. Similarly, the response and data bus is shown as a single data bus.

Since for evaluating multithreaded workloads the design space is very large, we modeled one particular system for evaluations in this section. The system modeled is a multicore processor with 8 cores and 2 clusters. Each application is run with eight threads. The baseline is similar to the system shown in Figure 16. In the clustered architecture, the IO cache is connected to the request bus of one of the clusters. The IO cache sends requests to the cluster it is connected to from where these requests travel to the neighboring cluster. The size of L3 cache was reduced to 4 MB as is appropriate for working sets of Splash benchmarks. The latency of L3 cache was adjusted to be 20 cycles.

5.3.2. Profiling Results for Splash Benchmarks. We profiled the Splash benchmarks to determine the sharing patterns for different types of requests described in Section 3.6. For the profiling results, the benchmarks were run to completion. These results will help to understand the intercluster policies chosen for communication in the next section. Figure 17 shows the results for simple read requests. The two series local and neighbor, correspond to the requests that are served locally and those that are served by the neighboring cluster. The Y-axis shows the number of read requests that are served by the L2 cache of some other core either in the local or a neighboring cluster. In most benchmarks, the read requests that are served locally are more than those served remotely. Especially, in case of Ocean, the spatial locality in terms of sharers is very high. Figure 18 shows the results for read exclusive requests. In this case, the response corresponds to a response with data and grant to write to the cache block. As can be seen, usually the owner of data resides in a remote cluster. However, in case of Ocean, Barnes and Cholesky, there is good spatial locality among sharers.

Figure 19 shows the results for upgrade requests. Apart from Ocean, there is not much spatial locality among sharers. This implies that mostly the owner resides in a remote cluster.

The breakdown in terms of the number of different types of requests is shown in Figure 20. As can be seen, the majority of requests are simple read requests. This is the majority of traffic that needs to be optimized by using reconfigurable interconnect techniques. In the case of Cholesky, Ocean, FMM, and LU, there are a significant number of writeback accesses.

Overall, the results are good for exploiting the interconnect techniques described in this paper. The read requests have a 50% chance of being served locally and thus may not need to be sent to another cluster.

5.3.3. Intercluster Communication Policies. The policies for intercluster traffic bypass are selected as follows. As shown in the previous section, there is a significant opportunity to speed up read requests because, on average for all benchmarks, there is more than 50% probability that these requests will be served by some core within a cluster. Therefore, the read requests are only sent to the neighboring cluster if necessary, that is, if the request is not serviced locally. Although the L3 cache snoops the request from the snoop bus of both clusters, it does not take any action on the request unless it is made sure that the request cannot be serviced by either cluster. The request queues (RQ1 and RQ2) in Figure 11 are divided into two banks. Read request from

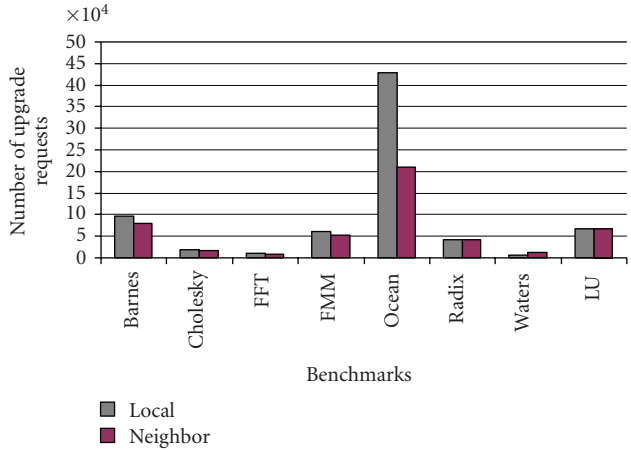


Figure 19: Profiling results for upgrade requests.

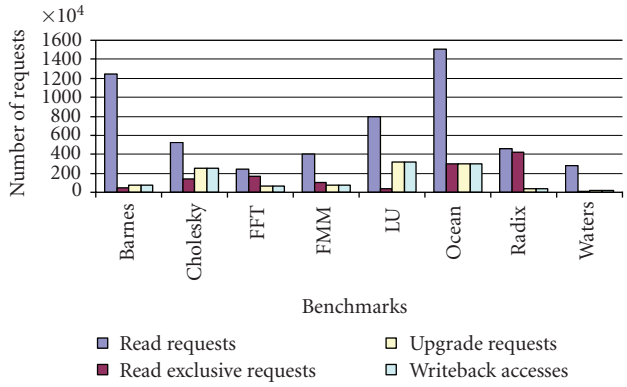


Figure 20: Comparison of number of different type of requests for splash benchmarks.

a local cluster is queued up in one of the banks of the request queue of a neighboring cluster. It waits in the queue until signaled by the local cluster that the request has been satisfied within the cluster. At this point, the counters associated with the queue of the neighboring cluster are adjusted to ignore the request.

The upgrade requests and read exclusive requests do not make up a large portion of total requests. These requests are sent to the neighboring cluster as soon as possible. It should be pointed out that we use the earliest response from any core in the processor for acquiring ownership to a cache block. For instance, in case of an upgrade request, if an owner of a cache block is found in a cluster from which the request originated, the response is sent to the requesting core with ownership of the block. At the same time, the upgrade request is sent to the other cluster to invalidate any cached copies.

5.3.4. Performance Results for Splash Benchmarks. In the following results, we will show performance improvement in terms of reduction in latency for different types of requests generated by L2 caches. In particular, we will show the average latencies for simple read requests, read exclusive requests, and upgrade requests. Finally, each graph shows the

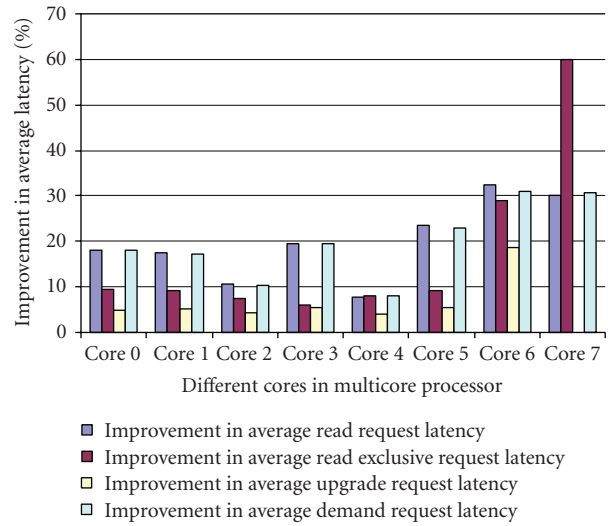


Figure 21: Improvement in average latency for different type of L2 cache requests-LU factorization.

improvement in latency of demand accesses. As a matter of clarifying terminology, we will first describe the difference between an access and a request with respect to an L2 cache before defining demand accesses.

When the L2 cache is accessed to supply data to the L1 cache, the access by L1 cache could end up either as a hit or a miss. In case of a cache miss, the L2 cache controller makes a request consisting of address and related information to identify the request and sends this request over the interconnect. Therefore, a request over the interconnect in this case is a consequence of L2 cache miss. On the other hand, an L2 access is a consequence of L1 cache miss which could end up as either an L2 cache hit or L2 cache miss.

Demand accesses are made up of all the accesses that are generated by the L2 cache. Therefore, these accesses are a sum of read accesses, read exclusive accesses, upgrade accesses, accesses for acquiring locks, and so forth. Demand requests are those generated by L2 cache controller as a consequence of misses resulting from demand accesses.

In the results presented in this section, the simulations were run in a detailed mode for a variable number of instructions (in most cases, each core ran for about 20 million instructions in detailed mode) after fast-forwarding the initial phase for 500 million instructions. The results were collected after booting up of the Linux kernel and loading the scripts.

Figure 21 shows the improvement in L2 cache miss latency for different types of requests for the LU benchmark from the Splash benchmark suite.

The miss latency is measured as the average of all L2 cache misses of a particular type. As expected, the greatest performance improvement is for simple read requests. This is because if a read request is served within a cluster, it is not sent across the cluster. Also, there is some improvement for upgrade requests and requests for exclusive data despite the fact that they suffer from the overhead of communicating

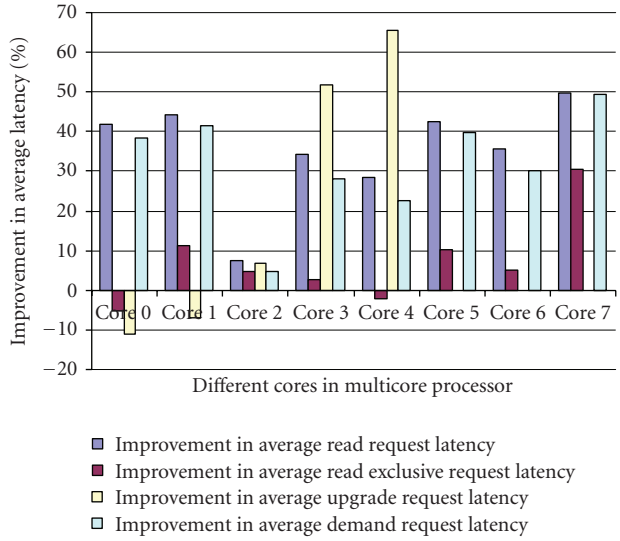


Figure 22: Improvement in average latency for different type of L2 cache requests-FFT.

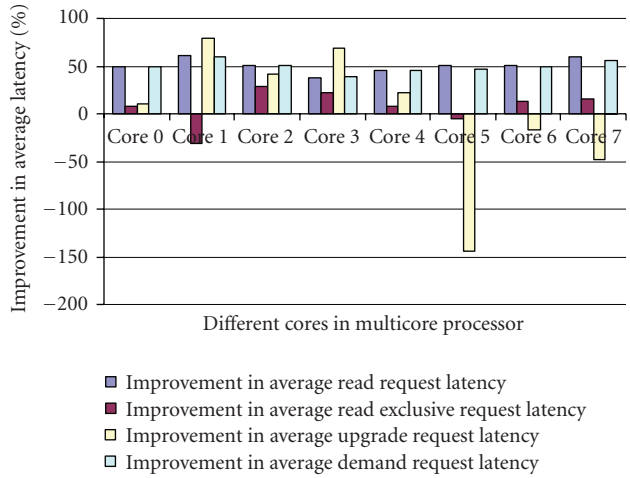


Figure 23: Improvement in average latency for different type of L2 cache requests-OCEAN.

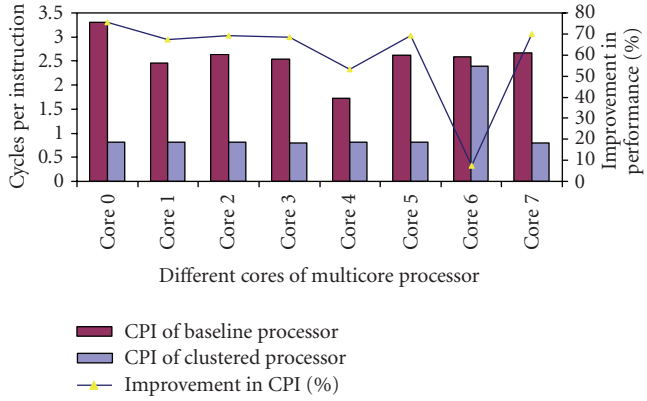


Figure 24: Improvement in CPI-BARNES.

with a neighboring cluster for setting up a circuit switched path. This is because these requests are sent to other clusters as soon as possible. In addition to setting up paths for sending requests as soon as possible, these requests also suffer from the latency of intercluster logic. The average latency of demand accesses always shows improvement. In particular, some cores observe an improvement of 30% for demand accesses. Also, for core 7 the improvement for read exclusive requests is 60% despite the overhead of intercluster logic. This behavior is due to the fact that when we cluster the architecture, we provide improved bandwidth for all requests. With a clustered architecture, there are now 4 cores contending for access to the shared-bus instead of 8 in the baseline case.

Figure 22 shows similar results for the fft benchmark. These results show the negative impact of the latency introduced by the intercluster logic on performance. In particular, core 0 and core1 observe a reduction in performance for requests of exclusive data. In the case of fft, despite the use of ASAP policy for exclusive requests, there was a high waiting time since there were many pending requests waiting to be completed in the local shared-bus of the neighboring cluster. However, it should be noted that overall performance, which is measured by the improvement in average latency of demand request is still significantly high because the majority of requests are simple read requests which still observe a large performance improvement.

Figure 23 shows similar results for the Ocean benchmark. Note that for core 5, the average cache latency has a large negative impact due to the time spent by the requests in request queues while they are waiting for access to the neighboring cluster. The overall improvement in demand accesses is still significant for all cores.

Figure 24 shows the overall performance improvement in terms of CPI for Barnes. All cores in the system except core 6 benefit from improved performance using our proposed shared-bus. The average performance improvement for the entire system is 60%. This observation, combined with the improvement in average latency of L2 cache accesses, illustrates that the reduction in the average latency of L2 accesses translates into considerable performance boost in terms of CPI for memory-intensive applications.

Overall, the above results indicate good performance gains for multi-threaded applications when executed on the proposed interconnect architecture while utilizing the reconfigurable interconnect techniques in this paper. Moreover, the impact of the overhead of intercluster logic is not significant. It has also been shown that since most requests are simple read requests, it is often possible to only broadcast them within the home cluster. This eliminates unnecessary waste of bus bandwidth by constraining part of the coherence traffic within the bounds of a single cluster.

5.4. Reconfiguration Overhead Analysis. In this section, we examine the overhead involved in reconfiguration of bus. First, note that in the case of multiprogramming workloads where complete isolation is required, the bus need not be reconfigured at all. In the case of multi-threaded workloads

where dynamic reconfiguration of bus is required, reconfiguration overhead is incurred each time we switch the bus from using the IN policy to ASAP policy and vice-versa. Note that in Figure 20 the probability of having a simple Read Request (for which we use an IN policy) over the interconnect is far greater than the Upgrade Request or Read Exclusive Request (for both of which we use ASAP policy). Also note that since reconfiguration bits are small, the reconfiguration time is negligible.

6. Related Work

The analysis of phase behavior of programs and its impact on underlying hardware structures are well studied. Architectural adaptations based upon workload characteristics can be done in a reactive manner or in a predictive manner. The latter implies the presence of hardware predictors to predict the future behavior of programs by exploiting the periodicity of repeating patterns. In [4], the authors propose to use a predictive approach to architecture adaptation rather than a reactive one. They propose many predictors to predict the expected behavior of a program in the future. A similar work is done by authors of [21]. Dynamic optimization of architecture parameters and hardware structures such as pipeline and memory hierarchy have also been proposed [3, 22]. More recently, there have been proposals for dynamic adaptation of architecture for thermal management [23]. Our interconnect architecture can be tweaked according to workload characteristics either statically by acquiring knowledge of the workload from the application layer or dynamically by employing sharing pattern predictors. Such predictors have been proposed in previous work [11, 24]. Moreover, our interconnect can be adapted on the run based upon the type of traffic. This is possible because of low-latency overhead of reconfigurable logic.

The use of clustering in multiprocessors is not new. It is a straightforward observation that the latency of shared-bus is a problem because requests by all cores in the system are serialized on the bus to maintain coherence. In this respect, Wilson [25] did the classic work on scaling the bus-based multiprocessors by making clusters of processors and connecting them in a hierarchical fashion. Coherence is also maintained in a hierarchical manner. However, we propose the use of simple logic structures to configure the size and behavior of clusters based upon expected communication patterns. Based upon the amount of reconfigurable logic placed on the chip, a certain number of clusters are totally independent. The reconfigurable logic can fuse two clusters to make them act like one at the expense of additional latency overhead. However, the philosophy behind our technique is to efficiently support single-threaded, multiprogramming, and coarse-grained multi-threaded workloads.

Several works exist that reduce the broadcast-based traffic required to maintain coherence using filtering of snoop requests [26, 27]. Such techniques were initially proposed for web servers and they result in an area overhead for maintaining extra information in directories. Our architecture allows different broadcast policies under the isolation and the IN/ASAP configurations based on profiling or traffic type. Further, techniques such as coarse-grained coherence tracking and snoop filtering can augment our architecture further.

In [28], the authors proposed reconfigurable on-chip interconnects for CMPs. However, their proposal is to provide a more fine-grained fabric that can implement different on-chip network topologies depending upon the application. If the arrival of jobs in the system is completely random, then the run-time support required to fully utilize their architecture could be complex. A related proposal for programmable interconnects for array processors was given in [29].

With regard to on-chip interconnects, the authors of [30] use a technique which sets up circuit-switched routing paths at run-time for directory-based NoCs [31]. They propose a hybrid network router architecture for on-chip interconnects that supports both circuit switching and packet switching. In order to exploit the pairwise sharing among cores, the circuit-switched paths are reutilized. Instead of sending each request to the home directory, the likely sharer is predicted and the request is routed directly to the likely sharer. Our scheme is based on a much simpler interconnection architecture where intercluster logic sets up circuit-switched paths to maintain the illusion of a shared-bus.

The flow of requests in Figure 7(b) is similar to that in ring-based interconnects [18]. However, our architecture supports different policies for forwarding and isolating requests between clusters. Also, in traditional ring-based interconnection networks, there is a point-to-point connection between every two nodes and thus the ordering schemes required for maintaining coherence result in an additional overhead [32]. Furthermore, if the number of cores and clusters in the architecture is increased, the flow of requests in the form of a ring is no longer maintained.

In [33], the authors proposed to maintain coherence at the granularity of regions. The information regarding shared regions is stored in a region coherence array (RCA). As long as no other processors are sharing data in that region, the processor can address the data in that region without generating broadcast traffic. In order to avoid broadcast and send requests directly to memory controllers, the authors made use of direct memory connection from the cache to the memory controller on chip. The region coherence technique can be also used in our architecture, to provide an alternative granularity for setting the hardware bits of the reconfigurable switches.

In [34], the authors use programmable switches and a large number of fine-grained processing elements to make a specialized parallel computer that adapts to different communication patterns. More recently, authors in [35] introduced core fusion for chip multiprocessors in which a group of cores could dynamically morph into large CPU. In [36], the authors propose to share a pool of programmable resources among cores to increase their utility. In [37], the authors provide techniques to predict the performance of reconfigurable networks for distributed shared memory multiprocessors using optical interconnects.

7. Summary and Conclusions

In this work, we have demonstrated the benefits of integrating reconfigurability into the interconnection network of a multicore processor in order to enable adaptability to workload characteristics. In this manner the interconnection network can adapt to a big range of different workloads that may consist of single-threaded and multi-threaded applications at a small area cost. As the basis of our interconnection network, we used shared-bus interconnects and we described techniques at the software level to utilize the proposed interconnects efficiently. One implication of these techniques is the need to avoid abstraction of the interconnect architecture from the programmer and the Operating System in future multicore processors. Our experimental results show that the proposed reconfigurable interconnect can offer significant performance benefits for both single-threaded applications and multi-threaded applications (up to 60% performance gains). We have also discussed the hardware support required to build the reconfigurable interconnects and analyzed its area cost.

Furthermore, we have discussed ways to support weaker consistency models in a clustered shared-bus interconnect architecture where the inherent global ordering of the traditional shared-bus interconnect does not exist. We have also looked into the potential of interconnect-aware scheduling of threads onto cores by the operating system. An interconnect-aware Operating System is proposed to allow it to make intelligent scheduling decisions based on the characteristics of the applications.

We strongly believe that adaptive interconnects will be critical in achieving efficient execution of highly diverse workloads in future multiprocessors. As the amount of heterogeneity in workloads and cores increases, the potential improvement of performance due to our proposed interconnection scheme is also going to rise.

Acknowledgments

This paper is partially supported by NSF CCF 07-02501 and NSF CCF 07-46608. The authors used machines donated by Intel.

References

[1] P. Wielage and K. Goossens, "Networks on silicon: blessing or nightmare?" in *Proceedings of the Euromicro Symposium on Digital System Design (DSD '02)*, pp. 423–425, 2002.

[2] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: a tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[3] R. Iris Bahar and S. Manne, "Power and energy reduction via pipeline balancing," in *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*, pp. 218–229, 2001.

[4] E. Duesterwald, C. Cascaval, and S. Dwarkadas, "Characterizing and predicting program behavior and its variability," in *Parallel Architectures and Compilation Techniques*, pp. 220–231, 2003.

[5] L. N. Bhuyan, Q. Yang, and D. P. Agrawal, "Performance of multiprocessor interconnection networks," *Computer*, vol. 22, no. 2, pp. 25–37, 1989.

[6] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, San Francisco, Calif, USA, 2003.

[7] S. Brown and J. Rose, "FPGA and CPLD architectures: a tutorial," *IEEE Design and Test of Computers*, vol. 13, no. 2, pp. 42–57, 1996.

[8] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, Norwell, Mass, USA, 1999.

[9] J. Balfour and W. J. Dally, "Design tradeoffs for tiled CMP on-chip networks," in *Proceedings of the 20th International Conference on Supercomputing (ICS '06)*, pp. 187–198, 2006.

[10] R. Kumar, V. Zyuban, and D. M. Tullsen, "Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling," in *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA '05)*, pp. 408–419, June 2005.

[11] A.-C. Lai and B. Falsafi, "Memory sharing predictor: the key to a speculative coherent DSM," in *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA '99)*, pp. 172–183, 1999.

[12] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 simulator: modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, 2006.

[13] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele Jr., "Lock-free reference counting," in *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC '01)*, pp. 190–199, 2001.

[14] P. Stenström, "Survey of cache coherence schemes for multiprocessors," *Computer*, vol. 23, no. 6, pp. 12–24, 1990.

[15] J. Rose and S. Brown, "Flexibility of interconnection structures for field-programmable gate arrays," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 3, pp. 277–282, 1991.

[16] J. B. Carter, "Design of the Munin distributed shared memory system," *Journal of Parallel and Distributed Computing*, vol. 29, no. 2, pp. 219–227, 1995.

[17] P. Sweazey and A. J. Smith, "A class of compatible cache consistency protocols and their support by the IEEE futurebus," in *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA '86)*, pp. 414–423, 1986.

[18] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner, "POWER5 system microarchitecture," *IBM Journal of Research and Development*, vol. 49, no. 4-5, pp. 505–521, 2005.

[19] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA '81)*, pp. 81–87, 1981.

[20] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pp. 24–36, 1995.

[21] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," *SIGARCH Computer Architecture News*, vol. 31, no. 2, pp. 336–347, 2003.

[22] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO '00)*, pp. 245–257, 2000.

[23] R. Jayaseelan and T. Mitra, "Dynamic thermal management via architectural adaptation," in *Proceedings of the 46th ACM/IEEE Design Automation Conference (DAC '09)*, pp. 484–489, July 2009.

[24] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Memory coherence activity prediction in commercial workloads," in *Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI '04)*, pp. 37–45, 2004.

[25] A. W. Wilson Jr., "Hierarchical cache/bus architecture for shared memory multiprocessors," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 244–252, 1987.

[26] A. Moshovos, G. Memik, A. Choudhary, and B. Falsafi, "JETTY: filtering snoops for reduced energy consumption in SMP servers," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA '01)*, pp. 85–96, October 2001.

[27] A. Moshovos, "RegionScout: exploiting coarse grain sharing in snoop-based coherence," *SIGARCH Compututer Architecture News*, vol. 33, no. 2, pp. 234–245, 2005.

[28] M. M. Kim, J. D. Davis, M. Oskin, and T. Austin, "Polymorphic on-chip networks," in *Proceedings of the 35th International Symposium on Computer Architecture (ISCA '08)*, pp. 101–112, June 2008.

[29] L. K. John and E. John, "A dynamically reconfigurable interconnect for array processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 1, pp. 150–157, 1998.

[30] N. Eisley, L.-S. Peh, and L. Shang, "In-network cache coherence," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '06)*, pp. 321–332, December 2006.

[31] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proceedings of the 38th Annual Design Automation Conference (DAC '01)*, pp. 684–689, 2001.

[32] M. R. Marty and M. D. Hill, "Coherence ordering for ring-based chip multiprocessors," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '06)*, pp. 309–320, December 2006.

[33] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "Improving multiprocessor performance with coarse-grain coherence tracking," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*, pp. 246–257, 2005.

[34] L. Snyder, "Introduction to the configurable, highly parallel computer," *Computer*, vol. 15, no. 1, pp. 47–56, 1982.

[35] P. Salverda and C. Zilles, "Fundamental performance constraints in horizontal fusion of in-order cores," in *Proceedings of the 14th IEEE International Symposium on High Performance Computer Architecture (HPCA '08)*, pp. 252–263, February 2008.

[36] M. A. Watkins, M. J. Cianchetti, and D. H. Albonesi, "Shared reconfigurable architectures for CMPs," in *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 299–304, September 2008.

[37] W. Heirman, J. Dambre, I. Artundo et al., "Predicting reconfigurable interconnect performance in distributed shared-memory systems," *Integration, the VLSI Journal*, vol. 40, no. 4, pp. 382–393, 2007.