Workload Adaptive Shared Memory Multicore Processors with Reconfigurable Interconnects

Shoaib Akram, Rakesh Kumar, and Deming Chen
Department of Electrical and Computer Engineering, University of Illinois at Urbana Champaign
{sakram3,rakeshk,dchen}@illinois.edu

***Abstract*—Interconnection networks for multicore processors are designed in a generic way to serve a diversity of workloads. For multicore processors, there is a considerable opportunity to achieve an improvement in performance by implementing interconnects which adapt to different program phases and to a variety of workloads. This paper proposes one such interconnection network for medium-scale (up to 32 cores) shared memory multicore processors and the associated means at the software level to utilize it effectively. The proposed architecture uses clustering to divide the cores on the chip among many groups called clusters. Reconfigurable logic is inserted between clusters to support either isolation or different policies for communication among clusters. The experiments show that the isolation property of clusters can improve overall throughput of a multicore processor by as much as 60% for multiprogramming workloads consisting of two and four applications. The area-overhead of the additional logic is shown to be minimal.**

I. Introduction and Motivation

The design of an efficient interconnection network for a chip multiprocessor (CMP) is a challenging problem. The relative delay of interconnection wires increase with shrinking process technologies which increases the latency of interconnect compared to the rest of system. Also, increasing number of cores demands more bandwidth from the interconnection network. In fact, for a single core in a CMP, the performance impact due to the latency of an interconnection network can be as high as that of a cache miss.

One limiting factor to the efficiency of any interconnection network is that it is designed to serve a diversity of workloads. For a single application, the variation in performance demands from different resources in a uni-processor is a well-studied problem[5]. For a CMP, the workloads are more diverse. This includes a mix of multiprogramming workloads and multi-threaded applications with different communication patterns. Different workloads stress the interconnection network in different manners. For instance, Fig.1A shows the varied manner in which conflicting requests over the shared bus increase as more applications are introduced in a 8-core CMP (refer Table.II for parameters). This is due to different memory requirements of applications. Also, there is a considerable variation in the bandwidth requirements during different phases of a single workload. This is shown in Fig.1B for some workloads each consisting of two SPEC benchmarks. In this paper, we suggest a reconfigurable approach to the design of interconnection networks so that the interconnect could be configured on-demand based on 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.

This paper aims at making the shared bus interconnects found in current chip multiprocessors workload-adaptive. Although Network-on-Chip (NoC) has been proposed as a scalable solution for CMPs[1], the use of shared bus interconnect is likely to continue in the near future for chips with moderate number of cores. This is because design of Snoopy-based interconnection networks is simple and supporting traditional memory consistency mechanisms on snoopy-based interconnection network is well-understood. Directory-based schemes as used in NoC suffer from additional area overhead of directories and directory-management hardware which may be prohibitive for chips with small to moderate number of cores.

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.

II. Baseline Interconnection Network Architecture

A classical shared bus interconnect for a CMP using snoopy cache coherence[16] is shown in Fig.2[10]. The shared bus interconnect shown consists of multiple pipelined 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 hung from the snoop bus. The request travels to the end of the address bus from where it is stored in another queue. From there, the request travels across the snoop bus. All components connected to the snoop bus source the request and check to see if they could serve the request. Access to the Address bus is provided by the arbiter.

Each component connected to the snoop bus sources the request and generate an appropriate response after some time on the response bus. This involves looking up the tags to check for the requested data in case of caches. All responses travel to the end of response bus where a piece of logic

978-1-4244-4938-5/09/$25.00©2009 IEEE

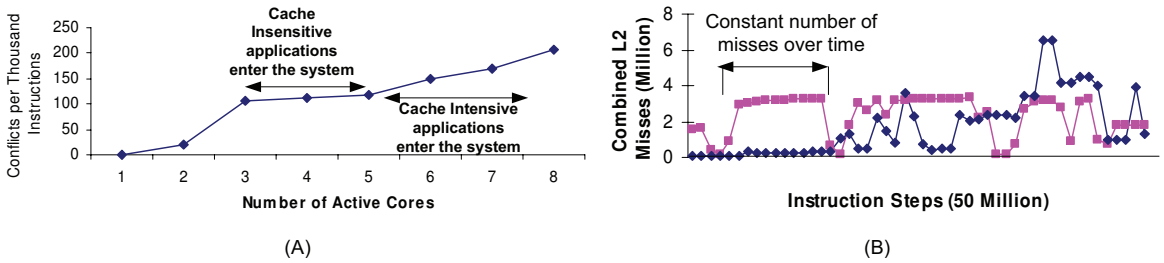

Fig. 1. Examples of Workload Interaction with Interconnection Network

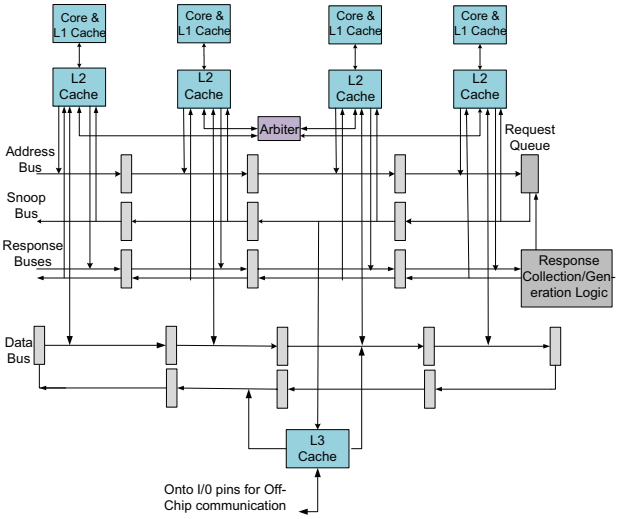

Fig. 2. Detailed Model of a Shared Bus Interconnect for Chip Multiprocessors

(book-keeping logic) collects the responses 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 etc.

Two possibilities exist for snooping the request from the shared bus. The first involves waiting for the responses to be collected by the book-keeping logic from the response bus and then sending the request to cache controller below if none of the private caches of other cores could service the request. Other approach is to snoop from somewhere along the snoop bus. The two approaches have a delay-power tradeoff.

From Fig.2, it could be seen that the overhead of a single transaction involves traversing the point-to-point links on different buses and the logic overhead of arbiters etc. It should be noted that not all cores need to take part in the coherence mechanism described above. In particular, an application which is part of a multiprogramming workload could send the request to L3 controller once L2 miss is detected. Similarly, requests generated by separate multi-threaded applications need not arrive at a common ordering point as is the case in Fig.2.

III. A Framework to Support Reconfigurable Interconnects

In this section, we will describe our modified architecture that relaxes the serialization constraint on bus-based interconnection network by clustering the cores into groups and localizing coherence traffic generated by an independent workload within a cluster.

A. Overview

The high-level view of the baseline multicore processor for eight cores is shown in Fig.3A. In the baseline architecture all cores are connected through a monolithic shared bus. There could be multiple L3 controllers serving different requests simultaneously destined for different banks. The proposed architecture for the multicore chip is shown in Fig.3B. In Fig.3B, 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, inter-cluster 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

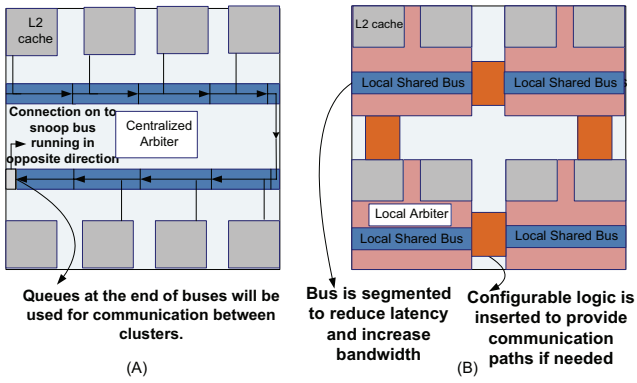

Fig. 3. A) Baseline Multicore Architecture versus; B) Clustered Multicore Architecture

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 our architecture, we assume a MOESI snoopy-based cache coherence protocol[16] within a cluster. We assume a weaker-consistency model for access to shared data across clusters. In particular, writes to data shared across clusters is protected by locks.

B. Reconfigurable Interconnection Logic

The reconfigurable logic between clusters has two goals. First, it provides on-demand isolation between clusters, thereby not having to do coherence across clusters. This will be particularly helpful for multiprogramming workloads as they could be scheduled within one cluster. Secondly, for multi-threaded workloads, based upon the expected communication patterns, it supports two different policies for communication among clusters.

- **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 book-keeping logic. If the probability of finding data in the other cluster is high, this policy reduces latency for the requesting cluster.
- **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 multi-threaded applications that communicate rarely.

In later 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 programmer regarding the expected communication pattern among different threads. Fig.4 illustrates the rationale behind the two policies supported by inter-cluster logic. The next two

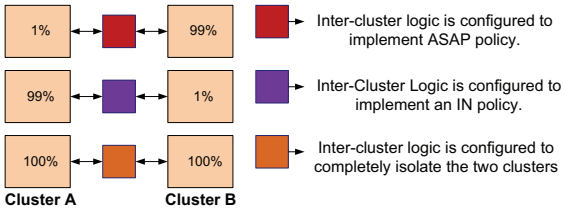

Fig. 4. Example Scenarios for Using the Two Policies for Communication between Clusters and Isolation Property of Clusters. In the first case of ASAP, Cluster A has a 1% chance of finding data locally and 99% chance of finding data in Cluster B.

sections provide an implementation of the reconfigurable logic between clusters with the goals described in this section.

C. Additional Hardware Components

First, we describe the additional logic components that will be utilized. We also provide the area and timing overhead of the additional components for $65nm$(TSMC) technology using Synopsys Design Vision.

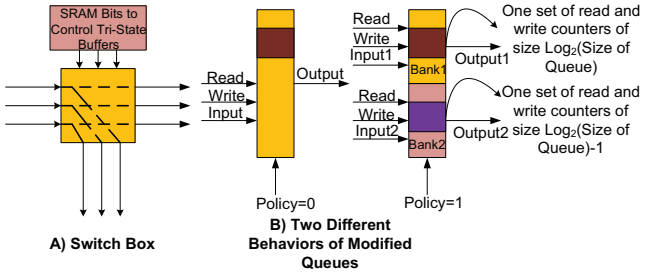

Fig. 5. Two Major Components Used for Reconfigurable Logic between Clusters

1) Switch Boxes: The switch box fabric used for routing purposes in Field Programmable Gate Arrays[14] is an important structure to utilize in on-chip reconfigurable interconnects. A Switch box, as shown in Fig.5A, 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 $Fs * W$, where W is the width of the bus. We used tri-state buffers as switches inside our switch block. The area-overhead of switch-box for bus width of 64 bits was found to be 430 square micro-meters.

2) Modified Queues: Queues are the primary mode of communication in our design as shown in Fig.3. 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 $Q/2$ take requests from the local cluster and another bank of size $Q/2$ buffers requests from the neighboring cluster. The two behaviors are depicted in Fig.5B. This simulation of polymorphic behavior by logic components maximizes their utility for reconfigurable systems. 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 Fig.5B. The area-overhead of the modified queue was estimated to be 20% more than that of the base queue.

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 multiplexor was found to be 200 square micro-meters for a bus width of 64 bits.

The area overhead of the above logic components required for inter-cluster logic is not prohibitive thus providing the designer with a 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 $65nm$, we chose a clock cycle of $1.5GHz$ for simulation results in Section.VI. All components passed the one cycle delay constraint for bus frequency of $1.5GHz$.

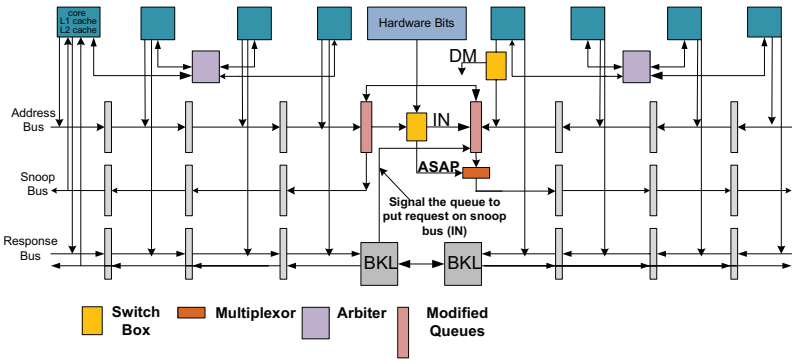

Fig. 6. Inter-Cluster Logic between Two Clusters and Direct Memory Connection Provided to One Core

D. Complete Design of Inter-Cluster Logic for Two Clusters

Fig.6 shows two clusters connected through the inter-cluster logic. Additional logic 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 (name 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 neighboring cluster through the path labeled ASAP in Fig.6, where a multiplexor puts 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 incoming request along path labeled IN in Fig.6 and the neighboring queue is signaled to store the request. The Book-Keeping Logic (BKL) of local cluster, only after collecting responses and finding out that request can not be serviced locally, signals the neighboring queue to issue the request on its snoop bus. The queues in this case are configured as multi-banked as was discussed in Fig.5B.

E. 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 L2 cache and the input port of L3 cache controller is useful in Fig.2. 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 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 DM in Fig.6 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.

IV. System Support

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

A. 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 programmer through annotations in the code. Following annotations are applicable to the architecture described in III.

- **Single Threaded** This is a single-threaded application.
- **Coarse Grained Sharing** This group of threads is part of a multi-threaded application and the threads have coarse-grained sharing among them.
- **Fine Grained Sharing** This group of threads is part of a multi-threaded application and the threads share data at a finer granularity.

The use of annotations to configure the underlying parameters through hardware bits is the same as in [4].

B. Operating System Support

The annotations are 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.III-C. Also, modern operating systems (OS) 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.

We first show how annotations can be used to select policies discussed in III-B and make good scheduling decisions by OS for multi-threaded applications. Fig.7A shows that for 4 threads (T1,T2,T3,T4) mapped to a 4-core CMP, six possible communicating thread pairs are possible. Fig.7B shows how

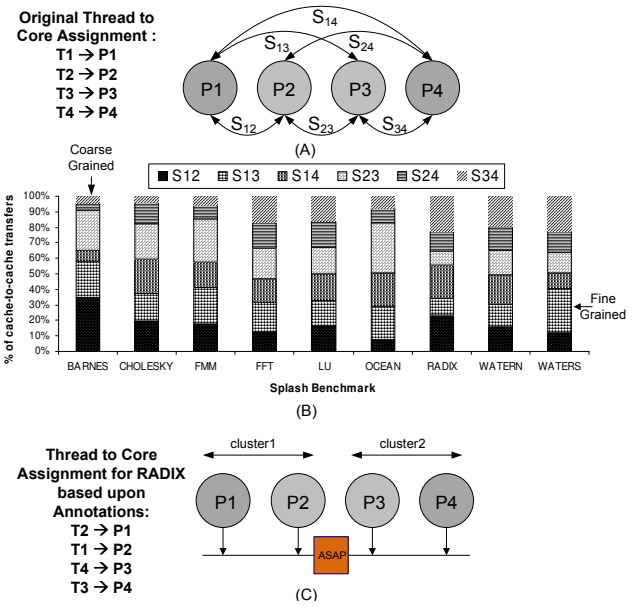

Fig. 7. A) Four Cores and Possible Communication Pairs; B) Example of Annotations in Splash Benchmarks Based upon Expected Communication Patterns among Thread Pairs; C) Resulting Scheduling Decisions and Policy Selection between Clusters

annotations could be applied based upon the expected sharing pattern among threads for Splash benchmarks[17][1]. The benchmarks were run under load-balanced conditions using the Linux kernel running on top of a multiprocessor simulator [2]. In Fig.7B, *Sxy* implies that, of the total cache-to-cache transfers that took place during the execution of application, *Sxy* took place between core x and y. Fig.7C shows the scheduling decisions and communication policy selected by the OS for RADIX benchmark for a 4-core CMP and 2 clusters. In order to understand the scheduling decisions and selected policy, note the values of *Sxy* for different values of x and y in case of RADIX. 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. This frequent communication between T1 and T4 is also the reason these threads are scheduled on P2 and P3 respectively.

The OS can also make good scheduling decisions for single-threaded applications and multiprogramming workloads. For instance, single-threaded applications forming part of the multiprogramming workload could be mapped to a single cluster. This will give the OS larger decision-space when a multi-threaded application enters the system. 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 Fig.2, latency of access to many resources (arbiters, queues) is location dependent. If an application communicates with the arbiter very often, scheduling it closer to the arbiter will increase its performance.

[1]Note that in Splash benchmarks, n processes each corresponding to one of the n processors are created at the beginning of application. The different processes co-operate to complete the job. Therefore, in this context, our use of term thread rather implies a "group of threads".

TABLE I
NODE PARAMETERS

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

V. METHODOLOGY

We evaluated our proposed architectural techniques using the M5 simulator[2]. We modified the M5 shared bus interconnect to implement separate address, snoop, response and data Buses as shown in Fig.2. 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 bi-directional 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 Fig.2) after arbitrating for it. For our modeled system, the L3 controller always snoops the request from the middle of 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 our experiments across four technology generations. The scaling of frequency of cores is taken from ITRS roadmap. The frequency of shared buses is assumed half of the core frequency. This co-relates with existing CMP architectures[15]. 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 $400mm^2$ 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.II. 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 rational 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 arbiter is not modeled. However, behavior of 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

TABLE II
CORE PARAMETERS

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

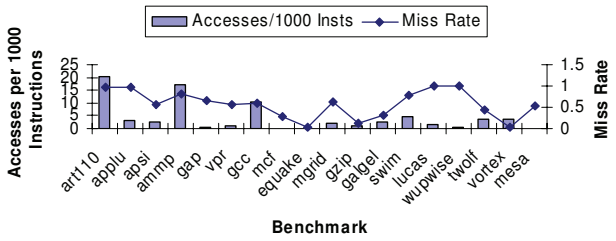

Fig. 8. Miss Rates and Accesses per Thousand Instructions for Considered SPEC Benchmarks

on the underlying cache miss rates of workloads, the miss-rates of considered SPEC benchmarks are shown in Fig.8. For the duration Lucas and Wupwise were run, L2 miss rate is very close to 1 because during this phase no reads takes place and all writes miss in the cache.

VI. RESULTS

In this section, we show the results for the improvement in performance of using the proposed multi-core 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.

- **DM** One core has been provided with direct connection to memory as shown in Fig.6.
- **CX** The cores are divided into X clusters. For instance, for $32nm$ 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.

A. Performance Analysis for Single Applications

Fig.9A shows the average latency incurred by L2 cache misses for various size 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. 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 could be seen in Fig.9A that the direct memory approach falls behind as we make clusters with two cores. This is because by providing a direct memory connection, we only gets rid of the latency incurred by address and snoop bus. 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 III-E. Fig.9B shows the results of performance improvement with this combined effect over the base case for all technology nodes. Performance gained is very high as we scale down the technology node and applications have high miss rates.

B. 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 from the benchmarks shown in Fig.8. The workloads were created to have a varied representation of L2 cache miss rates. In the following experiments, we initially run the workload on baseline processor. Subsequently, we run the workloads on clusters of finer granularity (less number of cores per cluster) and note the reduction in latency. Our results indicate that as we adapt the architecture to best serve the workload, overall performance is always improved.

Fig.9C shows that for a multi-core processor with four cores modeled after $90nm$ technology and running two applications, there is a moderate performance gain with clustering. Fig.9D shows the results for different levels of clustering for a CMP with sixteen cores($45nm$. Performance gains increase as we scale down because the relative delay of global wires increases and clustering has greater advantage. From the data collected for $90nm$ and $45nm$ with two applications, we observed that in some cases the CPI increases by as much as 33% over baseline architectures. This offsets the advantage of a 50% increase in core frequency as we scale down. Our proposed architecture soothes this impact of wire delays considerably. The performance results for four applications running on a 32 core multi-core processor are shown in Fig.9E. For $32nm$, we observed that the performance gains for some workloads are as high as 60%.

Note that in both Fig.9D and Fig.9E, the performance gained with C4 and C8 is almost equal. This is because in both cases, the number of cores in the cluster becomes equal to the number of threads. The impact of reduced delay is now offset by conflicts for the arbiter and shared bus since there are more requests per unit of time. This could be overcome my mapping multiprogramming workloads across clusters. This is also the reason why we only observed moderate gains for two applications in $90nm$ case as shown in Fig.9C.

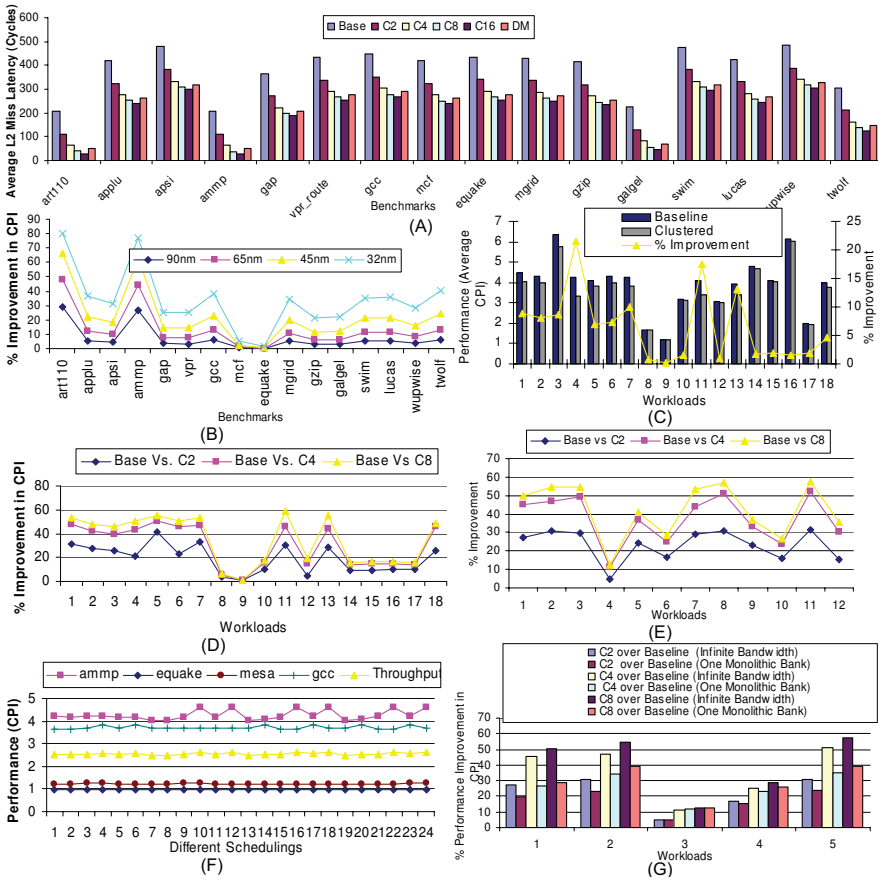

Fig. 9. Performance Improvement of Different Architectures Over Baseline; A)Average L2 Miss Latency for Different Architectures Running Single Application (32nm); B)Performance Improvement for Single Application Using a Cluster of 2 Cores and Direct Memory Connection to Shared L3 Cache; C)Performance Improvement for Different Architectures Running 2 Applications (90nm); D)Performance Improvement for different Architectures Running Two Applications (45nm); E)Performance Improvement for Different Architecture Running Four Applications (32nm); F)Impact of Scheduling on Performance within a Single Cluster; G)Impact of L3 Bandwidth Limitation on Performance Gained with Clustering

C. Performance Analysis of Scheduling within a Cluster

Fig.9F shows the impact of scheduling on the performance of a workload running on a single cluster with four cores. Different curves show the performance of individual applications and the overall throughput of the complete workload. The figure shows that we can achieve a 6% improvement in overall throughput and 12% improvement in the performance of the most cache-intensive benchmark (Ammp) by using interconnect-aware scheduling. We analyzed many results for scheduling 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 by scheduling described in this section 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 best spot is reserved (closest to arbiter etc) for the most memory-intensive application.

D. 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 correspondingly. While we considered the availability of a large number of banks in our results shown in Fig.9, in this section we will do some analysis of the dependence of our proposed techniques on the bandwidth of shared L3 cache below. 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 $36MB$. Hits are queued up while misses are not affected due to the presence of large number of MSHRs[9].

Fig.9G indicate that workloads that generate relatively low interconnect traffic (workloads 3 and 4) and hence having little potential for performance improvement with clustering are not affected by the L3 bandwidth. However, with limited bandwidth, workloads with high accesses to memory suffer significantly. We also analyzed the impact of performance loss due to the presence of no MSHRs in the L3 cache.

Needless to say, as we make clusters of finer granularity, the performance loss becomes extremely high. Our conclusion is that the interconnect optimizations proposed in this paper should be complemented by employing aggressive techniques to increase the bandwidth of underlying shared memory.

VII. Related Work

On the clustering side, Wilson[7] did the classical 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. Several works exist that reduce the broadcast-based traffic required to maintain coherence using filtering of snoop requests [3],[13]. Such techniques were initially proposed for web servers and they result in an area-overhead for maintaining extra information in directories. Our architecture does not prevent unnecessary broadcasts but reduces their impact on performance by localizing them. Further, techniques such as coarse-grained coherence tracking and snoop filtering can augment our architecture further. In [8], 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 [11]. In terms of on-chip interconnects, although our architecture does not have a global ordering point as is the case in conventional shared bus architectures, the inter-cluster logic sets-up circuit-switched paths to maintain the illusion of a shared bus. A similar technique of setting up circuit-switched routing paths at run-time is used by authors of [6] but for NoC-based interconnect and directory-based coherence scheme. The inter-cluster logic in our architecture is significantly less complex than a switch or router used in NoCs. The flow of requests in Fig.3B is similar to that in ring-based interconnects[15]. 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[12].

In general, no literature exists on augmenting the prevalent shared bus interconnects for CMPs by using programmable interconnects.

VIII. Summary and Conclusions

In this paper, we presented design techniques to improve the performance of shared bus interconnects for multi-core processors. The proposed techniques configure the interconnect in accordance with the workload at run-time and require minimal system-level support. We presented results for reduction in CPI of multiprogramming workloads with various levels of clustering. We also provided some optimizations for single-threaded performance. Our results show that significant performance benefits are possible when the interconnection is made workload-adaptive. As the amount of heterogeneity in workloads (and processors) increases, the benefits of our approach are only going to increase.

Acknowledgements

This work is partially supported by the NSF CCF 07-02501 grant. We used machines donated by Intel. We also would like to thank Mr. Lu Wan of the ECE department of University of Illinois-Urbana Champaign for helpful discussions.

References

[1] James Balfour and William J. Dally. Design tradeoffs for tiled cmp on-chip networks. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 187–198, New York, NY, USA, 2006. ACM.

[2] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, July-Aug 2006.

[3] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 246–257, Washington, DC, USA, 2005. IEEE Computer Society.

[4] John B. Carter, Contact John, and B. Carter. Design of the munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 29, 1995.

[5] E. Duesterwald, C. Cascaval, and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. pages 220–231, Sept.-1 Oct. 2003.

[6] Noel Eisley, Li-Shiuan Peh, and Li Shang. In-network cache coherence. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 321–332, Washington, DC, USA, 2006. IEEE Computer Society.

[7] Andrew W. Wilson Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the 14th annual international symposium on Computer architecture*, 1987.

[8] M.M. Kim, J.D. Davis, M. Oskin, and T. Austin. Polymorphic on-chip networks. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 101–112, June 2008.

[9] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 81–87, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[10] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Proceedings of the 32nd International Symposium on Computer Architecture*, June 2005.

[11] Lizy Kurian and Eugene John. A dynamically reconfigurable interconnect for array processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(1):150–157, March 1998.

[12] Michael R. Marty and Mark D. Hill. Coherence ordering for ring-based chip multiprocessors. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 309–320, Washington, DC, USA, 2006. IEEE Computer Society.

[13] Andreas Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. *SIGARCH Comput. Archit. News*, 33(2):234–245, 2005.

[14] J. Rose and S. Brown. Flexibility of interconnection structures for field-programmable gate arrays. *Solid-State Circuits, IEEE Journal of*, 26(3):277–282, Mar 1991.

[15] B. Sinharoy, R. N. Kalla, J. M. Tendler, and R. J. Eickemeyer. Power5 system microarchitecture. *IBM J. RES. and DEV.*, 49(4/5):505–521, July/September 2005.

[16] Per Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990.

[17] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.