Thread Criticality Support in On-Chip Networks

Yuho Jin, Ruisheng Wang, Woojin Choi, Timothy Mark Pinkston
University of Southern California
3740 McClintock Avenue
Los Angeles, California
{yujin, ruishenw, woojinch, tpink}@usc.edu

ABSTRACT

Multicore computing is becoming the mainstream approach in computer system designs to effectively use growing transistor budgets for harnessing performance and energy-efficiency. Increasing the parallelism with more cores requires careful management, allocation, or partitioning of shared resources to cope with varying resource demands from running threads. Predicting critical (or slowest) threads and accelerating execution of those threads can reduce execution time of parallel applications by balancing the execution of threads to synchronization points. The on-chip network is an increasingly important component that services communication of threads running on cores. As the communication latency of threads affects thread criticality, it should be considered and optimized. In this work, we explore thread criticality support in on-chip networks. We propose a flow control technique that reserves router resources to accelerate communication from critical threads. Furthermore, we present thread criticality support in arbiter designs. Our evaluation shows that implementing criticality awareness in an on-chip interconnect design reduces execution time by 22% and increases system throughput by 18% for a 64-core processor.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Performance

Keywords

On-Chip Network, Multicore, Thread Criticality

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
NoCArc '10, December 4, 2010, Atlanta, Georgia, USA
Copyright 2010 ACM 978-1-4503-0397-2 ...$10.00.

1. INTRODUCTION

To leverage performance benefits from many available processor cores on a chip, future applications will be highly parallelized through diverse programming techniques. Any load imbalance on hardware resources across executing threads, however, limits parallelism and causes performance loss and energy inefficiency. Careful hardware resource management optimized to different resource demands of threads is therefore needed to provide continuous performance improvement in multicores.

The on-chip network (OCN) or Network-on-Chip (NoC) is widely accepted as a solution for communication architecture in large-scale integration of on-chip components. In a multicore system, the OCN is the subsystem that provides connectivity among processing units, storage elements, and off-chip interfaces for efficiently transporting data in minimal time, high volume, and least energy. The OCN is expected as a critical resource shared by other subsystems [6, 7, 11] and, hence, the design of the OCN significantly impacts system throughput and application execution time. The OCN is also popularly adopted as the backbone in building the shared memory architecture of multicore processors.

Previous studies [3, 5, 12] show that threads in parallel applications executing on multicores have different stall times when measuring arrival times at barriers, which are meeting points for all threads. The main reason for different execution times of threads is that each thread has a different demand on shared on-chip resources such as interconnect bandwidth, cache capacity, and memory controller bandwidth. Different amounts of resource demands and usages in each thread's execution result in different execution times of threads. As the slowest thread, i.e., *critical thread*, determines execution time of parallel applications, predicting thread criticality and accelerating the execution of critical threads can reduce overall application execution time. In recent work, one of the sources for slowing down progress of thread execution is accumulated cache miss penalty [3]. A thread criticality predictor based on cache miss penalty is used for designing a load-balanced task scheduler and dynamic voltage frequency scaling.

In this work, we exploit thread criticality support in OCN design by flow control and priority-based arbitration techniques. As the OCN connects many L2 cache banks for NUCA (Non-Uniform Cache Architecture) [9], network latency contributes a large portion of L2 cache access latency. Reducing network latency for communications from critical threads can help to reduce their miss penalty, thereby accelerating execution of critical threads. To achieve this goal,

we propose a bypass flow control technique through a state-preserving crossbar switch. The switch maintains state of the default output port for each input port. By dedicating one virtual channel (VC) in each physical channel to use a preserved state, regular pipeline stages that packets go through at each router can be avoided. Each router dynamically finds the best preserving state that increases router bypass chances for communications from critical packets. We also explore priority-based arbiters in routers, where the priority of packets conforms to the criticality of requesting threads. This type of arbitration reduces stall time of packets from critical threads in the case of router resource contention and, thus, leads to reduced cache miss latency for critical threads.

Simulation results with a 64-core system show that application execution time and system throughput are, on average, improved by 22% (up to 102%) and 18% (up to 73%), respectively. Furthermore, thread criticality support in the on-chip network achieves, on average, packet latency reduction of 14% and network power reduction of 10%.

The rest of this paper is organized as follows. Motivation for this work from characterization of thread criticality is presented in Section 2. Thread criticality support through bypass flow control and priority-based arbitration is described in Section 3. The evaluation methodology is presented in Section 4, and simulation results of the proposed techniques are presented in Section 5. Finally, related work is noted in Section 6, and Section 7 concludes the paper.

2. CHARACTERIZATION OF THREAD EXECUTION BEHAVIOR

Though applications are highly parallelized to spread computation work across processor cores, execution can be imbalanced among cores due to different demands on shared resources such as cache capacity, on-chip interconnect bandwidth, and off-chip bandwidth. One of the sources of load imbalance which varies the execution speed of threads is different amounts of cache misses and associated cache miss penalty [3].

Table 1 lists characteristics of the slowest thread relative to those of the fastest thread among all threads from the PARSEC benchmark [4] running on an OCN-based 64-core system. The slowest thread shows a significantly larger number of accesses to L2 cache and off-chip memory than the fastest thread. The slowest thread injects a much larger number of packets for cache operations and consumes more on/off-chip bandwidth than the fastest thread. On average, the slowest thread has 17.7 times larger miss penalty than the fastest thread. This huge difference causes different execution speeds of threads.

As the L2 cache is organized as a multi-bank design with an on-chip interconnection network for technology scaling problems [9], interconnect latency becomes dominant over the access latency of cache storage. Indeed, in an 8×8 mesh-based multicore processor with 3-cycle hop latency, 6-cycle access latency for a 4KB bank contributes 14% of the total access time, but the round trip of a single-flit packet from core to cache bank takes 86% (36 cycles) of the total access time (42 cycles) for 12 hops under uniform traffic. Thus, determining the criticality of a thread based on its accumulated miss penalty should reflect the effect of communication latency in the interconnect.

Table 1: Performance discrepancy across threads in the 64-core system. Numbers are shown as a ratio of slowest thread to fastest thread.

benchmark	cycles	L1 misses	L2 hits	L2 misses	miss penalty
blackscholes	4.0	4.8	4.7	6.8	3.9
bodytrack	14.7	8.6	10.0	4.9	16.3
canneal	98.1	70.2	90.0	35.9	104.6
facesim	10.4	9.6	10.8	7.0	10.1
ferret	6.7	11.6	9.3	16.3	6.6
fluidanimate	3.5	3.0	3.3	2.7	3.6
raytrace	23.1	11.6	16.2	6.1	24.6
streamcluster	4.7	4.9	5.9	2.8	5.5
swaptions	7.7	6.9	7.6	5.9	7.9
vips	5.9	6.4	7.0	3.6	5.9
x264	7.0	10.0	10.1	9.1	6.1
average	16.9	13.4	15.9	9.2	17.7

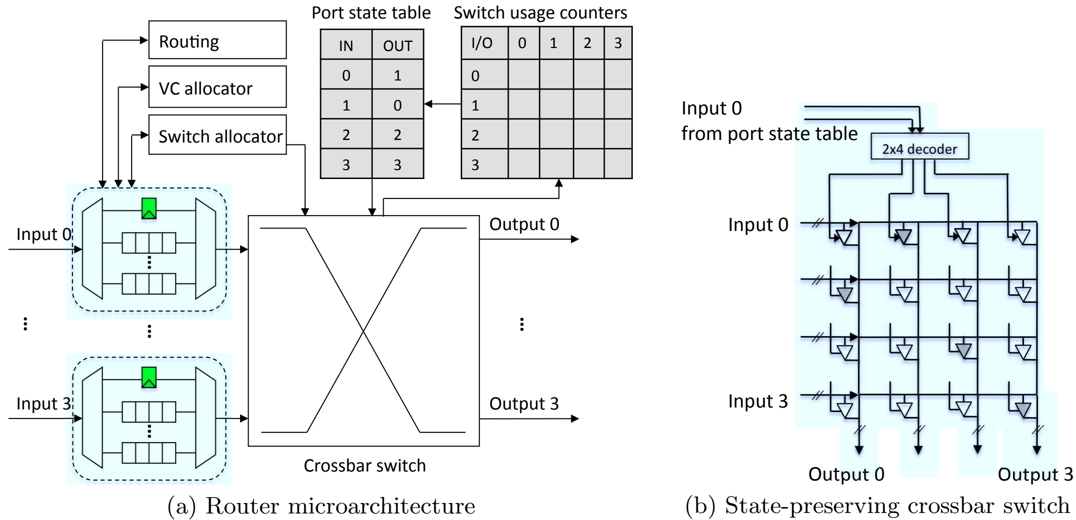
3. THREAD CRITICALITY SUPPORT IN NETWORK ARCHITECTURE

3.1 Conventional Virtual Channel Flow Control

We first describe building blocks of conventional 2-stage pipelined virtual-channel router architecture and present the motivation of embedding bypass flow control in existing routers to achieve low latency. Flits (flow control units) of a packet are stored in a buffer divided into multiple VCs. The first stage of a router consists of VC allocation (VA) and switch allocation (SA). VA operation is performed for a head flit to reserve a free output VC. SA does arbitration for switch input and output ports. Furthermore, SA and VA can occur during the same cycle [15]. The second stage is switch traversal (ST), where a flit traverses the corresponding input port and output port allocated during the SA stage. Routing computation (RC) is performed at a previous upstream router (e.g., lookahead routing) so that RC does not require a separate stage. Finally, after a flit ejects from a router, it traverses a link (LT) and enters into a neighboring downstream router. In this scenario, each flit experiences a total 3 cycles per hop, which is broken down to 2 cycles per router and 1 cycle per link.

The crossbar switch is an important component for achieving high throughput in router designs. Multiple flits can traverse the switch at the same time up to a maximum number of ports. When port contention occurs, arbitration performed in the SA stage resolves port conflicts. Input port arbitration resolves conflicts among packets from different input VCs within the same input port, while output port arbitration does so among packets from different input ports but to the same output port.

Channels are statically divided into multiple VCs. Each input VC is associated with one FIFO buffer implemented as registers or memories to hold packets when packets cannot be forwarded to the proper resource due to conflicts with other packets. The use of VCs can be extended to avoid deadlocks generated by cyclic channel dependence from adaptive routing algorithms or cache coherence protocols. The VA stage allocates one output VC to a requesting packet in one input VC.



(a) Router microarchitecture (b) State-preserving crossbar switch

Figure 1: Router design to bypass pipeline stages.

If a router reserves port and VC resources at the same time of packet arrival, the first stage required for SA and VA can be avoided for packet transportation. Likewise, if a switch maintains an internal path for the requested input port and output port for a packet, the second stage for ST can be integrated with LT as a single cycle.

3.2 Bypass Flow Control

As mentioned previously, a VC wormhole router has multiple pipeline stages to achieve high throughput. However, multiple stages increase per-hop latency by imposing one cycle per stage, which increases communication latency proportionally to hop count. This high-throughput feature can be overridden by reserving router resources in accordance with application characteristics.

Here we propose router bypass flow control, which reduces router latency to zero when a packet follows the preallocated path. The basic idea is to reserve network resources for proper switching patterns that accelerate communication for critical threads. When a flit follows the reserved input and output ports, it can directly enter into the LT stage by skipping all stages of the router pipeline.

For this objective, a router preserves a state that defines one preferred output port for each input port without conflicts. One VC is dedicated to each port for router bypassing. We call this VC a *bypass* VC in contrast to a regular VC. Through this mechanism, packets that use those input and output ports in a preserving state can traverse a router without switch allocation and VC allocation. We discuss how to find a preserving state in Section 3.3.

Figure 1a shows the router design for bypass flow control. The input buffer is split into one *bypass* VC and multiple regular VCs. The *bypass* VC is associated with the latch. When a router is not able to use ports in a preserving state, a packet allocated to a *bypass* VC is stored in this buffer. The port state table is updated periodically based on the best switching pattern for critical threads. Switch usage counters track switching patterns for packet traversal in a router. Due to lookahead routing, an upstream router propagates the packet's route to a downstream router to increment the corresponding counter for an input port and an output port.

Figure 1b shows our design of the state-preserving switch crossbar used for bypass flow control. Connection to one output port for each input port is controlled by a decoder. The input signal to this decoder comes from the port state table. Figure 1b shows a preserving state given by the port state table in Figure 1a.

As an upstream router makes a bypassing decision for a downstream router, a preserving state in a downstream router is forwarded to each upstream router. Because route information for a downstream router is available at an upstream router owing to lookahead routing, an upstream router can prevent allocating a *bypass* VC for a packet that does not match with a preserving state in a downstream router. Moreover, the status of an output bypass VC at an upstream router prevents the use of that output VC for another packet, which is cost-free because this function already exists in a conventional router design.

As a *bypass* VC and regular VCs share the same physical channel, conflicts on the same port (i.e, the same physical channel) can occur. In this case, packets on regular VCs always have higher priority than packets on *bypass* VC. This means that though packets are assigned to skip pipeline stages through a *bypass* VC at an upstream router, packets can experience regular pipeline stages at a downstream router. Similarly as with cut-through flow control, when one flit in the same packet cannot bypass a router, all the following flits also cannot bypass a router. This guarantees correct transport of all flits of the same packet for VC flow control and obviates extra hardware of reordering flits at destinations. A packet cannot bypass a router through the *bypass* VC if switch allocation occurs during the previous cycle. This means that there are waiting packets for regular VCs or switch ports different from ones in a preserving state. Conservatively using shared physical channels for a *bypass* VC provides no starvation for regular VCs. In addition, router bypassing is not allowed if an output VC has no credits for the input buffer at a downstream router. This condition is necessary, because a packet that is transported through a *bypass* VC can be stored in the buffer (i.e., transported through router pipelines). Compared to EVC flow control [10], this mechanism does not have overheads for

credit management to check buffer availability in routers far from the current router.

3.3 Switch Usage Counter

One important decision in our router design is how to find a preserving state that a crossbar will maintain. Finding states to reduce communication latency of packets from as many critical threads as possible will reduce miss penalty significantly. Here, we assume that a packet has a criticality indicator encoded as priority in the header. When a high-priority packet traverses a router, one counter for the corresponding input and output ports increases. When a change of the preserving state is triggered, all the values stored in counters are scanned in increasing order to find the best output port for each input port. A port state table is updated with the found mapping result between input and output ports to provide a preserving state for the next interval. Counters should be selectively updated only for critical threads to reduce their execution time as much as possible.

3.4 Priority-Based Arbiter

When a packet experiences regular pipeline stages, it requires VC arbitration, switch arbitration, or speculative switch arbitration. Arbiter design provides another opportunity to deliver packets from critical threads with high priorities over those from non-critical threads. Thus packets from critical threads experience less stall time from arbitration operations. Packet priority is assigned consistently using the thread criticality indicator, meaning that threads causing large miss penalty have high priorities. To provide a fair chance to all waiting packets, we increase the priority by one for packets that are not chosen for arbitration outcomes. This increment of priority is confined in each arbitration unit and does not modify a packet's original priority of thread criticality.

3.5 Thread Criticality Predictor

We modify a previously proposed thread criticality predictor [3], assuming a monolithic L2 cache. It accounts for number of per-core L2 hits and L2 misses to estimate accumulated L1 miss penalty during a fixed interval. However, as a L2 cache consists of multiple banks connected by a mesh network, a distribution of cache blocks for each thread significantly affects miss penalty. Therefore, in our design, we consider a round-trip distance from each core to each bank for L2 hits or to each memory controller for L2 misses for accurate estimation. To estimate the sum of miss penalties for L2 hits, each bank of L2 cache has per-core hit counters. Per-core miss counters are not enough to estimate the sum of L2 miss penalties for L2 misses, because different network latency between L2 cache banks and memory controllers affects L2 miss latency. To precisely estimate this latency occurring only in L2 misses, per-core miss counters are further extended to per-core and per-memory-controller miss counters. As those distances are determined at design time, a preset value regarding network distance for each counter is multiplied to the counter value. In our 64-core configuration, each L2 bank has 64×1 hit counters and 64×4 miss counters for 4 memory controllers. Therefore, a total of $64 \times 64 \times 5$ counters are required in a 64-core system with 64 L2 banks. At the completion of each interval, all those values are forwarded through control packets to a node located in the center of the network for criticality prediction. Moreover, a central node needs to forward prediction results to all routers. To prevent congestion on a central node, we assume global coordination is achieved by a sequential visit of all nodes through a single control packet.

4. METHODOLOGY

Table 2: Parameters used in 64-core system.

Parameter	Value
Private L1 I&D caches	32KB, 4-way, 1 cycle, LRU
Shared L2 cache	16MB, 16-way, 6 cycles, LRU
Memory controllers	4
Cache block size	64B
Memory latency	256 cycles
Network topology	8×8 mesh
Link bandwidth	16B/cycle
Virtual channel	4 per protocol class, 3 classes
Input buffer	4-flit depth

Our simulation infrastructure is based on SIMICS [13] full system simulator with GEMS [14] and GARNET [1]. SIMICS [13] is configured to model a 64-core chip running Solaris 10 with UltraSPARCIII+. Each core has a 3GHz in-order execution unit, 32KB instruction/data L1 caches, and a slice of shared 256KB L2 cache totaling 16MB capacity. A mesh network connects 64 homogeneously structured nodes. Coherence is managed by MOESI protocol. The chip has four memory controllers to service off-chip traffic for misses in L2 cache and each controller is connected to a different router in a network. GARNET models the 4-stage (RC, VA, SA, and ST) pipelined wormhole router, which amounts to no-load latency of 4 cycles for head flits and 2 cycles for middle/tail flits. We have modified this original 4-stage router model to the 2-stage router model by implementing speculative switch allocation and lookahead routing. Therefore, the baseline router can transport all types of flits with 2 cycles. Link is set as 16B width, $2.2mm$ length, and 1-cycle traversal delay. A dimension-ordered routing algorithm determines packet routes in a mesh network. Orion [16] is used to measure impacts of different designs on network power consumption. Energy consumption of network components modeled with default $100nm$ technology in Orion is scaled to $32nm$ technology. The main system parameters are listed in Table 2. In this work, we run a suite of PARSEC benchmarks including emerging multithreaded applications in recognition, mining, and synthesis (RMS) compiled with pthread programming model.

5. PERFORMANCE EVALUATION

For thread criticality support in an on-chip network, thread criticality is encoded as priority in the packet header. In this work, 64 priority levels are defined to differentiate threads running on each core. Each level can be encoded as 6 bits (5% of total flit data) included in the header. To adapt runtime variation of thread criticality, switch usage counters are incremented for packets from only the top four critical threads (6.25% of all threads). We use 100K cycles as the interval size of collecting those counter values for thread criticality prediction.

We compare three designs for performance evaluation. The baseline has no support for thread criticality. The sec-

ond design uses bypass flow control and round-robin arbitration routers. The third design uses bypass flow control and priority-based arbitration routers.

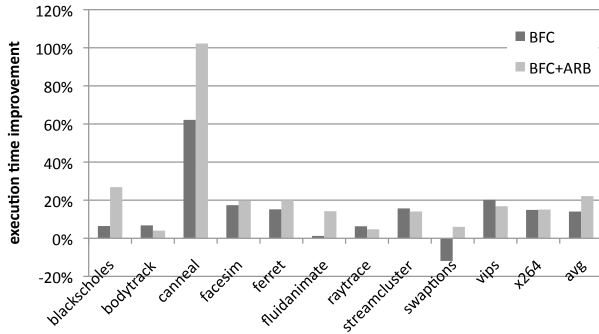


Figure 2: Application Execution Time.

Figure 2 shows the percentage of execution time improvement over the baseline design in each benchmark. Bypass flow control and priority-based arbitration for thread criticality support are indicated by **BFC** and **ARB**, respectively. Overall, on-chip network support for thread criticality improves execution time by 22% on average (up to 102%), as critical threads can run faster due to the reduced communication delay. The bypass flow control technique solely contributes 63% of execution time improvement. Small performance improvement from priority-based arbitration is due to low network load, leading to little contention in resource arbitration.

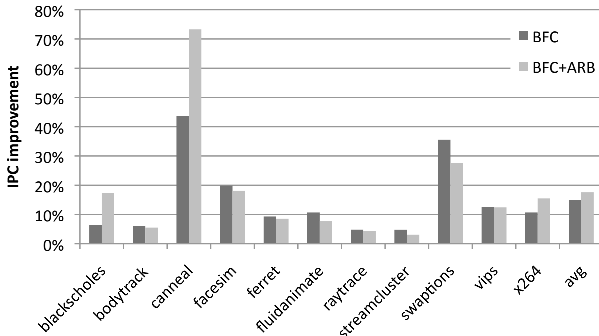


Figure 3: System Throughput.

Figure 3 shows the percentage of system throughput improvement measured by average of instructions per cycle (IPC) for each core. If critical threads run faster, IPCs on cores executing critical threads can increase. Speeding up the execution of critical threads enables faster transactions on cache blocks shared with other non-critical threads, which also improves IPCs of cores executing non-critical threads. This type of benefit would be more pronounced when parallel applications exhibit a high degree of cooperation with more communications across threads. Criticality support with our two techniques improves 18% of system throughput on average, which results in better utilization of system resources.

We further analyze the impact of thread criticality support on network performance. Figure 4 shows average packet latency representing the difference between injection and

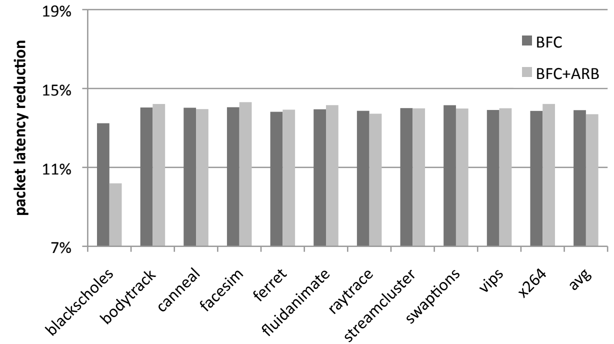


Figure 4: Packet Latency.

ejection times of packets. Bypass flow control obviously helps to reduce packet latency by 14% on average compared to the baseline design. Our objective is to reduce packet latency in miss penalty for only critical threads that determine application execution time, which can increase packet latency for non-critical threads. As packet latency is reduced by bypass flow control, the network can service more packets from more cache transactions for the same amount of time. Hence, bypass flow control improves network throughput (measured by number of packets per cycle) by 10% on average.

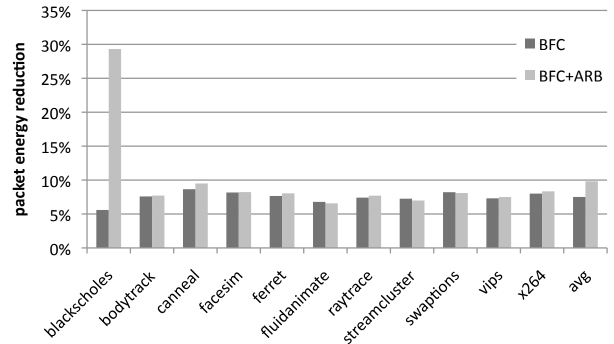


Figure 5: Packet Energy Consumption.

Another benefit from bypass flow control is power savings as buffer read/write and arbitration operations are reduced with router bypassing. In particular, as VC buffers are main power/area consumers in on-chip networks, reducing circuit activities in buffers can save a large portion of network power. We report average energy reduction for delivering a packet in Figure 5. Bypass flow control with round-robin arbitration saves packet energy by 4% on average compared to the baseline, while bypass flow control with priority-based arbitration saves energy by 9%. With respect to power consumption, we observe that some benchmarks such as *bodytrack*, *fluidanimate*, *raytrace*, and *swaptions* consume more network power than the baseline design. Examination of those benchmarks shows that fast execution of critical threads increases coherence messages to handle increased transitions of cache blocks per unit of time.

6. RELATED WORK

There has been prior work that focuses on exploring service management in on-chip networks [6, 7, 11]. In [11], a

frame amortizing usage of bandwidth in a large time window is used to effectively share on-chip network bandwidth. In [6], stall time criticality of packets is used to improve system throughput through priority-based arbitration across single-threaded applications. All three studies work well for shared bandwidth management under high contention, but have nominal improvement under low contention as providing differentiated service to each application or flow is difficult.

Flow control is explored to shorten long latency incurred by packet switching in conventional router pipelines [2, 8, 10]. In [10], the idea of express virtual channels is proposed to allow packets to bypass intermediate routers in the same dimension of a mesh network. In [8], hybrid circuit switching is implemented to improve performance of cache coherence protocols by skipping router pipeline stages through an established circuit as shared cache blocks are frequently accessed. In [2], a pseudo-circuit is proposed to find a more highly utilized bypass path within a router, while an original circuit sets up an end-to-end path. Although these studies lead to higher network performance, improving application execution time without exploring its thread behavior can be limited.

7. CONCLUSION

As on-chip networks become ubiquitous as the communication substrate of multicore systems, it is crucial for management and allocation of network resources to be able to flexibly adapt to the behavior of application execution. In a parallel application domain where threads have different execution speeds, the slowest executing thread determines execution time of the application. Therefore, accelerating the execution of critical threads by preserving access to required resources improves application performance. In this work, we explore thread criticality support in an on-chip network design to reduce the communication cost of critical threads. To achieve this objective, we use two techniques. Bypass flow control allows router pipeline stages to be skipped by using a state-preserving crossbar switch that maintains frequent routes between input and output ports for packets produced by critical threads. Priority-based arbitration for VCs and switches enables reduction of stall time for those packets under resource contention. Results using a simulated 64-core system show that criticality support in an on-chip network improves application execution time, system throughput, and network energy savings.

8. ACKNOWLEDGMENTS

The research described in this paper was supported, in part, by the National Science Foundation (NSF) grants CCF-0541417, CCF-0946388, and a CIFellows Postdoc Award.

9. REFERENCES

[1] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A detailed on-chip network model inside a full-system simulator. In *Proceedings of ISPASS*, pages 33–42, 2009.

[2] M. Ahn and E. J. Kim. Pseudo-Circuit: Accelerating Communication for On-Chip Interconnection Networks. In *Proceedings of MICRO*, 2010.

[3] A. Bhattacharjee and M. Martonosi. Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors. In *Proceedings of ISCA*, pages 290–301, 2009.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of PACT*, October 2008.

[5] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *Proceedings of PACT*, pages 240–249, 2008.

[6] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das. Application-aware prioritization mechanisms for on-chip networks. In *Proceedings of MICRO*, pages 280–291, 2009.

[7] B. Grot, S. W. Keckler, and O. Mutlu. Preemptive virtual clock: a flexible, efficient, and cost-effective QOS scheme for networks-on-chip. In *Proceedings of MICRO*, pages 268–279, 2009.

[8] N. D. E. Jerger, L.-S. Peh, and M. H. Lipasti. Circuit-switched coherence. In *Proceedings of NOCS*, pages 193–202, 2008.

[9] C. Kim, D. Burger, and S. W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *Proceedings of ASPLOS*, pages 211–222, 2002.

[10] A. Kumar, L.-S. Peh, P. Kundu, and N. K. Jha. Express Virtual Channels: Towards the Ideal Interconnection Fabric. In *Proceedings of ISCA*, pages 150–161, 2007.

[11] J. W. Lee, M. C. Ng, and K. Asanovic. Globally-Synchronized Frames for Guaranteed Quality-of-Service in On-Chip Networks. In *Proceedings of ISCA*, pages 89–100, 2008.

[12] J. Li, J. F. Martínez, and M. C. Huang. The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors. In *Proceedings of HPCA*, pages 14–23, 2004.

[13] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, 2002.

[14] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 33(4):92–99, 2005.

[15] L.-S. Peh and W. J. Dally. A Delay Model and Speculative Architecture for Pipelined Routers. In *Proceedings of HPCA*, pages 255–266, 2001.

[16] H. Wang, X. Zhu, L.-S. Peh, and S. Malik. Orion: a Power-Performance Simulator for Interconnection Networks. In *Proceedings of MICRO*, pages 294–305, 2002.