Darsim: A Parallel Cycle-Level NoC Simulator

DARSIM: a parallel cycle-level NoC simulator

Mieszko Lis* Keun Sup Shim* Myong Hyon Cho* Pengju Ren† Omer Khan* Srinivas Devadas*
*Massachusetts Institute of Technology, Cambridge, MA, USA †Xi'an Jiaotong University, Xi'an, China

***Abstract*—We present DARSIM, a parallel, highly configurable, cycle-level network-on-chip simulator based on an ingress-queued wormhole router architecture. The parallel simulation engine offers cycle-accurate as well as periodic synchronization, permitting tradeoffs between perfect accuracy and high speed with very good accuracy. When run on four separate physical cores, speedups can exceed a factor of 3.5, while when eight threads are mapped to the same cores via hyperthreading, simulation speeds up as much as five-fold.**

Most hardware parameters are configurable, including geometry, bandwidth, crossbar dimensions, and pipeline depths. A highly parametrized table-based design allows a variety of routing and virtual channel allocation algorithms out of the box, ranging from simple DOR routing to complex Valiant, ROMM, or PROM schemes, BSOR, and adaptive routing. DARSIM can run in network-only mode using traces or directly emulate a MIPS-based multicore.

Sources are freely available under the open-source MIT license.

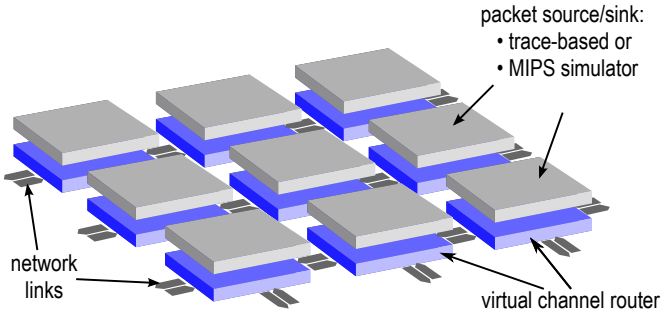


Fig. 1. A network-on-chip system simulated by DARSIM. The gray tiles (top) can be trace-driven packet injectors or cycle-level MIPS core models; the blue tiles (bottom) are cycle-level models of a flit-based virtual-channel wormhole router. While the illustration shows a 2D mesh, DARSIM can construct a system with any interconnect geometry.

I. INTRODUCTION

In the recent years, architectures with several distinct CPU cores on a single die have become the standard: general-purpose processors now include as many as eight cores [1] and multicore designs with 64 or more cores are commercially available [2]. Experts predict that by the end of the decade we could have as many as 1000 cores on a single die [3].

For a multicore on this massive scale, connectivity is a major concern. Current interconnects like buses, all-to-all point-to-point connections, and even rings clearly do not scale beyond a few cores. The relatively small scale of existing network-on-chip (NoC) designs has allowed plentiful on-chip bandwidth to make up for simple routing [4], but this will not last as scales grow from the 8×8 mesh of a 64-core chip to the 32×32 dimensions of a 1000-core: assuming all-to-all traffic and one flow per source/destination pair, a link in an 8×8 mesh with XY routing carries at most 128 flows, but in a 32×32 mesh, the worst link could be on the critical path of as many as 8,192 flows.

Future multicores will, therefore, require a relatively high-performance network and sophisticated routing. In such complex systems, complex interactions make real-world performance difficult to intuit, and designers have long relied on cycle-level simulations to guide algorithmic and architectural decisions; NoCs are no different. On a multicore scale, however, a cycle-level system simulator has high computation requirements, and taking advantage of the parallel execution capabilities of today's systems is critical.

With this in mind, we present DARSIM, a highly configurable, cycle-level network-on-chip simulator with support for a variety of routing and VC allocation algorithms. Its multithreaded simulation engine divides the work equally among available processor cores, and permits either cycle-accurate precision or increased performance at some accuracy cost via periodic synchronization. DARSIM can be driven by synthetic patterns or application traces or by a built-in MIPS simulator.

In the remainder of the manuscript, we first outline the design and features of DARSIM in section II. Next, in section III, we review the capabilities of DARSIM and discuss speed vs. accuracy tradeoffs using complete runs of selected SPLASH-2 applications [5] as well as simulations using synthetic traffic patterns. Finally, we review related research in section IV and offer concluding remarks in section V.

II. DESIGN AND FEATURES

In this section, we outline the range of systems that can be simulated by DARSIM, and discuss the techniques used to parallelize simulations.

A. Traffic generation

Figure 1 shows the system simulated by DARSIM. Each tile contains a flit-based NoC router, connected to other routers via point-to-point links with any desired interconnect geometry, and, optionally, one of several possible traffic generators. These can be either a trace-driven injector or a cycle-level MIPS simulator; a common bridge abstraction presents a simple packet-based interface to the injectors and cores, hiding the details of DMA transfers and dividing the packets into flits and facilitating the development of new core types.

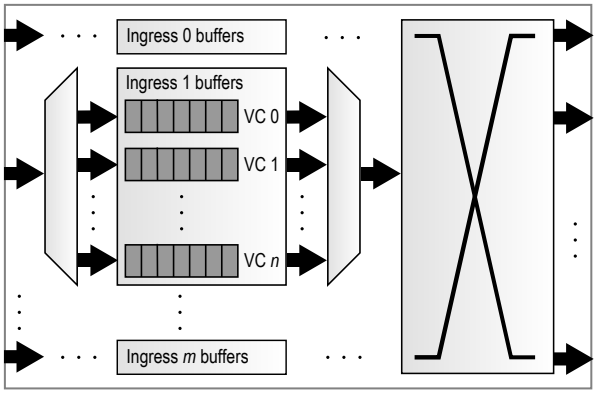


Fig. 2. Basic datapath of an NoC router modeled by DARSIM. Packets arrive flit-by-flit on ingress ports and are buffered in ingress virtual channel (VC) buffers until they have been assigned a next-hop node and VC; they then compete for the crossbar and, after crossing, depart from the egress ports.

1) Trace-driven injector: The simple trace-driven injector reads a text-format trace of the injection events: each event contains a timestamp, the flow ID, packet size, and possibly a repeat frequency (for periodic flows). The injector offers packets to the network at the appropriate times, buffering packets in an injector queue if the network cannot accept them and attempting retransmission until the packets are injected. When packets reach their destinations they are immediately discarded.

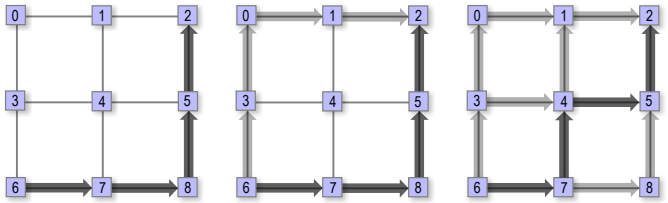
2) MIPS simulator: Each tile can be configured to simulate a built-in MIPS core with a private memory at cycle level; the simulator models a single-cycle in-order MIPS architecture and can be loaded with a statically linked binary compiled with a MIPS cross-compiler such as GCC.

To directly support MPI-style applications, the network is directly exposed to the processor core via a system call interface: the program can send packets on specific flows, poll for packets waiting at the processor ingress, and receive packets from specific queues. The sending and receiving process models a DMA, freeing the processor while the packets are being sent and received. Each processor features a private memory, and the simulated hardware does not model a shared memory abstraction.

B. Network model

Figure 2 illustrates the basic datapath of a NoC router modeled by DARSIM. There is one ingress port and one egress port for each neighboring node, as well as for each injector (or CPU core) connected to the switch; each ingress port contains any number of virtual channel buffers (VCs), which buffer flits until they can traverse the crossbar into the next-hop node.

As in any ingress-buffered wormhole router, packets arrive at the ingress ports flit-by-flit, and are stored in the appropriate virtual channel buffers. When the first flit of a packet arrives at the head of a VC buffer, the packet enters the route computation (RC) stage and the next-hop egress port is determined according to the routing algorithm. Next, the packet waits in the VC allocation (VA) stage until granted a next-hop virtual channel according to the chosen VC allocation scheme. Finally, in the switch arbitration (SA) stage, each flit of the packet competes for access to the crossbar and transits to the next node in the switch traversal (ST) stage. The RC and VA steps are active once per packet (to the head flit), while the SA and ST stages are applied per-flit.



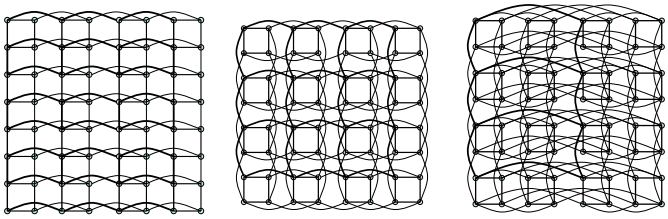
(a) XY (b) O1TURN (c) two-phase ROMM

Fig. 3. Example routes for a flow between a source (node 6) and a destination (node 2) for three oblivious routing algorithms. A single path is highlighted in dark gray while other possible paths are shown in light gray.

1) Interconnect geometry: The nodes in a system modeled by DARSIM can be configured with pairwise connections to form any geometry, including rings, multi-layer meshes (see Figure 4), and tori. Each node may have as many ports as desired: for example, most nodes in the 2D mesh shown in Figure 1 have five ports (four facing the neighboring nodes and one facing the CPU); the number and size of virtual channels can be controlled independently for each port, allowing the CPU↔switch ports to have different VC configuration from the switch↔switch ports.

2) Routing: DARSIM supports oblivious, static, and adaptive routing. A wide range of oblivious and static routing schemes is possible by configuring per-node routing tables. These are addressed by the flow ID and the incoming direction $\langle prev\_node\_id, flow\_id \rangle$, and each entry is a set of weighted next-hop results $\{\langle next\_node\_id, next\_flow\_id, weight \rangle, \cdots\}$. If the set contains more than one next-hop option, one is selected at random with propensity proportionate to the relevant *weight* field, and the packet is forwarded to *next_node_id* with its flow ID renamed to *next_flow_id*.

For example, in the case of simple XY routing, shown in Figure 3a, the routing tables for nodes 2, 5, 6, 7, and 8 would contain one entry for the relevant flow, addressed by the previous node ID (or 6 for the starting node 6) and the



(a) 3D mesh–x1 (b) 3D mesh–x1y1 (c) 3D mesh–xcube

Fig. 4. Planar view of three example multilayer mesh interconnect geometries which can be directly configured in DARSIM.

flow ID; the lookup result would direct the packet to the next node along the red path (or 2 for the terminal node 2) with the same flow ID and weight of 1.0. Static routing [6] is handled similarly. For O1TURN routing [7], illustrated in Figure 3b, the table at the start node (6) would contain two next-hop entries (one with next-hop node 3 and the other with next-hop node 7) weighted equally at 0.5, and the destination node (2) would have two entries (one arriving from node 1, and the other from node 5); the remaining tables do not differ from XY.

DARSIM's table-driven routing directly supports probabilistic oblivious routing algorithms such as PROM [8]. Probabilistic routing algorithms which first route the packet to a random intermediate node (say via XY routing) and only then to the final destination (e.g., Valiant [9] and its minimum-rectangle variant ROMM [10]). For example, the red path in Figure 3c shows one possible route from node 6 to node 2 in a two-phase ROMM scheme: the packet is first routed to node 4 and then to its final destination. To fill the routing tables, we must solve two problems: (a) remember whether the intermediate hop has been passed, and (b) express several routes with different intermediate destinations but the same next hop as one table entry. The first problem is solved by *changing the flow ID* at the intermediate node, and renaming the flow back to its original ID once at the destination node; the second problem corresponds to sending the flow to one of two possible next-hop nodes weighted by the ratio of possible flows going each way regardless of their intermediate nodes. Consider, as an example, the routing entries at node 4 for a flow from node 6 to node 2. A packet arriving from node 3 must have passed its intermediate hop at node 4 (because otherwise XY routing to the intermediate node would have restricted it to arriving from node 7) and can only continue on to node 5 without renaming the flow. A packet arriving at node 4 from node 7 must not have passed its intermediate node (because otherwise it's out of turns in its second XY phase and it can't get to its destination at node 2); the intermediate node can be either node 1 (with one path) or node 4 itself (also with one path), and so the table entry would direct the packet to node 1 (without flow renaming) or to node 5 (with flow renaming) with equal probability.

3) Virtual channel allocation: Like routing, virtual channel allocation (VCA) is table-driven. The VCA table lookup uses the next-hop node and flow ID computed in the route computation step, and is addressed by the four-tuple $\langle prev\_node\_id, flow\_id, next\_node\_id, next\_flow\_id \rangle$. As with table-driven routing, each lookup may result in a set of possible next-hop VCs $\{\langle next\_vc\_id, weight \rangle, \cdots\}$, and the VCA step randomly selects one VC among the possibilities according to the weights.

This directly supports dynamic VCA (all VCs are listed in the result with equal probabilities) as well as static set VCA [11] (the VC is a function of on the flow ID). Most other VCA schemes used to avoid deadlock, such as that of O1TURN (where the XY and YX subroutes must be on different VCs), Valiant/ROMM (where each phase has a separate VC set), as well as various adaptive VCA schemes like the turn model [12], are easily implemented as a function of the current and next-hop flow IDs.

Finally, DARSIM supports VCA schemes where the next-hop VC choice depends on the *contents* of the possible next-hop VCs, such as EDVCA [13] or FAA [14].

4) Bidirectional links: DARSIM allows inter-node connections to be bidirectional: links can optionally change direction as often as on every cycle based on local traffic conditions, effectively trading off bandwidth in one direction for bandwidth in the opposite direction [15]. To achieve this, each link is associated with a modeled hardware arbiter which collects information from the two ports facing each other across the link (for example, number of packets ready to traverse the link in each direction and the available destination buffer space) and suitably sets the allowed bandwidth in each direction.

5) Avoiding adversarial traffic patterns: The performance of routing and VC allocation algorithms can be heavily affected by the regular nature of the synthetic traffic patterns often used for evaluation: for example, a simple round-robin VCA scheme can exhibit throughput unfairness and cause otherwise equivalent flows to experience widely different delays if the traffic pattern injects flits in sync with the round-robin period. Worse yet, a similarly biased crossbar arbitration scheme can potentially block traffic arriving from one neighbor by always selecting another ingress port for crossbar traversal.

While relatively sophisticated arbitration algorithms have been developed (e.g., *i*SLIP [16]), the limited area and power in an NoC, together with the requirement for fast line-rate decisions, restricts the complexity of arbitration schemes and, consequently, their robustness to adversarial traffic patterns. Instead of selecting one such algorithm, therefore, DARSIM employs randomness to break arbitration ties: for example, the order in which next-in-line packets are considered for VC allocation, and the order in which waiting next-in-line flits are considered for crossbar traversal, are both randomized. While the pseudorandom number generators are by default initialized from an OS randomness source, the random seeds can be set by the user when exact reproducibility is required.

C. Concurrency, synchronization, and correctness

DARSIM takes advantage of modern multicore processors by automatically distributing simulation work among the available cores; as we show in Section III, this results in significant speedup of the simulations.

The simulated system is divided into tiles comprising a single virtual channel router and any traffic generators connected to it (cf. Figure 1), as well as a private pseudorandom number generator and any data structures required for collecting statistics. One execution thread is spawned for each available processor core (and restricted to run only on that core), and each tile is mapped to a thread; thus, some threads may be responsible for multiple tiles but a tile is never split across threads. Inter-thread communication is thus limited to flits crossing from one node to another, and some fundamentally

Characteristic	Configuration
Topology	8×8 2D mesh
Routing	XY, O1TURN, ROMM
VC alloc	dynamic, EDVCA
Link bandwith	1 flit/cycle
VCs per port	4, 8
VC buffer size	4, 8 flits
Avg. packet size	8 flits
Traffic workloads	transpose, bit-complement, shuffle, H.264 decoder profile; SPLASH-2 traces: FFT, RADIX, SWAPTIONS, WATER
Warmup cycles	200,000 for synthetic traffic; 0 for applications
Analyzed cycles	2,000,000 for synthetic traffic; full running time for applications
Server CPU	Intel i7 960 4-core with HT
# HT cores used	1, 2, 4, 6, 8
Sync period	clock-accurate, 5, 10

TABLE I
SYSTEM CONFIGURATIONS USED IN SIMULATION

sequential but rarely used features (such as writing VCD dumps).

Functional correctness requires that inter-tile communication be safe: that is, that all flits in transit across a tile-to-tile link arrive in the order they were sent, and that any metadata kept by the virtual channel buffers (e.g., the number of flits remaining in the packet at the head of the buffer) is correctly updated. In multithreaded simulation mode, DARSIM accomplishes this by adding two fine-grained locks in each virtual channel buffer—one lock at the tail (ingress) end of the VC buffer and one lock at the head (egress) end—thus permitting concurrent access to each buffer by the two communicating threads. Because the VC buffer queues are the only point of communication between the two threads, correctly locking the ends when updates are made ensures that no data is lost or reordered.

With functional correctness ensured, we can focus on the correctness of the performance model. One aspect of this is the faithful modeling of the parallelism inherent in synchronous hardware, and applies even for single-threaded simulation; DARSIM handles this by having a positive-edge stage (when computations and writes occur, but are not visible when read) and a separate negative-edge stage (when the written data are made visible) for every clock cycle.

Another aspect arises in concurrent simulation: a simulated tile may instantaneously (in one clock cycle) observe a set of changes effected by another tile over several clock cycles. A clock-cycle counter is a simple example of this; other effects may include observing the effects of too many (or too few) flit arrivals and different relative flit arrivals. A significant portion of these effects is addressed by keeping most collected statistics with the flits being transferred and updating them on the fly; for example, a flit's latency is updated incrementally at each node as the flit makes progress through the system, and is therefore immune to variation in the relative clock rates of different tiles.

The remaining inaccuracy is controlled by periodically synchronizing all threads on a barrier. 100% accuracy demands that threads be synchronized twice per clock cycle (once on the positive edge and once on the negative edge), and, indeed, simulation results in that mode precisely match those obtained from sequential simulation. Less frequent synchronizations are also possible, and, as discussed in Section III, result in significant speed benefits at the cost of minor accuracy loss.

III. DISCUSSION

In this section, we first argue that the details of network configuration, microarchitectural details, and congestion effects make a significant difference in network performance by comparing DARSIM simulation results on several configurations. Then, we quantify the speed advantages of DARSIM's parallelized design, and the accuracy vs. speedup tradeoffs offered by loose synchronization.

A. Methods

To support our claims with concrete examples, we ran DARSIM simulations with various system configurations. The salient configuration features used in various combinations in our experiments are listed in Table I.

SPLASH-2 application traces were obtained by running the benchmarks [5] in the distributed x86 multicore simulator Graphite [17] with 64 application threads and using Graphite's estimated network model; all network transmissions caused by memory accesses were logged and the traces were then replayed in DARSIM. To obtain significant network congestion, the x86 core was assumed to run on a clock ten times faster than the network. This was necessary because the SPLASH benchmarks were written for a multiprocessor environment where the cost of inter-processor communication was much higher, and thus particular attention was paid to frugal communication; the plentiful bandwidth and relatively

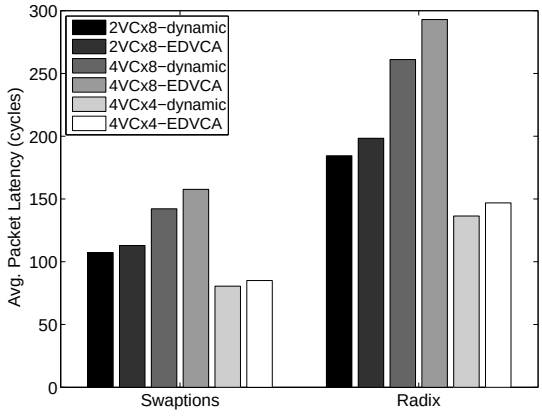


Fig. 5. In-network latency for different VC buffer configurations. Counterintuitively, increasing the number of VCs from 2 to 4 while keeping the VC sizes constant at 8 flits actually *increases* in-network latency because packets can be buffered inside the network. When total VC memory size is held constant, doubling the number of VCs to 4 (and correspondingly halving their capacities to 4 flits) decreases latency as expected. We ran the same experiment on other applications (WATER, and FFT) but the results exhibited the same pattern and so we omit them for brevity.

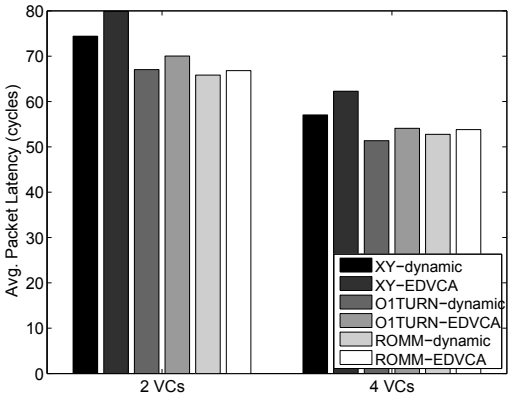


Fig. 6. The effect of routing and VC configuration on network transit latency in a relatively congested network on the WATER benchmark: while O1TURN and ROMM clearly outperform XY, the margin is not particularly impressive. (The remaining application benchmarks are broadly similar except for scaling due to higher or lower network load, and are not shown).

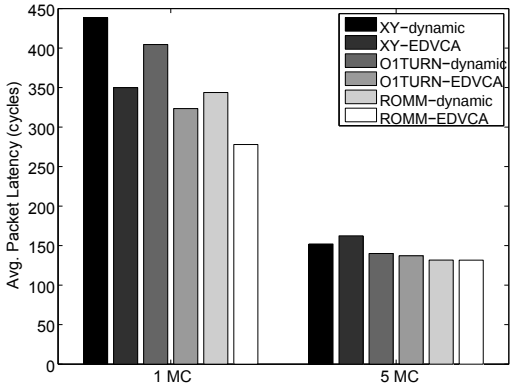


Fig. 7. The effect of varying the number of memory controllers on in-network latency (and therefore memory system performance). While multiple memory controllers significantly reduce congestion, replacing one memory controller (1 MC) with five (5 MC) does not increase performance anywhere close to five-fold.

short latencies available in NoC-based multicores make this kind of optimization less critical today.

Although each simulation collects a wide variety of statistics, most reports below focus on average in-network latency of delivered traffic—that is, the number of cycles elapsed from the time a flit was injected into a network router ingress port to the time it departed the last network egress port for the destination CPU—as most relevant to current and future cache-coherent shared-memory NoC multicores. For speedups, we measured elapsed wall-clock times with DARSIM the only significant application running on the relevant server. Finally, to quantify the accuracy of the loosely synchronized simulations, we first ran DARSIM with full clock-accurate synchronization to obtain a baseline; we then repeated the experiment with different synchronization periods (but the same random number seed etc.) and compared the reported average latencies as an accuracy measurement.

B. Simulation-driven architectural decisions

A cycle-accurate simulation is indispensable when congestion must be modeled accurately. For example, network congestion can have significant effects on how the network configuration—say the number and size of the virtual channels—affects in-network latency (i.e., the latency incurred after the relevant flit is seen by the processor as successfully sent). Intuitively, adding more virtual channels should generally allow more packets from different flows to compete for transmission across the crossbar, increase crossbar efficiency, and therefore reduce observed packet latency. While this holds when traffic is light, it may have an opposite effect in a relatively congested network: as Figure 5 illustrates on two SPLASH-2 applications, doubling the number of VCs while holding the size of each VC constant causes the observed in-network latency in a relatively congested network to actually *increase*. This is because the total amount of buffer space in the network also doubled, and, when traffic is heavy and delivery rates are limited by the network bandwidth, the flits at the tails of the VC queues must compete with flits from more VCs and thus experience longer delays. Indeed, when the VC queue sizes are halved to keep the total amount of buffer space the same, the 4-VC setup exhibits shorter latencies than the 2-VC equivalent as originally expected.

While in a lightly loaded network almost any routing and VC allocation algorithm will perform well, heavier loads lead to different congestion under different routing and VC algorithms and performance is significantly affected; again, accurately evaluating such effects calls for a cycle-level simulator and real applications. Figure 6 shows the effect of routing and VC allocation scheme on performance of the SPLASH-2 WATER benchmark in a relatively congested network. While the algorithms with more path diversity (O1TURN and ROMM) do lower observed in-network latency, the performance increase is not as much as might be expected by considering the increased bandwidth available to each flow.

Modern network-on-chip designs can reduce congestion by placing several independent memory controllers in different parts of the network. Since in a cache-coherent system a memory controller generally communicates with all processor cores, modeling congestion is critical in evaluating the tradeoff between adding memory controllers and controlling chip area and pin usage. For example, Figure 7 shows in-network latency for two cache-coherent systems: one with one memory controller and the other with five. Although performance clearly improves with five memory controllers, the improvement does not approach five-fold reduction especially for the more congestion-friendly routing and VC allocation schemes. More significantly, the two choices impose different constraints on selecting the routing and VC allocation logic: while the congestion around a central memory controller makes controlling congestion via routing and VC allocation, the average latency in a system with five memory controllers does not vary significantly under different routing algorithms

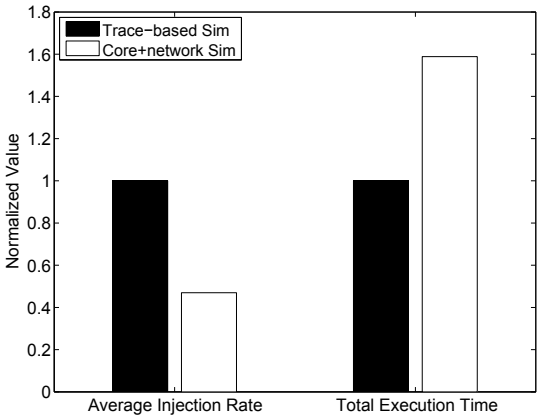


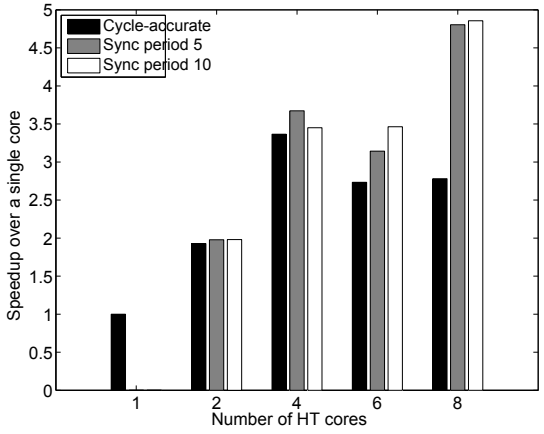
Fig. 8. Trace-based simulation lacks the feedback loop from the network to the sending (or receiving) core; this allows cores to inject packets unrealistically fast and permits the application to finish much earlier than realistically possible.

and EDVCA, and the designer might choose to save area and reduce implementation complexity in the network switch.

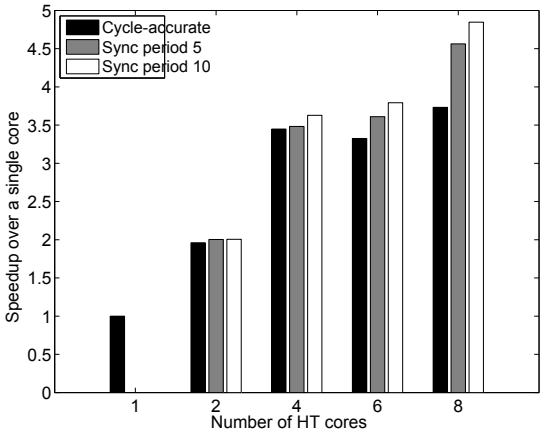
Much of the research in network-on-chip microarchitecture relies on synthetic traffic patterns or application traces collected under the assumption of an ideal interconnect network. This approach generally ignores the interdependencies among the various flows and the delays caused by instructions that must wait until network packets are delivered (for example, memory accesses that miss the per-core cache hierarchy). For these reasons, a precise evaluation of the performance of NoC-based multicores in running real applications requires that the CPU core and the network be simulated together.

To quantify the differences between trace-driven and real application traffic, we implemented Cannon's algorithm for matrix multiplication [18] in C targeting the MIPS core simulator that ships with DARSIM. We ran the simulation on 64 cores and applied it to a 128×128 matrix; to stress the network, cores were mapped randomly, per-cell data sizes were assumed to be large, and computations were taken to be relatively fast. For the trace version, we assumed an ideal single-cycle network, logged each network transmission event, and later replayed the traces in DARSIM; for the combined core+network version, we ran the benchmark with the MIPS cores simulated by DARSIM directly interacting with the on-chip network.

The results, shown in Figure 8, illustrate that the processor cores may have to spend significant amounts of time waiting for the network. On the one hand, a destination node waiting for a packet may block until the packet arrives. On the other hand, the sending node may have to wait for the destination core to make progress: when the destination is nearby (e.g., adjacent), even a relatively short packet can exceed the total buffer space available in the network, and the sending core may have to stall before starting the following packet until the current packet has been at least somewhat processed by the destination core and network buffers have freed up.



(a) low traffic



(b) high traffic

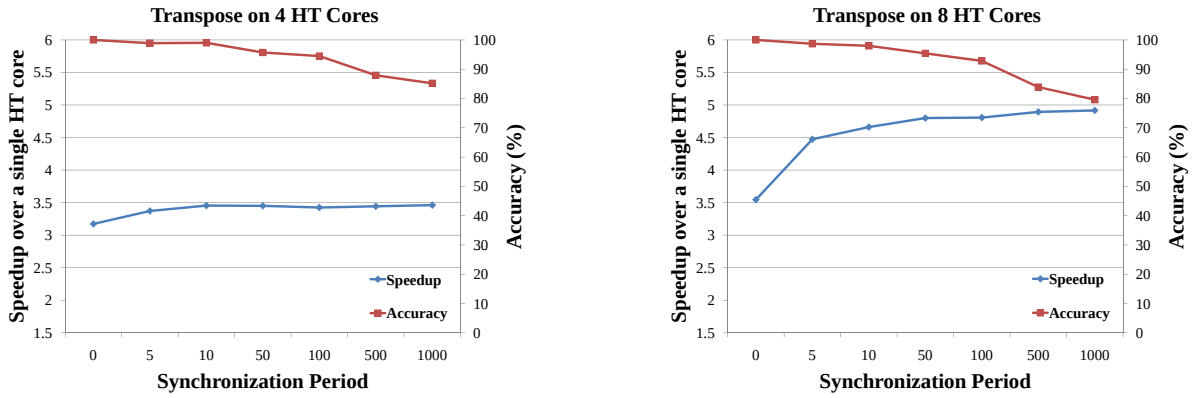
Fig. 9. Speedups achieved by running the SHUFFLE benchmark with DARSIM parallelized to hyperthreaded (HT) cores (ratios normalized to single-core performance). Fully synchronized simulation speed scales almost linearly with the number of physical cores (4), but drops when more than one hyperthreaded core is used per physical core. Results from other benchmarks show largely identical trends and are omitted for brevity.

C. Parallel speedup

In this section we focus on performance gains achieved by DARSIM's parallel runtime and performance versus accuracy tradeoffs enabled by loose synchronization.

To evaluate possible gains, we repeated a series of 64-node NoC simulations with DARSIM running 1, 2, 4, and 8 threads on a quad-core Core i7 with two hyperthreads on each core (for a total of 8 cores visible to the OS). Threads were mapped to different virtual cores (the first four to separate physical cores, and the next four pairwise shared with the first four) and restricted to the same core throughout the simulation. To avoid bias towards any specific routing scheme, simulated tiles were assigned to threads randomly in such a way that each thread was responsible for roughly the same number of tiles.

Figure 9 illustrates the performance increase obtained relative to the number of cores. Performance scales almost linearly as long as the threads run on separate physical processor cores, reaching $3.5\times$ speedup on four separate physical cores. When



(a) 4 HT cores

(b) 8 HT cores

Fig. 10. Speedup and accuracy drop somewhat with very infrequent synchronization, but accuracy remains above 80% even on eight hyperthreaded cores. The other benchmarks show the same trends, but are even less affected by large synchronization periods, with accuracy never dropping below 90%.

some physical processors begin to execute two threads via hyperthreading, however, performance of the fully synchronized simulation drops significantly; this bottleneck is mostly due to contention, and loose synchronization recovers some speed and the speedup with synchronization on every tenth cycle almost reaches 5× on eight virtual cores. Threads in high-traffic experiments have more work to do in every simulated clock cycle and are therefore less sensitive to synchronization inefficiency.

For the experiments with loose synchronization every 5 or 10 cycles, we quantified accuracy by comparing the reported average latency with the corresponding fully synchronized run. When each thread ran on a separate physical core, accuracy never dropped below 97.4% for synchronization period 5 and 95.3% for period 10 (both observed in the low-traffic TRANSPOSE experiment). Accuracy was somewhat worse on six virtual cores (i.e., two of the four cores running two threads), dropping to 92.1% for synchronization every 5 cycles and 90.6% for synchronization on every 10$^{\text{th}}$ cycle.

We reasoned that, as the number of cores increases, more and more loose synchronization may be required to offset synchronization costs. We therefore directly evaluated speedup and the inherent loss of accuracy against synchronization periods ranging from cycle-accurate to synchronization every 1,000 cycles.

Figure 10 shows the worst case among the four synthetic benchmarks we ran: with a synchronization period of 1,000 cycles, accuracy hovers around 86% for the 4-core simulation using separate physical cores and drops to 80% on 8 virtual cores (the other benchmarks show the same trend, with accuracy dropping much more slowly and never below 90%).

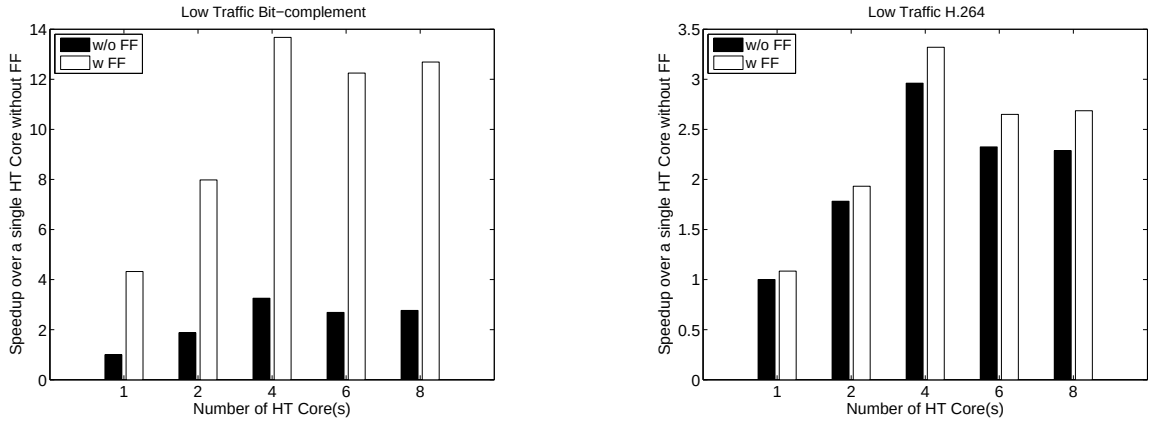
The 4-core simulations show nearly consistent 3.25×–3.5× speedup over the entire range of synchronization periods, while the 8-core simulations benefit significantly from looser synchronization. This is because, in the case of 4 cores, the computation in each core is larger and most of the possible speedup is achieved even in the fully synchronized, clock-accurate case by distributing the work among the four cores. In the 8-core case, however, the per-core workload for each simulated cycle is much smaller, and so increasing the synchronization period shows a comparably large speedup. Naturally, once the period grows enough for the computational workload assigned to each core to overcome the synchronization penalty, speedup reaches a saturation point.

DARSIM can fast-forward the clocks in each tile when there are no flits buffered in the network and no flits about to be injected for some period of time. Because in that situation no useful work can possibly result, DARSIM advances the clocks to the next injection event and continues cycle-by-cycle simulation from that point without altering simulation results.

Clearly, heavy traffic loads will not benefit from fast-forwarding because the network buffers are never drained and DARSIM never advances the clock by more than one cycle. Figure 11 shows that the benefit on low-volume traffic depends intimately on the traffic pattern: an application which, like bit complement, has long pauses between traffic bursts, will benefit significantly; an application which spreads the small amount of traffic it generates evenly over time like the H.264 profile will rarely allow the network to fully drain and therefore will benefit little from fast-forwarding.

IV. Related work

One NoC simulator that stands out among the many simple, limited-purpose software NoC simulators is Garnet [19]. Like DARSIM, Garnet models an NoC interconnection network at the cycle-accurate level: the model allows either a standard ingress-queued virtual channel router with a rigid five-stage pipeline or a flexible egress-queued router. Integration with GEMS provides a full-system simulation framework and a memory model, while integration with ORION [20] provides power estimation. RSIM [21] simulates shared-memory multiprocessors and uniprocessors designed for high instruction-level parallelism; it includes a multiprocessor coherence protocol and interconnect, and models contention at all resources. SICOSYS [22] is a general-purpose interconnection network simulator that captures essential details of low-level simulation, and has been integrated in RSIM. Noxim [23] models a



(a) bit complement

(b) H.264 profile

Fig. 11. Performance benefits from fast-forwarding. Unlike low-traffic bit complement, which sends traffic in coordinated bursts and sees significant speedups when the network is idle, the low-traffic H.264 profile gains little because packets are sent relatively constant frequency and the network is rarely fully drained.

mesh NoC and, like DARSIM, allows the user to customize a variety of parameters like network size, VC sizes, packet size distribution, routing scheme, etc.; unlike DARSIM, however, it's limited to 2D mesh interconnects.

Highly configurable architectural modeling is not a new idea. The Simplescalar toolset [24] can model a variety of processor architectures and memory hierarchies, and enjoys considerable popularity among computer architecture researchers. Graphite [17] is a PIN-based multicore simulator that stands out for its ability to model thousands of cores by dividing the work among not just multiple cores on the same die but multiple networked computers. Finally, the growth in complexity and the need for ever-increasing amounts of verification has led to the development of hardware emulator platforms like RAMP [25], which, though more difficult to configure, are much faster than software solutions.

V. Conclusion

We have introduced DARSIM, a highly configurable, cycle-accurate network-on-chip simulator with a parallelized simulation engine that scales nearly linearly with the number of physical cores in the processor while preserving cycle-accurate behavior, and allows the user to obtain even more speed at the cost of some accuracy via loose synchronization.

Two areas of ongoing development include an accurate power model (e.g., ORION [20]) and the integration with a parallel OS-level full-system simulator such as Graphite [17].

References

[1] S. Rusu *et al.*, "A 45nm 8-core enterprise Xeon® processor," in *Proceedings of the IEEE Asian Solid-State Circuits Conference*, 2009, pp. 9–12.

[2] S. Bell *et al.*, "TILE64 - processor: A 64-Core SoC with mesh interconnect," in *Proceedings of the IEEE International Solid-State Circuits Conference*, 2008, pp. 88–598.

[3] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the Design Automation Conference*, 2007, pp. 746–749.

[4] D. Wentzlaff *et al.*, "On-Chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, 2007.

[5] S. Woo *et al.*, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the International Symposium on Computer Architecture*, 1995, pp. 24–36.

[6] M. A. Kinsy *et al.*, "Application-aware deadlock-free oblivious routing," in *Proceedings of the International Symposium on Computer Architecture*. Austin, TX, USA: ACM, 2009, pp. 208–219.

[7] D. Seo *et al.*, "Near-optimal worst-case throughput routing for two-dimensional mesh networks," in *Proceedings of the International Symposium on Computer Architecture*, 2005, pp. 432–443.

[8] M. H. Cho *et al.*, "Path-based, randomized, oblivious, minimal routing," in *International Workshop on Network on Chip Architectures*, 2009, pp. 23–28.

[9] L. G. Valiant and G. J. Brebner, "Universal schemes for parallel communication," in *Proceedings of the ACM Symposium on Theory of Computing*. Milwaukee, Wisconsin, United States: ACM, 1981, pp. 263–277.

[10] T. Nesson and S. L. Johnsson, "ROMM routing: A class of efficient minimal routing algorithms," in *Proceedings of the International Workshop on Parallel Computer Routing and Communication*. Springer-Verlag, 1994, pp. 185–199.

[11] K. S. Shim *et al.*, "Static virtual channel allocation in oblivious routing," in *International Symposium on Networks-on-Chip*, 2009, pp. 38–43.

[12] C. J. Glass and L. M. Ni, "The turn model for adaptive routing," in *Proceedings of the International Symposium on Computer Architecture*. Queensland, Australia: ACM, 1992, pp. 278–287.

[13] M. Lis *et al.*, "Guaranteed in-order packet delivery using exclusive dynamic virtual channel allocation," MIT CSAIL, Tech. Rep. MIT-CSAIL-TR-2009-036, 2009.

[14] A. Banerjee and S. Moore, "Flow-aware allocation for on-chip networks," in *Proceedings of the International Symposium on Networks-on-Chip*, 2009, pp. 183–192.

[15] M. H. Cho *et al.*, "Oblivious routing in On-Chip Bandwidth-Adaptive networks," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2009, pp. 181–190.

[16] N. McKeown, "The *i*SLIP scheduling algorithm for input-queued switches," *The IEEE/ACM Transactions on Networking*, vol. 7, no. 2, pp. 188–201, 1999.

[17] J. E. Miller *et al.*, "Graphite: A distributed parallel simulator for multicores," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2010, pp. 1–12.

[18] L. E. Cannon, "A cellular computer to implement the Kalman Filter Algorithm," Ph.D. dissertation, Montana State University, 1969.

[19] N. Agarwal *et al.*, "GARNET: a detailed on-chip network model inside a full-system simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 33–42.

[20] H. Wang *et al.*, "Orion: a power-performance simulator for interconnection networks," in *Proceedings of the International Symposium on Microarchitecture*, 2002, pp. 294–305.

[21] V. S. Pai *et al.*, "RSIM: Rice Simulator for ILP Multiprocessors," *SIGARCH Comput. Archit. News*, vol. 25, no. 5, p. 1, 1997.
[22] V. Puente *et al.*, "Sicosys: An integrated framework for studying interconnection network performance in multiprocessor systems," *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, vol. 0, p. 0015, 2002.
[23] (2010) Noxim, the NoC Simulator. [Online]. Available: http://noxim.sourceforge.net/
[24] T. Austin *et al.*, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.
[25] Arvind *et al.*, "RAMP: Research Accelerator for Multiple Processors — a community vision for a shared experimental parallel HW/SW platform," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-05-1412, Sep 2005.