Achieving Predictable Performance through Better Memory Controller Placement in Many-Core CMPs[*]

Dennis Abts[*] dabts@google.com
Natalie Enright Jerger[§] enright@eecg.toronto.edu
John Kim[‡] jjk12@northwestern.edu
Dan Gibson[†] gibson@cs.wisc.edu
Mikko Lipasti[†] mikko@engr.wisc.edu

[*]Google Inc.
Madison, Wisconsin

[§]University of Toronto
Toronto, ON

[‡]Northwestern University
Evanston, Illinois

[†]University of Wisconsin
Madison, Wisconsin

Abstract

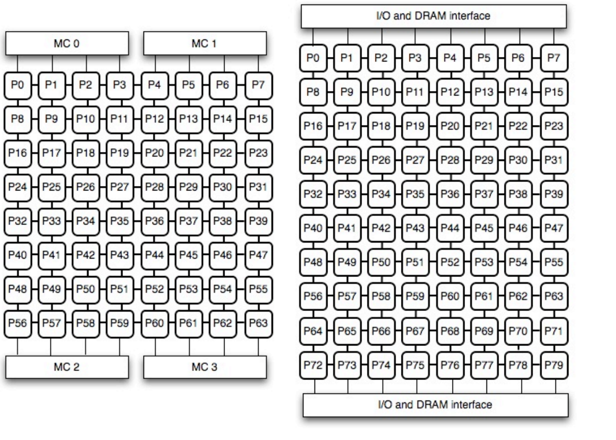
In the near term, Moore's law will continue to provide an increasing number of transistors and therefore an increasing number of on-chip cores. Limited pin bandwidth prevents the integration of a large number of memory controllers on-chip. With many cores, and few memory controllers, where to locate the memory controllers in the on-chip interconnection fabric becomes an important and as yet unexplored question. In this paper, we show how the location of the memory controllers can reduce contention (hot spots) in the on-chip fabric, as well as lower the variance in reference latency which provides for predictable performance of memory-intensive applications regardless of the processing core on which a thread is scheduled. We explore the design space of on-chip fabrics to find optimal memory controller placement relative to different topologies (i.e. mesh and torus), routing algorithms, and workloads.

1 Introduction

Increasing levels of silicon integration are motivating system on chip (SoC) and chip multiprocessor (CMP) designs with large processor counts and integrated memory controllers. Proof-of-concept designs from both Tilera [22, 26] and Intel [13, 25] integrate as many as 80 cores on a single piece of silicon. System architects are faced with the trade-off of many lightweight cores (with simple, in-order issue) versus fewer heavyweight cores (with aggressive speculation, multiple issue, etc); however, both design points require abundant DRAM bandwidth to feed the memory hierarchy.

The most significant design impediment to increasing scale, is limited pin bandwidth to memory devices. Memory bandwidth has improved with recent high-speed differential signaling [10], FB-DIMM technology [8] and on-board memory buffers to serve as pin expanders converting from narrow serial channels to a wide address/data/control bus used by the memory part. Nonetheless, packaging constraints limited primarily by the number of available pins restrict the number of memory controllers to a small fraction relative to the number of processing cores. The reality of *many* cores with *few* memory controllers raises the important question of *where* the memory controllers should be located within the on-chip network.

The Tilera "Tile Architecture" [26] is implemented as an 8×8 two-dimensional mesh of *tiles* (Figure 1a). Packets are routed using dimension-order routing and wormhole flow control. The Tilera on-chip network uses five independent physical networks to



(a) Tilera 8x8 organization (b) Intel 80-core Teraflop chip 10x8 organization

Figure 1. Two recent many-core CMPs that use a two-dimensional mesh of tiles.

[*]This research was supported in part by NSF grants under ..., an IBM PhD fellowship, as well as grant and equipment donations from IBM and Intel. The authors would like to thank the anonymous reviewers for thsir comments and suggestions for improving this work

isolate traffic[1], where each full-duplex link is 32-bits wide in each direction. Each of the physical networks has a 5-ported router in each tile that flows packets in the north, south, east, and west directions of the 2D mesh, as well as ingress and egress traffic from the processor. In aggregate, the five networks provide 1.28 Tb/s of bandwidth per tile, or 2.56 Tb/s of minimum bisection bandwidth for the 8×8 mesh. Each physical network corresponds to a different communication model, either shared memory or direct communication via user-level messaging.

The Intel 80-core design [25] is organized as a 10×8 two-dimensional mesh (Figure 1b) of tiles, where each tile embodies a processing core which interfaces to a 5-ported router. The switch operates at 4GHz and routes packets using wormhole flow control between tiles. Two virtual channels are provided for deadlock avoidance[2]. Each input queue is only 16 flits deep (each flit is 39-bits), since it only has to cover the latency × bandwidth product between tiles. Each full-duplex router link can move a 39-bit flit (32-bits of data and 7-bits of sideband) on every 4GHz clock, providing 16GB/s per direction. The processor has 16GB/s of ingress/egress bandwidth into/from the router, for a total non-blocking bandwidth of 80GB/s per tile. A round-robin arbitration across the router input ports is used for fair access to the output ports.

The designs from Intel [25] and Tilera [26] both use a mesh topology for the on-chip network and have the memory controllers positioned near the top and bottom of the fabric (Figure 1). As the number of processor cores grow, it is not practical to assume each tile will have a memory controller directly attached[3]. As a result, a many-core CMP with n processors and m memory ports will have $\binom{n}{m}$ possible permutations for *where* the memory controllers could be located within the on-chip fabric. These different memory controller configurations can have a dramatic impact on the latency and bandwidth characteristics of the on-chip network, especially for a mesh topology which is not edge symmetric like a torus, for example. Furthermore, by reducing the *variance* in packet latency as well as channel load, the on-chip network is less sensitive to the processor core on which a thread is scheduled. Through careful placement of the memory controllers we can improve performance and provide predictable latency-bandwidth characteristics regardless of where a thread executes on the CMP.

1.1 Contributions

Modern flip-chip packaging allows sufficient escape paths from anywhere on the chip, which leaves open the question of *where* to place each memory controller within the on-chip network so that we minimize both latency and link contention. The memory controllers themselves do not have to be part of the on-chip network, rather, the ingress/egress ports to/from the memory are. Additionally, the number of on-chip network ports to/from memory does not need to be equal to the number of memory controllers. The combination of *where* the memory controllers are located and routing algorithm will significantly influence how much traffic each link will carry. In this paper, we make several contributions to on-chip interconnection networks:

- Most prior research has focused on intra-chip (processor-to-processor) communication. This is the first work to evaluate the impact *location* of memory controller and the influence of processor-to-memory traffic for on-chip networks.
- We propose *class-based deterministic routing* (CDR)[4] to load-balance the processor-to-memory traffic in a 2D on-chip mesh topology. Other routing algorithms such as O1turn [19] cannot effectively load balance processor-to-memory traffic.
- We explicitly enumerate all possible permutations of memory controllers in both mesh and torus topologies that are common in two-dimensional on-chip networks, and use extensive simulation to find the configuration that minimizes the *maximum channel load*. We show that exhaustive simulation is possible for modest-sized on-chip networks ($k < 7$), however, larger networks such as an 8×8 mesh, require a heuristic-guided search to deal with the computational complexity that arises from a large search space.
- We explore the design space of on-chip networks to show how memory controller location and routing algorithm can improve the latency and bandwidth characteristics as well as reduce *variance* of the network for both synthetic and full system workloads. Our solution provides predictable performance regardless of which processor core is used to execute a memory-intensive thread.

1.2 Paper organization

The remainder of this paper is organized as follows. Section 2 provides further background and motivation for optimal placement of memory controllers. In Section 3, we briefly describe our methodology and present results for initially pruning the design space in Section 4. We then describe the our more detailed simulation methodologies and discuss the impact of placement and routing on the latency and bandwidth characteristics of the on-chip network in Section 5. In Section 6 we discuss other prior work. Finally, Section 7 summarizes our contributions.

2 Motivation

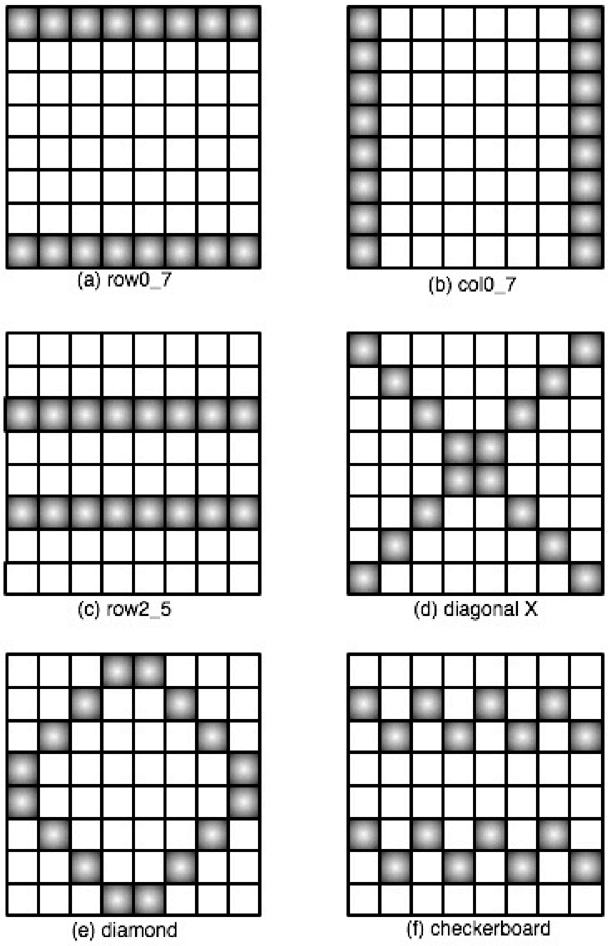
Typical multi-core processor designs, common in most modern servers, use a conventional crossbar design [2] that provides a 1:1 ratio of memory controllers and processor cores. Compared to an aggressive out-of-order processor with substantial hardware dedicated to handling multiple outstanding cache misses per core, the simpler processing cores of a many-core architecture will *demand* less memory bandwidth. Architectures that increase memory-level parallelism, by allowing more outstanding cache misses from each core, increase link contention in the on-chip fabric.

[1] Four of the five networks use dimension-ordered routing. The *static network* uses a circuit-switch-like mechanism to establish a channel from source to destination and then efficiently streams data without requiring route computation at each hop.

[2] The two virtual channels are used for to segregate request and reply traffic, thus avoiding protocol deadlock.

[3] Due to limited number of pins, it is not practical since each FB-DIMM interface, for example, requires 10 northbound differential signals, and 14 southbound differential signals.

[4] pronounced “cedar”


(a) row0_7

(b) col0_7

(c) row2_5

(d) diagonal X

(e) diamond

(f) checkerboard

Figure 2. Different memory controller configurations.

Figure 2 shows some point specific designs that we evaluate for 16 memory controller ports embedded in an 8×8 array of tiles (shaded squares represent a memory port co-located with a processor tile). The number of memory ingress/egress ports, or *taps* does not have to be the same as the number of physical memory channels. As shown in Figure 1a, the Tilera chip has 16 tiles with memory taps, and only four memory controllers. This organization makes it very important to *uniformly* spread the processor-to-memory traffic across all the available memory ports. Although [26] does not specifically discuss how the references are distributed, we use the address bits immediately above the cache line address to choose the memory port for a given address. Our baseline design (Figure 2a) was chosen because it was representative of how both Intel and Tilera chose to implement their memory interface.

We assume that a mesh or torus on-chip network with radix k will have $2k$ memory ports. These $2k$ memory ports will be multiplexed to a smaller number of memory controllers as dictated by the available pin bandwidth of the design [5]. Guided by intuition, we then chose several other configurations we thought might perform better (Figures 2c-f). However for scientific rigor, we did not limit our search of the design space to a small handful of configurations. Instead, we enumerated all possible configurations and simulated each one with 10,000 trials of random permutation traffic.

Intuitively, the `row2_5` configuration will have a lower average hop count for each processor to access all of the memory controllers. To the first order, this will improve average performance over the `row0_7`. However, the goal of this work is to find configurations that provide good performance and fair access to each memory controller. To that end, we examine the variation in latency experienced by processors to access each memory controller. A lower variance indicates that a memory controller configuration provides both fair and predictable access from all processors.

3 Simulation Methodology

We use several simulation environments, corresponding to different levels of abstraction and detail, to explore this broad design space. The first is a simple, and fast, *link contention* simulator which traces the path a packet takes through the network and increments a count on each link that it traverses. This count represents the channel load, or *contention*, that would be observed by the link if the processors were simultaneously active. The second environment is a detailed, event-driven simulator used to explore topology and routing sensitivity to different memory controller placement alternatives. It provides flit-level granularity and detailed simulation for *synthetic* workloads. Lastly, we have a detailed full system simulator that allows real workloads to be applied to what we learned from the previous two environments.

By providing multiple simulation approaches at differing levels of abstraction, we are able to validate and gain a better understanding of issues that may exist in one simulation environment, but not others. For example, after studying the *distribution* of memory references (Figure 4) in the TPC-H benchmark, it was apparent that some memory controllers were accessed much more frequently than others – with some memory controllers having up to 4× the load of others. To mimic this *hot spot* traffic pattern, we applied this traffic pattern as input to the event-driven network simulator by choosing the destination memory controller according to the distribution observed by the full system simulator. In this way, we were able to validate simulation models at differing levels of abstraction.

4 Pruning the Design Space

We use a link contention simulator, a genetic algorithm and a random simulation to prune the large design space of memory controller placement. Once a small subset of memory controller placements has been selected, we will describe, in detail the evaluation methods and results that provide further insight into this problem (Section 5).

[5] The Tilera design multiplexes 4 memory ports to 1 memory controller.

4.1 Link contention simulator

To gain a better understanding of how placement affects contention, or specifically, the *maximum channel load*, within the on-chip network we develop a simple simulator that traces the path of each packet. The *maximum channel load* is the load (in packets) on the channel carrying the largest fraction of traffic [5]. The network is modeled as a group of nodes interconnected with unidirectional channels. As a packet traverses each unidirectional channel, we increment a counter associated with that channel, and compare the count to the current *max_channel_load* value. We keep track of the maximum channel load as a *proxy* for the delivered bandwidth, since the accepted bandwidth will ultimately be limited by the channel with the highest contention, or channel load. All processor-to-memory references are modeled by having each processor choose a random memory controller in which to send a request. Once the request packet reaches the destination memory controller, the reply packet is sent back – again, tracing the path of the packet as it heads back to the requesting processor tile. We perform 10,000 trials averaging the maximum channel load across all the trials. This average value is used as a figure of merit for evaluating different memory configurations.

We use the contention simulator to enumerate all possible placement options, and then simulate 10,000 trials for each configuration. We do this for both mesh and torus topologies. For symmetric topologies, such as the torus, there is a lot of symmetry that can be exploited, however, our simulator does not take this symmetry into account. So, it is possible for multiple configurations to be viewed as *best* (i.e. perform identically). An on-chip network with n tiles and m memory controllers will have $\binom{n}{m}$ possible memory configurations that must be compared against each other in order to choose the best .

For small on-chip networks, say 4×4 mesh, with 8 memory ports, we have a total of $\binom{16}{8}$, or 12,870 different placements to evaluate. A 5×5 mesh with 10 memory ports has 3,268,760 different possibilities, and a 6×6 mesh has over one billion possible placements. Thus an 8×8 mesh with 16 memory ports has 4.9×10^{14} different configurations.making exhaustive search of the design space intractable for any network larger than 6×6. To deal with this complexity we use two approaches: genetic algorithms, and random simulation.

4.1.1 Genetic algorithm

Genetic algorithms [7] (GAs) take a heuristic-based approach to optimization. GAs are inspired by DNA's ability to encode complicated organisms into simple (if lengthy) sequences. Each sequence represents a potential solution to the problem under optimization. In our case, we represent our solutions as a bit vector; set bits in the vector represent locations of memory controllers in our topology. In the course of execution, solutions are combined to produce new solutions (analogous to chromosomal crossover), and new solutions are randomly perturbed (i.e., mutated) with some probability to prevent convergence on local minima. Each new solution is evaluated, and assigned a fitness.

The nature of crossover, mutation, and fitness evaluation operations is specific to the problem to be solved. The fitness of each solution is the reciprocal of the maximum channel load for that configuration. Our crossover algorithm selects two parent solutions from a large population, with probability proportional to the potential parents' fitness, then randomly selects bits from the parents to form a new solution. The mutation operation simply swaps adjacent bits in the vector. In order to maximize the effectiveness of our heuristic, we never evaluate a particular bit vector more than once. Instead, we repeatedly apply mutation to redundant solutions until a new solution is discovered. Our genetic simulator executes a fixed number of generations or returns a solution when stagnation occurs in the population[6].

4.1.2 Random simulation

We extended our link contention simulator to perform a random walk of the design space. We begin by randomly selecting a valid memory controller configuration, and keep track of which configuration has the least contention. Again, as our figure of merit, we use *maximum channel load* as a proxy for accepted bandwidth. The configuration with the lowest maximum channel load will have less congestion and as a result, the best delivered bandwidth. When we find a configuration that is better than all other previously explored, we annotate the configuration and clear the effort counter. An *effort* parameter to the simulator determines how many configurations we search before terminating the simulation and declaring a solution. Through experimentation, we found that an effort level of 7,000 provided a reasonable trade-off between search quality and time to solution, which was usually less than a few hours.

4.2 Results

At a high level, our link contention simulator is used to provide a first-order comparison of different memory controller configurations and exhaustively search for an optimal solution in relatively small on-chip networks (e.g. $k \leq 6$). When exhaustive search becomes intractable ($k > 6$) we use heuristic-guided search to find near-optimal solutions.

We began by exhaustively simulating 4×4, 5×5, and 6×6 mesh and torus on-chip networks. From the exhaustive simulation, a clear pattern emerged – configurations that spread the processor-to-memory traffic across the diagonal of the mesh performed notably better than others. Intuitively, this makes sense, since the link contention simulator uses dimension-ordered routing (X then Y) to route packets. If the memory controllers were all in the same row (as in Figure 2a) then the reply packets would get very congested. Table 1 shows the simulation results for an 8×8 array of tiles organized as shown in Figure 2. These configurations were chosen by extrapolating analogous patterns from smaller network sizes such as 6 × 6 which could be exhaustively searched. The `diamond` and `diagonal X` configurations perform 33% better than the baseline `row0_7` (i.e. Tilera memory configuration).

For larger networks, we had to rely on heuristic-guided search to find near-optimal solutions. The best solution we found via random search had a maximum channel load of 9.35, within 5% of

[6] Stagnation is defined as no improvement in observed fitness over some interval.

Memory Controller Configuration		Max. Channel Load	
		Mesh	Torus
row0_7	Figure 2a	13.50	9.25
col0_7	Figure 2b	13.50	9.25
row2_5	Figure 2c	13.49	9.22
diagonal X	Figure 2d	8.93	7.72
diamond	**Figure 2e**	**8.90**	**7.72**
checkerboard	Figure 2f	10.24	7.69

Table 1. Summary of link contention for memory configurations shown in Figure 2.

the diamond and diagonal X configurations. The genetic algorithm with a population size of 500 configurations for 100 generations, yielded a near optimal solution with a maximum channel load of 9.21, within 4% of the diamond and diagonal X memory configurations.The solutions generated from the genetic algorithm followed the trend of memory controllers clustered along the diagonals of the mesh.

Since diamond has better physical layout properties than diagonal X and the same performance, we will focus our discussion on it. Specifically, the diamond does not locate multiple memory controllers in the center of the chip, increasing escape complexity. By locating the memory controllers in an optimal manner, we can reduce the average latency and reduce the amount of energy expended per bit transported. Spreading the memory ports in a uniform manner such as the diamond will spread the thermals across a wider area. The diagonal pattern would increase accesses in the center of the chip would could increase thermals there.

In addition to search for the optimal configuration, we use the randomized search to sweep through the design space to determine the impact of having *many* processor cores and *few* memory controllers. As we vary the number of memory controllers, we search for the best memory configuration, and note the maximum channel load for that configuration (Figure 3). For an 8×8 array, at least 12 memory controllers are required to adequately spread the processor-to-memory traffic across enough links to avoid hot spots, as shown in Figure 3. Even if every tile had a memory controller attached resulting in a perfectly uniform random traffic distribution, a mesh would still have a maximum channel load that was 3× the average channel load. Clearly, not all the congestion is due to the many-to-few traffic patterns in the processor-to-memory links; some contention is due to routing.

5 Detailed Evaluation

After first narrowing the design space to a smaller number of configurations, we perform more detailed simulation to gain further insight into the issues surrounding memory controller placement. We use a cycle-accurate network simulator with synthetic traffic to evaluate the impact of alternative memory controller placement within the on-chip fabric and explore different routing algorithms.

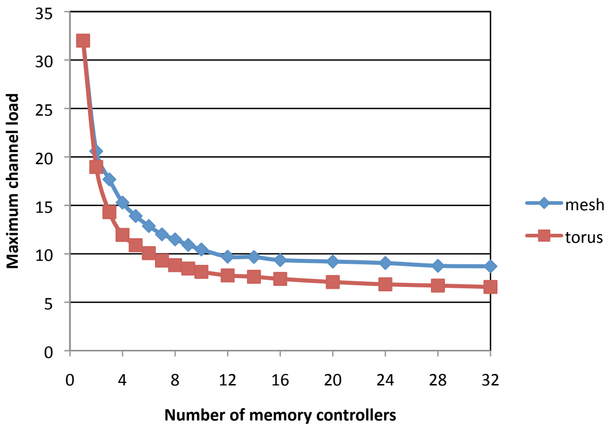


Figure 3. Maximum channel load versus number of memory controllers for an 8×8 array of tiles.

5.1 Routing algorithms

We evaluate the following memory controller placements described in the previous section which include row0_7, diamond, and row2_5 using synthetic traffic patterns. We start by evaluating an 8×8 mesh with well-understood dimension-order routing (DOR) algorithms, including: XY, YX, and XY-YX randomized routing.

- XY routing: dimension-ordered routing (DOR) where packets are first routed in the X dimension followed by the Y dimension.
- YX routing: DOR with where packets are first routed in the Y dimension followed by the X dimension.
- XY-YX routing (O1Turn [19]): at the source, the routing path to the destination is randomly selected, routing either XY DOR or YX DOR. This routing algorithm has been shown to be near-optimal for 2D mesh network [19].

From the link contention simulator, we determined that not all contention could be alleviated by smart memory controller placement. To further reduce contention we propose a new deterministic routing algorithm. Our proposed *class-based deterministic routing* (CDR) is simulated and compared against the above DOR algorithms.

- Class-based Deterministic Routing (CDR): takes advantage of both XY and YX routing but the path is determined by the message type: memory request packets use XY routing while memory reply packets take YX routing.

For both XY and YX routing, no additional virtual channels (VCs) are needed to break routing deadlock, but additional VCs are need to break protocol deadlock [4]. For XY-YX routing, in addition to VCs needed to break protocol deadlock, additional VCs are needed to break routing deadlock. However, for CDR routing, the

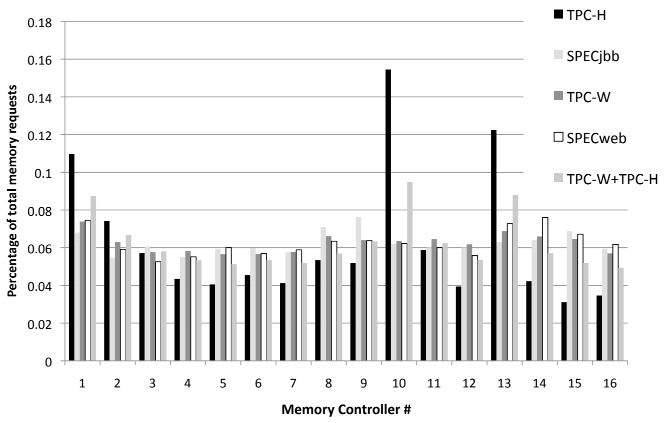


Figure 4. Distribution of Memory Controller access from several workloads

Table 2. Synthetic traffic simulation parameters

Parameters	Values
processors	64
memory controllers	16
router latency	1 cycle
inter-router wire latency	1 cycle
buffers	32 flits per input divided among the VCs
packet size	1 flit for request 4 flit for reply
virtual channels	2 for XY, YX, CDR 4 for XY-YX

VCs used to break routing deadlock can also be used to break protocol deadlock, thus the number of VCs needed is reduced compared to XY-YX routing.

5.2 Setup

To maximize the effective memory bandwidth, the traffic offered to each memory controller should be as close to uniform as possible. However, some applications may exhibit non-uniform traffic because of shared locks, for example. Thus, we evaluate alternative memory controller placement using both uniform random traffic, where each processor emits packets destined to a randomly selected memory controller, and *hot spot* traffic based on the distribution (as a percentage of total memory accesses) shown in Figure 4 with benchmarks and setup described in Section 5.4. Four out of five workloads, distribute accesses fairly uniformly across 16 memory controllers which validates the use of the uniform random traffic pattern; TPC-H's distribution generates *hot spot* traffic.

In the synthetic traffic evaluation, we use both open-loop simulation and closed-loop simulation. Open-loop simulation involves traditionally used metric of measuring latency vs. load to obtain network characteristics such as zero-load latency and the throughput. We also use closed-loop simulation where we measure the response of the network to compare overall performance [5]. For open loop simulation, packets were injected using a Bernoulli process. The simulator was warmed up under load without taking measurements until steady-state was reached. Then a sample of injected packets were taken during a measurement interval. Parameters used in both closed and open loop simulations can be found in Table 2.

To understand the impact of memory traffic, we separate the memory traffic into three different simulations in the open-loop evaluation using synthetic traffic patterns.

- Request traffic only (REQ) – processors only injected traffic destined for the memory controllers.
- Reply traffic only (REP) – only the memory controllers inject traffic into the network.
- Request and reply traffic – both request and reply traffic are injected into the network.

5.3 Detailed Simulation Results

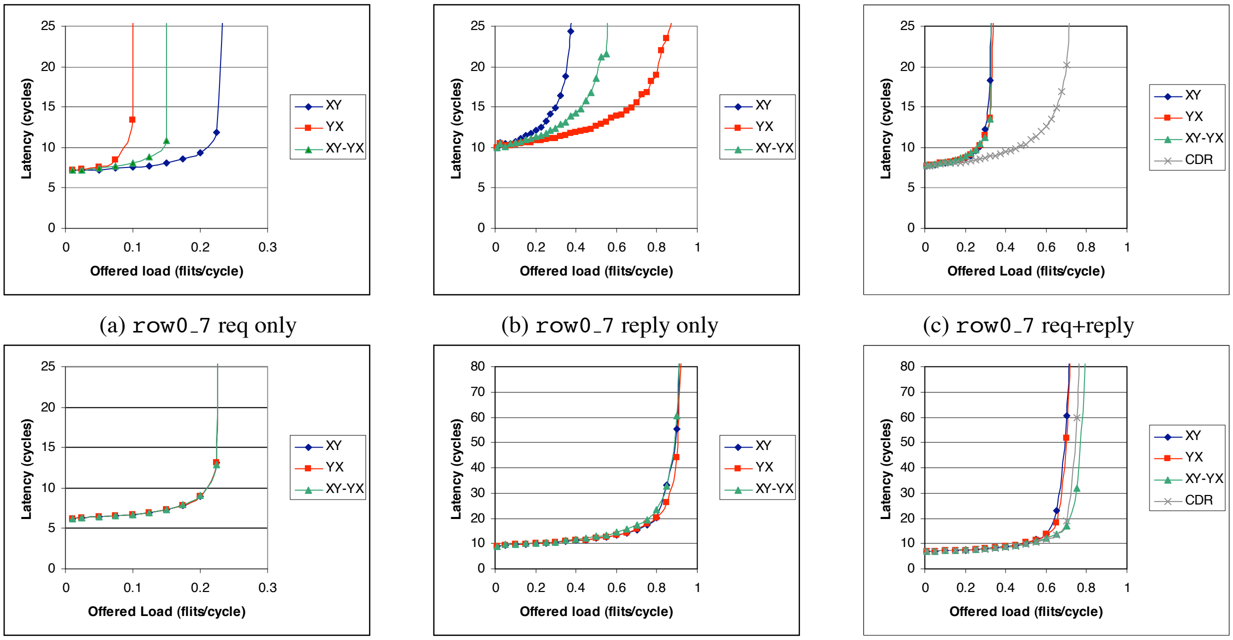
Using the results from Section 4, we discuss our detailed simulation results using synthetic traffic patterns for `row0_7` and `diamond` memory controller placements. As Table 1 shows, `diamond` and `diagonal X` perform about 33% better than the baseline `row0_7` placement. Due to better layout properties, we focus on the `diamond` as the optimal design over the `diagonal`.

5.3.1 `row0_7` Placement

The latency versus offered load curve is shown Figure 5 for the `row0_7` memory controller placement with uniform random (UR) traffic. For request traffic only, XY routing is sufficient and reaches maximum throughput of 0.25 [7]. However, YX routing performs poorly as it only achieves approximately half the throughput of XY routing. The use of randomization in routing (XY-YX) does not increase the performance and the achieved throughput is between XY and YX routing (Figure 5a). Since the `row0_7` placement distributes the memory controllers uniformly within the same row with, XY DOR routing load-balances the traffic to find the appropriate Y-dimension before sending the traffic to its destination. However, YX routing sends all the traffic initially to the two X coordinates where the memory controllers are located – thus, causing significant congestion on the channels in the X direction for rows 0 and 7 which contain the memory controllers.

The impact of memory controller placement and routing algorithm on memory traffic can be estimated by measuring the channel load (γ) since the network throughput (θ) is inversely propor-

[7] Since there are only 16 memory controllers and 64 processors, the maximum injection rate at each processor is 64/16 = 0.25.



(a) row0_7 req only

(b) row0_7 reply only

(c) row0_7 req+reply

(d) diamond req only

(e) diamond reply only

(f) diamond req+reply

Figure 5. Latency vs. offered load using uniform random traffic for (a,b,c) row0_7 placement and (d,e,f) diamond placement with with (a,d) request only traffic (b,e) reply only traffic and (c,f) both request and reply traffic combined.

tional to the worst-case (maximum) channel load [5]. The maximum channel load for an oblivious routing algorithm such as XY can be found by taking advantage of linearity of channel loading [23]. A block diagram of a $k \times k$ 2D mesh is shown in Figure 6 with γ_{X_i} (γ_{Y_i}) corresponding to the channel load of row (column) i. For request only traffic, with uniform traffic distribution and XY routing,

$$\begin{aligned} max(\gamma_{Xi}) &= \frac{k}{2} \times \frac{\lambda}{2} \\ max(\gamma_{Yi}) &= k(k-1) \times \frac{\lambda}{16} \end{aligned} \tag{1}$$

where λ is the injection rate of each processor. The $max(\gamma_{X_i})$ occurs in the middle or bisection of the network where $k/2$ nodes send $1/2$ (or $\lambda/2$) of their traffic to memory controllers located on the opposite half of the chip. The $max(\gamma_{Y_i})$ occurs at the top near the memory controllers as shown in Figure 6b with $k/(k-1)$ nodes sending traffic to the memory controller contributing to this channel. Since we assume uniform distribution among the 16 memory controllers, the actual load contributed from each processor will be $\lambda/16$. Thus, the throughput with XY routing is determined $max(\gamma_{Xi}, \gamma_{Yi})$.

$$\theta_{XY} = \frac{16}{k(k-1)\lambda} \tag{2}$$

With YX routing, the load on the channels will be

$$max(\gamma_{Xi}) = k\frac{k}{2} \times \frac{\lambda}{4}$$

$$max(\gamma_{Yi}) = (k-1)\frac{\lambda}{2} \tag{3}$$

For $i \neq 0$ or $k-1$, $\gamma_{Xi} = 0$ since all memory traffic is initially routed in the Y direction. Thus, the throughput with YX routing is determined by γ_X.

$$\theta_{YX} = \frac{8}{k \times k\lambda} \tag{4}$$

Based on Eqn 2 and Eqn 4, it can be seen that XY provides $2k/(k-1)$ increase in throughput compared to YX routing and with $k = 8$, XY results in ≈2.3 increase in throughput compared to YX as illustrated in Figure 5a.

With randomized XY-YX routing, XY routing is used for approximately 50% of the packets and the rest of the packets use YX routing. Thus, the channel load for XY-YX routing can be found as the following:

$$\begin{aligned} \gamma_Y(XY-YX) &= \frac{1}{2}\gamma_Y(XY) + \frac{1}{2}\gamma_Y(YX) \\ \gamma_X(XY-YX) &= \frac{1}{2}\gamma_X(XY) + \frac{1}{2}\gamma_X(YX) \end{aligned}$$

The actual channel load for XY-YX can be calculated by using Eqn 1 and Eqn 3 and it can be seen that XY-YX routing does not provide any performance benefits but achieves throughput that is between XY and YX routing as illustrated in Figure 5a.

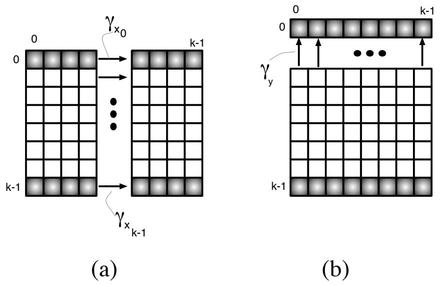


Figure 6. Channel load on a 2D mesh topology memory traffic with `row0_7` memory controller placement, illustrating the channel load on the (a) x-dimension and the (b) y-dimension.

For REP only traffic, the opposite is true in terms of the impact of routing. The use of XY routing creates a similar problem as the YX routing with REQ only traffic. Thus, YX routing provides better load-balancing for REP traffic, – i.e. transmits the packets to the appropriate row (or X dimension) and then, traverses the X dimension. Similar to REQ traffic, XY-YX does poorly.

When both the request and the reply traffic are combined (Figure 5c), both XY and YX routing perform similarly as the reply traffic creates a bottleneck for XY routing and request traffic creates a bottleneck for YX routing. However, the proposed CDR algorithm significantly outperforms other routing as it provides a nearly $2x$ increase in throughput. Both CDR and XY-YX routing take advantage of path diversity as some packets are routed XY and others are routed YX. However, by taking advantage of the characteristics of memory traffic (where $1/2$ the traffic will be request and the remaining $1/2$ is reply traffic), and the load-balanced traffic pattern, our deterministic routing based on the message type (CDR) load-balances all of channels to provide high throughput while adding randomization (XY-YX) achieves performance similar to XY or YX routing.

5.3.2 `diamond` Placement

With the `diamond` placement of the memory controllers, the different routing algorithms have very little impact on the overall performance as shown in Figures 5(d-f). Unlike `row0_7` placement which creates a *hot spot* row in the topology, the `diamond` placement distributes the memory controllers across all rows and columns. Thus, even with CDR, there is very little benefit in terms of latency or throughput (Figure 5f).

5.3.3 Closed-loop evaluation

We also evaluate the impact of routing algorithms and memory controller placement through closed-loop evaluation using a batch experiment to model the memory coherence traffic of a shared memory multiprocessor. Each processor executes a fixed number of remote memory operations (N) (e.g., requests to the memory controller) during the simulation and we measure the time required for all operations to complete. Each processor is allowed to have r outstanding requests before the processor needs to halt injection of packets into the network and wait until replies are received from the memory controller. This setup models the impact of MSHRs and increasing amount of memory level parallelism in a multiprocessor system; we evaluate the on-chip network using values of 4 and 16 for r and 1000 for N. Simulations showed that larger values for N do not change the trend in the comparisons.

Using CDR, we see that the underlying limitations of the memory controller placement are overcome; CDR results in significant improvements for the `row0_7` configuration as it balances the load to reduce the execution time by up to 45% with $r = 4$ and up to 56% with $r = 16$ (Figure 7a). With higher r, the network becomes more congested and thus, proper load-balancing through the use of CDR enables significant performance advantage. With the `diamond` placement and uniform random traffic (Figure 7b), the benefit of CDR is reduced but it still provides up to 9% improvement in performance. With the *hot spot* traffic, the benefit of CDR is reduced as it provides up to 22% improvement with the `row0_7` placement and up to 8% improvement with the `diamond` placement.

For the batch simulations, we also plot the distribution of completion time for each of the processors in Figure 8. With the `row0_7` placement, CDR provides not only higher performance in terms of lower completion time but also results is a much tighter distribution of completion – leading to a tighter variance. Tighter variance implies more fairness to access the memory controllers from all processors. Balancing the load through XY-YX and CDR with the `diamond` placement also results in a tighter distribution when compared to DOR.

5.4 Full System Simulation

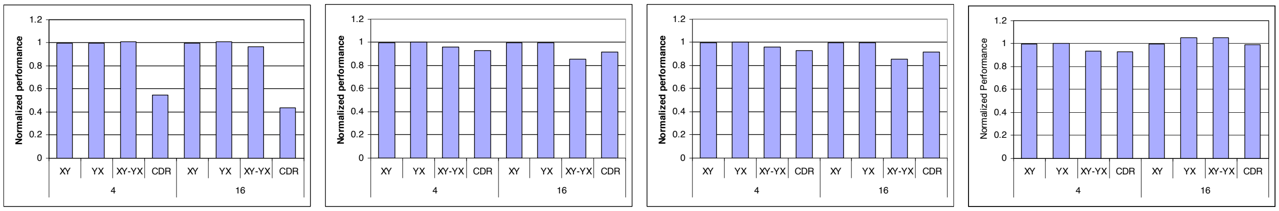
To gain additional insight, full system simulation [1,15] is used in conjunction with the above methods. Results are presented for the following commercial workloads: TPC-H, TPC-W [24], SPECweb99 and SPECjbb2000 [20]. Benchmark descriptions can be found in Table 3 with simulation configuration parameters listed in Table 4. In the link-contention and network-only simulators, only processor-to-memory and memory-to-processor traffic is considered. Full-system simulation includes additional traffic, e.g. cache-to-cache transfers that can interact with the memory-bound requests.

In order to evaluate large systems (8×8), we configure our simulation environment to support server consolidation workloads [6]. Each server workload runs inside of a virtual machine with a private address space; threads of the same virtual machine are scheduled in a 4×4 quadrant to maintain affinity. Memory requests from each virtual machine access all memory controllers on chip.

5.4.1 Full System Results

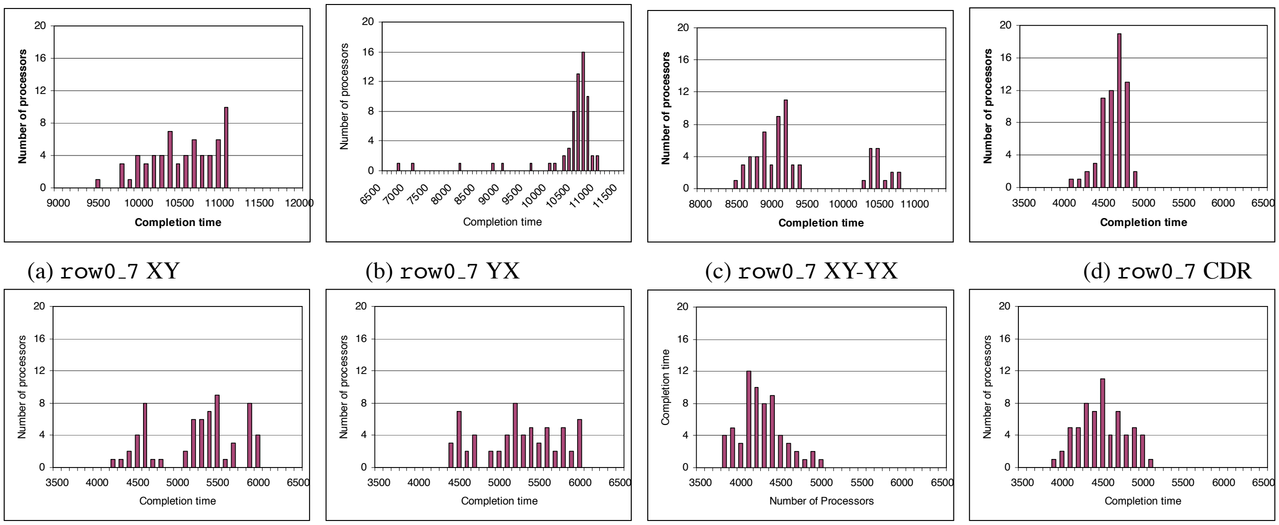
Full system simulation is used to validate results from the synthetic traffic simulations as well as provide inputs to the event driven network simulator. This simulation setup was used to generate the *hot spot* traffic used in the previous section.

Near-optimal placement can provide predictable and fair access to the memory controllers through the on-chip network. With the



(a)row0_7 uniform random (b) diamond uniform random (c) row0_7 hot spot (d) diamond hot spot

Figure 7. Batch experiment comparison with memory controllers placed at (a,c) row0_7 and (b,d) diamond using (a,b) uniform random traffic and (c,d) hot spot traffic. The x-axis label varies the routing algorithm as well as the r parameter.



(a) row0_7 XY (b) row0_7 YX (c) row0_7 XY-YX (d) row0_7 CDR

(e) diamond XY (f) diamond YX (g) diamond XY-YX (h) diamond CDR

Figure 8. Distribution of completion time for the batch workload. (a-d) row0_7 placement and (e-h) diamond placement with alternative routing algorithm – (a,e) XY routing, (b,f) YX routing, (c,g) XY-YX routing and (d,h) CDR.

closed-loop batch experiment results presented above, it is clear that placement can impact the performance distribution for synthetic workloads. In Figure 9, we show the average latency each processor observes to access the memory controllers versus the standard deviation across all processors with dimension-ordered XY routing. Down and to the left are the results with the lowest average latency and smallest standard deviation. Each workload is simulated with a diamond and a row0_7 configuration; we simulate four homogeneous server consolidation mixes and one heterogeneous mix of TPC-H and TPC-W. With the diamond configuration, each processor not only experiences lower latency, but there is less variation in the latencies observed by each processor. Choosing a good placement improves network latency to memory controllers by an average of 10% across the various workloads.

A lower standard deviation across the observed latencies between processors and memory controllers indicates that with a diamond configuration and simple routing, access to memory is both predictable and fair regardless of which processor core a thread is scheduled to execute on. The tight cluster of results for the diamond configuration indicates that an architect can size buffers and hardware structures to tolerate the average latency plus a small delta; these structures do not need to be sized to accommodate a long tail distribution of memory latencies as would be necessary with a poor placement.

5.5 Results Summary

To explore the design space, we use simulation techniques at multiple levels of abstraction starting with a fast link contention simulator that exhaustively simulates all possible permutations of memory controller placement, choosing the memory configuration that minimizes the maximum (worst-case) channel load. These

Table 3. Benchmark Descriptions

Benchmark	Description
TPC-H	TPC's Decision Support System Benchmark, IBM DB2 v6.1 running query 12 w/ 512MB database, 1GB of memory
SPECjbb	Standard java server workload utilizing 24 warehouses, executing 200 requests
TPC-W	TPC's Web e-Commerce Benchmark, DB Tier Browsing mix, 40 web transactions
SPECweb	Zeus Web Server 3.3.7 servicing 300 HTTP requests

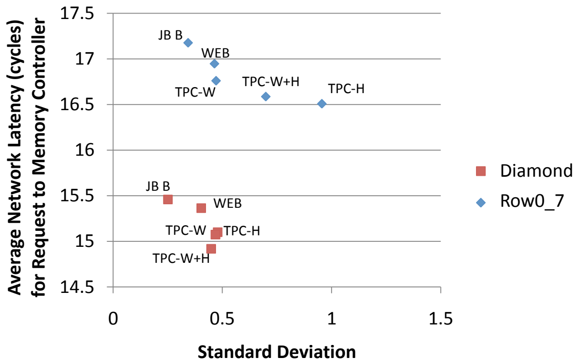


Figure 9. Standard Deviation vs Network Latency for Requests to Memory Controllers

Table 4. Full System Simulation Configuration

Parameters	Values
Processors	64 in-order cores
L1 I& D caches	16KB (2 way set associative) 1 cycle latency
L2 (Private)	64 KB (4 way set associative) 6 cycle latency
L3 (Shared)	8 MB (16 way set associative)
Memory Latency	150 cycles
Network Parameters	See Table 2

simulations narrowed down the field of candidates to those shown in Figure 2, with diamond placement performing the best using dimension-ordered routing because it was able to spread traffic across all rows and columns. We showed that the diamond placement has 33% less link contention compared to the baseline row0_7 placement used by Tilera [26]. We used two well-known simulation approaches: genetic algorithms [7] and randomized simulation to show our solutions for an 8×8 mesh could not be improved upon when exhaustive search proved to be computationally intractable.

We show that existing routing algorithms, including dimension-ordered routing (DOR) with either XY and YX as well as randomized XY-YX (O1Turn [19]) are not sufficient to load balance the processor-to-memory traffic on a 2D mesh. We show through detailed simulation that even a naive memory controller placement could be improved upon using better routing algorithm to avoid hot spots that would otherwise arise in the processor-to-memory traffic. Our proposed *class-based deterministic routing* (CDR – pronounced "cedar") routes *request* packets using XY dimension-ordered routing, and *reply* packets route using YX dimension-ordered routing. We show that implementing the baseline row0_7 placement with CDR routing can improve performance by up to 56% for uniform random (UR) traffic, and 22% improvement with *hot spot* memory traffic. With diamond placement, we see a more modest 8% improvement from the CDR routing algorithm because the diamond placement nicely spreads the offered load among the rows and columns of the mesh leaving less room for improvement from the routing algorithm.

Our full system simulation results show that the diamond placement has significantly lower *variance* than the row0_7 placement, as shown in Figure 9. This lower variance provides more predictable latency-bandwidth characteristics in the on-chip network regardless of which processor core the application is using. We also observe a 10-15% improvement in network latency with the diamond placement.

In this work we propose two complimentary solutions to address latency and bandwidth problems for on-chip access to memory ports. The first solution improves performance by relocating memory ports. Implementing this solution comes at no extra cost or power consumption for the architect. However, if relocation is not feasible, then an alternative solution of implementing CDR would improve latency and throughput for processor-to-memory and memory-to-processor traffic. CDR is a low cost routing algorithm. Due to its deterministic nature, only two virtual channels are need to break protocol deadlock (same as XY and YX). Combining these two techniques results in the best overall performance.

6 Related Work

In this work, we advocate for intelligent memory controller placement and routing of processor-memory traffic in on-chip networks. Both intelligent placement (such as the diamond) and CDR improve on-chip network load balancing which effectively increases fairness to the memory ports. In this section, we explore related work in the areas of quality of service and fairness as well on-chip placements solutions.

Recent work in quality of service (QoS) focuses on spreading accesses to uniformly utilize the memory controllers; efficiently distributing memory references will reduce interconnect pressure near the memory controllers as well. Recent NoC innovations [14] have explored techniques to provide quality of service within the on-chip network. This work provides quality of service for traffic with a single hot spot but does not address the impact on network performance when the placement of hot spot(s) can be chosen at design time. Our work shows that certain latency and bandwidth bottlenecks in the network can be avoided to a certain extent through near-optimal placement of memory controllers assuming memory controllers are likely to represent the hot spots in a large scale on-chip network going forward. Our work considers the impact that network hot spots have on each other due to proximity and through the use of TPC-H hot spot traffic with three memory controller hot spots.

Proposals to provide quality of service at the memory controllers [16–18] to date have not considered the impact of memory controller placement or how the on-chip network delivers those requests but rather focus on fair arbitration between different requests once they arrive at the memory controller. This work on fair memory controllers compliments our work on optimizing the on-chip network for memory traffic.

Significant research in the system-on-chip and application specific design communities addresses the challenge of how best to map tasks to physical cores on-chip [11,21]. Application specific designs are unique from general purpose ones in that communication patterns are known a priori and can be specifically targeted based on communication graphs. In the application-specific domain, research has been done to find the optimal mapping of tasks to cores and the optimal static routes between cores to achieve bandwidth and latency requirements. Due to the embedded nature of many application-specific designs, these algorithms often use energy minimization as their primary objective function. Work by Hung et al. [12] uses a genetic algorithm to minimize thermal hot spots through optimized IP placement. General purpose CMPs see less predictable traffic patterns; we model two traffic patterns derived from real workload behavior.

Significant research has focused on the impact of different memory technologies and their respective trade-offs in providing adequate off-chip latency and bandwidth [3,8]. Our work takes an alternative view of the system by considering the on-chip interconnect bandwidth and latency to the memory controllers.

In this work, we propose CDR, a simple deterministic routing algorithm that load balances processor-memory traffic. Similarly, an adaptive routing algorithm could be used to balance load; however, the use of adaptive routing in on-chip networks will significantly increase the complexity (i.e., increase in number of VCs, router pipeline latency, etc) such that the overall benefit will be minimal. Prior work [19] showed that if the pipeline delay of adaptive routing is considered, their O1Turn routing algorithm outperformed adaptive routing. Furthermore, adaptive routing can only be used for response packets – deterministic routes are necessary to preserve order of the read/write request packets to the memory controllers. Response packet could be adaptively routed; however, a 2D mesh will have limited path diversity and be subject to turn model constraints [9]. As a result, it will have only a few minimal adaptive candidates.

7 Conclusion

Aggressive many-core designs based on tiled microarchitectures will have dozens or hundreds of processing cores, but packaging constraints (i.e. the number of pins available) will limit the number of memory controllers to a small fraction of the processing cores. This paper explores how the *location* of the memory controllers within the on-chip fabric can play a central role in the performance of memory-intensive applications.

Intelligent placement can reduce maximum channel load by 33% with a `diamond` configuration, compared to the baseline `row0_7` configuration that is used by Tilera [26] and Intel [25]. We further improve upon this result by introducing the *class-based deterministic* (CDR) routing algorithm, which routes request and reply traffic differently to avoid hot spots introduced by the memory controllers. The CDR algorithm improves performance by 56% for uniform random traffic compared to the baseline `row0_7` placement and 8% with the `diamond` placement. Full system simulation further validates that the `diamond` placement reduces interconnect latency by an average of 10% for real workloads.

The small number of memory ports and memory controllers relative to the number of on-chip cores opens up a rich design space to optimize latency and bandwidth characteristics of the on-chip network. We demonstrate significant potential improvements in performance and predictability through an exploration of this design space.

References

[1] H. Cain, K. Lepak, B. Schwarz, and M. H. Lipasti. Precise and accurate processor simulation. In *Workshop On Computer Architecture Evaluation using Commercial Workloads*, 2002.

[2] P. Conway and B. Hughes. The AMD opteron northbridge architecture. *Micro, IEEE*, 27(2):10–21, March-April 2007.

[3] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary dram architectures. In *Proceedings of the International Symposium on Computer Architecture*, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[4] W. J. Dally. Virtual-channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, 1992.

[5] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, San Francisco, CA, 2004.

[6] N. Enright Jerger, D. Vantrease, and M. H. Lipasti. An evaluation of server consolidation workloads for multi-core designs. In *IEEE International Symposium on Workload Characterization*, 2007.

[7] E. Falkenauer. *Genetic Algorithms and Grouping Problems*. John Wiley & Sons Ltd., Chichester, England, 1997.

[8] B. Ganesh, A. Jaleel, D. Wang, and B. Jacob. Fully-buffered dimm memory architectures: Understanding mechanisms, overheads and scaling. In *International Symposium on High-Performance Computer Architecture*, pages 109–120, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[9] C. J. Glass and L. M. Ni. The turn model for adaptive routing. In *Proceedings of the International Symposium on Computer Architecture*, 1992.

[10] M. Horowitz, C.-K. K. Yang, and S. Sidiropoulos. High-Speed Electrical Signaling: Overview and Limitations. *IEEE Micro*, 18(1):12–24, 1998.

[11] J. Hu and R. Marculescu. Engergy-aware mapping for tile-based NoC architecture under performance constraints. In *Proceedings of ASP-DAC*, 2003.

[12] W. Hung, C. Addo-Quaye, T. Theocharides, Y. Xie, N. Vijaykrishnan, and M. J. Irwin. Thermal-aware IP virtualization and placement for networks-on-chip architecture. In *Proceedings of the International Conference on Computer Design*, 2004.

[13] Intel Tera-scale Computing Research Program: Teraflop Research Chip. http://techresearch.intel.com/articles/tera-scale/1449.htm.

[14] J. Lee, M. C. Ng, and K. Asanovic. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *Proceedings of the International Symposium on Computer Architecture*, pages 89–100, June 2008.

[15] K. M. Lepak, H. W. Cain, and M. H. Lipasti. Redeeming IPC as a performance metric for multithreaded programs. In *Proceedings of the 12th PACT*, pages 232–243, 2003.

[16] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *International Symposium on Microarchitecture (MICRO-40)*, December 2007.

[17] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *International Symposium on Computer Architecture (ISCA-35)*, June 2008.

[18] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 208–222, Washington, DC, USA, 2006. IEEE Computer Society.

[19] D. Seo, A. Ali, W.-T. Lim, N. Rafique, and M. Thottethodi. Near-optimal worst-case throughput routing for two-dimensional mesh networks. In *Proc. of the International Symposium on Computer Architecture (ISCA)*, pages 432–443, 2005.

[20] SPEC. SPEC benchmarks. http://www.spec.org.

[21] K. Srinivasan and K. Chatha. A technique for low energy mapping and rougin in network-on-chip architectures. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2005.

[22] Tilera Corporation. http://www.tilera.com.

[23] B. Towles and W. J. Dally. Worst-case traffic for oblivious routing functions. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 1–8, 2002.

[24] TPC. TPC benchmarks. http://www.tpc.org.

[25] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS. *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 98–589, Feb. 2007.

[26] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal. On-chip interconnection architecture of the tile processor. *Micro, IEEE*, 27(5):15–31, Sept.-Oct. 2007.