Reducing Energy and Increasing Performance with Traffic Optimization in Many-core Systems

George B. P. Bezerra
Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131
gbezerra@cs.unm.edu

Stephanie Forrest
Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131
forrest@cs.unm.edu

Payman Zarkesh-Ha
Dept. of Electrical and
Computer Engineering
University of New Mexico
Albuquerque, NM 87131
payman@ece.unm.edu

ABSTRACT

As the number of cores on a die continues to increase, it is necessary to optimize the traffic patterns of applications in order to minimize power consumption and maximize performance. We present a new approach for traffic optimization in many-core systems, which targets communication locality and load-balancing. Our approach works by mapping memory blocks to physical locations on the chip that are close to cores that access them, and by enforcing load balance by limiting the number of blocks mapped to each location. Communication locality reduces the average distance traveled by packets, which minimizes power and increases performance. Load-balancing avoids hotspots and improves cache utilization. Rather than treating every application in the same way, our method uses available information to produce mappings that are specially tuned for individual applications. Simulations performed on a 64-core system show a reduction in dynamic energy consumption of up to 81.6% and of 45.5% on average, and gains in performance of up to 13.2% on scientific benchmarks.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design aids—*simulation*; C.4 [**Performance of Systems**]: Modeling techniques

General Terms

Design, Performance, Theory

Keywords

Traffic optimization, memory-block mapping, communication locality, load-balancing, many-core, communication graph, non-uniform cache access

1. INTRODUCTION

As microprocessor technology advances towards parallel designs by increasing the number of cores on a die, optimization of the communication patterns between cores becomes critically important. With a larger number of cores the traffic volume increases, which leads to increased power consumption. High traffic volumes also consume more network bandwidth, causing contention and increasing message latency. More cores on a die also increase the average number of hops between source and destination of a message, which increases both latency and power.

In many-core Chip Multi-Processors (CMP), communication patterns can be improved by optimizing the mapping of memory blocks to caches. Because slices of cache are distributed among nodes, the latency in accessing a block varies with the distance between the requester and the corresponding directory or home node of the block. Considerable work exists in the literature targeting block mapping in Non-Uniform Cache Access (NUCA) architectures. In existing approaches, alternative cache management policies have been proposed that employ a combination of shared and private schemes [17, 14, 5, 6, 10], block migration [1, 12], or both [8]. All these works provide dynamic solutions to the mapping problem, which are useful when limited knowledge is available about the workload, and every application must be treated in the same way. In contrast, static solutions are more desirable in scenarios where information about applications is known in advance, such as High-Performance Computing (HPC). In this case, the mapping can be fine-tuned for individual applications, yielding increased gains.

In NUCA architectures, the mapping of memory blocks also has a large impact on energy consumption. In a remote cache access, the dynamic energy of a message increases proportionally to the distance between its source and destination. Consequently, mappings that minimize the average access distance could lead to significant energy reduction. In spite of the importance of energy efficiency to modern computer architecture design [9], only [5] analyzes energy consumption, reporting improvements of up to 13.9%, and only [1, 12] consider distance explicitly in their formulation. Most of the literature focuses on relatively small core counts, ranging from 4 to 8 in [17, 14, 5, 6, 12] and 16 in [1, 8, 10].

We present a new approach for traffic optimization in many-core systems that increases performance and energy efficiency by improving traffic locality and load balance. Our approach works by mapping blocks of memory to physical locations on the chip that are close to cores that access them. This method minimizes the average distance trav-

eled by packets and, consequently, energy and latency. To prevent hot-spots and enforce load balance, we constrain the maximum number of blocks assigned to each core. The simulation results for a 64-core system show a reduction in dynamic energy consumption of up to 81.6% and of 45.5% on average, and improvements in performance of up to 13.2% on benchmark applications.

The proposed approach was designed for environments in which one application is running at a time and the properties of the workload are well known. In order to automatically collect information about an application, a trace of all messages in the system for a given input set is generated. From the trace we extract the communication graph, which is used to produce the mapping. For most applications, the generated mappings are robust and can be reused for new input data with similar performance. Our method does not depend on the cache-coherence protocol used or on whether caches are shared or private.

This paper is organized as follows. Section 2 reviews the traditional block mapping for many-core architectures and its consequences in terms of traffic patterns. Section 3 describes the materials and methods used, such as experimental setup and benchmark workloads. Section 4 explains the technique used for optimizing traffic and presents the results. A discussion of the results and the methodology proposed is given in section 5, and section 6 concludes the paper.

2. TRAFFIC PATTERNS IN MANY-CORE

Many-core systems rely on a directory protocol to monitor the state and location of memory blocks in cache [13].[1] In the directory protocol, each core is the *directory node* for a subset of the memory addresses and is responsible for managing the information about those addresses. In shared-cache systems, the directory node (also called *home node*) is also the unique host of the address in cache, as duplicate cache lines are not allowed in this case. For simplicity, in the following we will refer to private caches, but the discussion applies to both shared and private schemes.

When a processor needs to access an address that is not in cache or to write to a memory location, communication with the directory takes place in order to, for example, fetch the block, or update its state in the directory. The specifics of the communication process depends on several factors, such as the cache-coherence protocol, the state of the cache block, and whether last-level caches are shared or private. Nonetheless, memory operations require frequent communication with the directory, no matter how the details of the communication process are implemented.

In the standard hardware implementation, blocks of physical memory are mapped uniformly among all directory nodes. The directory node D of a given address A is determined by $D = A \bmod N$ [7], where N is the number of nodes, as shown in figure 1 for a small system with only four cores. The advantages of this method are two-fold. First, the directory node of an address can be found using a `mod` computation, which requires simple hardware. Second, because blocks are uniformly distributed among all nodes, it prevents hotspots and enforces load balance.

[1]The directory-based approach is a scalable alternative to the snooping protocol, in which no centralized information about a cache block is kept. Instead, this information is broadcast to all cores, which consumes excessive network bandwidth and power for large core-counts.

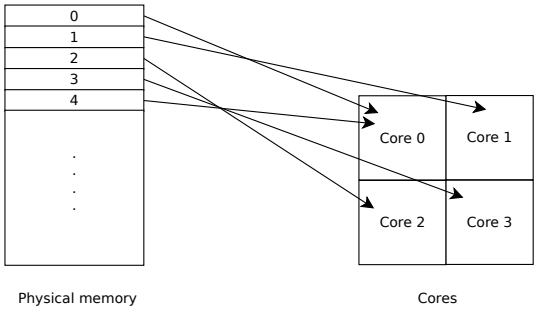

Figure 1: Blocks of physical memory are assigned to directory nodes present on the cores. The blocks are uniformly distributed in an interleaved manner.

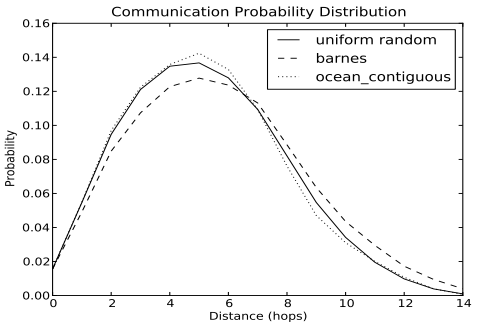

Figure 2: Communication probability distribution for two benchmark applications and uniform random traffic.

Although the above approach has advantages, it is suboptimal because it ignores any underlying structure in the communication pattern of the application. Since blocks are uniformly distributed, a core has on average an equal probability of communicating with any directory node. This results in traffic that is uniform and random. To illustrate this point, figure 2 shows the communication pattern of two applications running on a 64-core machine and how they compare to purely uniform-random traffic. The graph corresponds to the Communication Probability Distribution (CPD) [2], that is, the probability that a packet will travel a certain number of hops for a given application. The farther packets need to travel, the less communication locality. [2]

Uniform random traffic is undesirable because it does not exploit communication locality. The location to which a memory block is mapped is independent of the distance to the cores that are most likely to use it. As a result, packets have to travel longer distances, which increases latency and consumes more network bandwidth and power. The approach proposed in this paper is based on mapping blocks

[2]Uniform random traffic does not have a uniform distribution in figure 2 because of the 2D mesh topology. A node may send packets to other nodes with uniform probability, but each node has at most 4 neighbors 1-hop away, 8 neighbors 2-hops away, and so on, until the number of neighbors decreases as the boundaries of the mesh are reached. Therefore, the distribution reflects the average number of neighbors at a certain distance away from the node.

to directory nodes that are close to where they are most frequently accessed, thereby reducing power and increasing performance.

3. MATERIALS AND METHODS

Full-system simulations were performed with the Graphite simulator [15]. Graphite is an open-source parallel multi-core simulator developed by the MIT. In order to achieve high-performance, Graphite is not cycle-accurate, but it achieves high accuracy by using sophisticated synchronization techniques.

The simulations were performed with in-order, single issue cores. The L1-I and L1-D caches are 4-way set-associative with 32 KB cache-capacity, and 64-byte blocks. The L2-cache is 8-way set-associative with 512 KB capacity, and 64-byte blocks. Both caches are private. The directory used for cache coherence is $Dir_N NB$ [4], that is, full-map with no broadcast, and blocks were assigned to directories at the cache-line granularity. The directory cache is set-associative with 16384 entries. We used the MESI cache-coherence protocol.

Energy consumption in the network on-chip was measured with Orion-2 [11], which is included with Graphite. Each hop on the 2D-mesh network takes one cycle, and dimension-order routing was used as the routing algorithm. All simulations were performed on a 64-core system. The number of threads in each application is the same as the number of cores. As threads are spawned, the simulator assigns each new thread to the next available core, in order. Only one thread is assigned to each core.

The parallel applications used in the simulations are from the Splash-2 benchmark [16]. The input of most applications is defined by a random number generator. To produce new inputs, which were used in the analysis in section 4.4, we varied the seed of the generator. An exception is the `ocean` application, which has no input. In this case, we introduced variation by changing the parameters of the application, such as the error tolerance.

Because in Graphite the state of the simulator depends on the state of the host system, simulations are not deterministic and may vary slightly from run to run. A full statistical analysis of all applications would be impractical due to the high-cost of full-system simulations. Instead, we report the deviations obtained for one application. For `FFT`, a standard deviation of 0.08% in energy, 0.03% in runtime, and 0.2% in latency was obtained over 10 runs of the application.

In order to allow for a consistent mapping of memory blocks, it is convenient to have the address space of applications laid out deterministically. To ensure a deterministic layout, we disabled Address Space Layout Randomization (ASLR) in our experiments.

4. TRAFFIC OPTIMIZATION

This section describes the proposed approach for traffic optimization using memory-block mapping. The method for producing the optimized mapping and testing the results is divided into four steps:

1. Generate a communication graph from application traces obtained with the traditional block mapping (described in section 2);
2. Find the optimized block positions based on the communication graph;
3. Run the application with the optimized block mapping and measure the changes in performance; and
4. Run the application again with different input data.

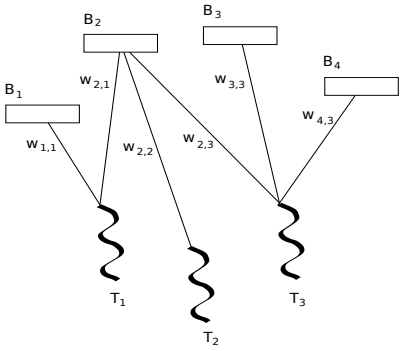

Figure 3: Simple illustration of a communication graph. There is no direct communication between threads or between blocks. All communication is between threads and blocks.

4.1 Network of blocks and threads

Communication in a shared-memory system can be modeled as a network of blocks and threads, in which links correspond to messages exchanged between them. Every message to and from a directory or home node is associated with an address that determines the memory block, and a core, which defines the thread. There is no direct communication between threads or between blocks. Following the above description, we define a communication graph $G = \{V, E\}$ as a weighted, undirected bipartite graph in which each vertex corresponds to a block (B) or a thread (T), and edges link blocks to threads, where the weight $w_{i,j}$ is the total communication (in bytes) between block B_i and thread T_j. Figure 3 depicts a schematic representation of a communication graph.

For each application, we extracted its communication graph by running the application with the standard block mapping described in section 2, and generating a trace of all the messages sent on the network. From the trace we created the communication graph by representing each message as an edge, where the size of the message is the weight of the edge. Multiple messages between the same source and destination do not create a new edge, but are used to increase the weight of an existing edge. Once the graph is constructed, it does not matter what cache-coherence protocol the machine uses, or whether caches are shared of private. Only the communication graph will be used in the optimization process.

An analysis of the network structure of graph G could reveal relevant information about an application. One important metric is the degree of a block, i.e., the number of edges connected to the block in the graph, which is related to its level of sharing. A block that has degree N, where N is the total number of threads, is shared by all threads. In this case, not much optimization can be done because the block has no affinity to any specific thread (this is not necessarily true if the weights differ considerably). The best location for such blocks would be in the central nodes of the mesh. On the other hand, if the block degree is 1, then the block is private, i.e., it is only accessed by one thread. This

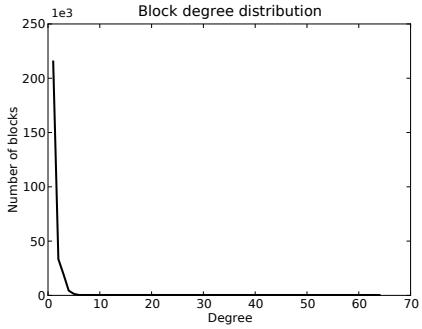

Figure 4: Degree distribution of blocks for the `ocean_contiguous` application.

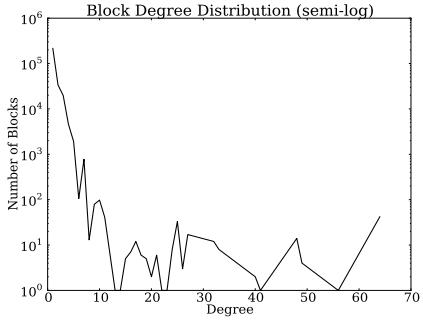

Figure 5: Semi-log plot of the block degree distribution for the `ocean_contiguous` application.

is the best case, because the block can be assigned to the directory at the core in which the thread is running.

Figure 4 gives an example of a typical degree distribution of the memory blocks of an application running on a 64-core machine. The figure indicates that the great majority of blocks have very small degree and, therefore, there is potential for optimization in this application. The figure may be misleading in that no blocks seem to exist with large degree. This happens because the number of blocks span several orders of magnitude. Figure 5 shows the same distribution in a semi-log plot, which in this case gives a clearer picture of the entire distribution. Notice a peak in the distribution when the degree is 64, corresponding to blocks that are shared by all threads.

The block degree distribution should not be interpreted as the only property influencing optimization. Several other factors also play an important role. For example, blocks can have different strengths (i.e., the sum of the weights of all links connected to a block) and in general the higher the degree the higher the strength of the block. The positions of the threads on the chip also have an impact on the end result. If a block has small degree but the threads it is connected to are located at a long distance apart from each other, the gains from optimization will be limited.

4.2 Traffic locality optimization

In order to improve traffic locality, blocks must be mapped as close as possible to cores that are running the threads that access them. In the current analysis, the positions of the threads are pre-defined, so the algorithm used to find the

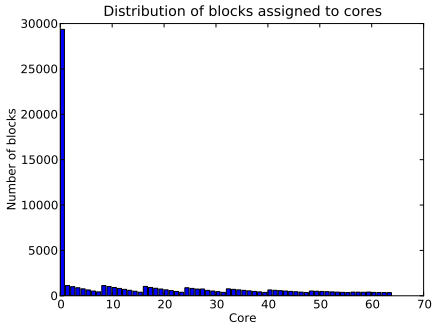

Figure 6: Distribution of blocks assigned to each core for `FFT`.

optimal position of a block is simple. It consists of checking all the N possible locations on the chip and choosing the one that minimizes the communication cost, which is defined as the sum of the weighted distances of all the links of that block. Equation 1 shows the formula for the position of a block B_i.

$$Pos(B_i) = \min_{1 \le p \le N} Cost(B_i, p) : \sum_{j=1}^{N} w_{i,j} \cdot D(p, T_j), \qquad (1)$$

where p is the candidate position for block B_i, D is the distance function, and $w_{i,j} = 0$ if there is no edge between B_i and T_j in the graph G.

By applying the above formula, large reductions on the average distance of a block to its threads can be achieved. However, this simple allocation of blocks does not work in practice, because some cores are likely to be assigned many more blocks than others. Figure 6 shows a histogram of the number of blocks assigned to each core according to the mapping described in equation 1, in which case the thread that runs the `main()` function is assigned a disproportionate number of blocks. The consequences of such uneven mapping would be catastrophic. Each node has a limited cache capacity, and when this capacity is reached it needs to make room for new blocks by replacing old ones, a process called eviction. This slows down the system and increases the number of messages.

In fact, running this block mapping for the `FFT` application, for example, increases the number of messages in the network by 23%, which outweighs the improvement in communication distance. To solve this issue, the next section discusses how the load can be balanced across cores.

4.3 Load-balancing

To avoid uneven mapping that could lead to hotspots, we add a penalty to the number of blocks assigned to a core in the cost function defined by equation 1. The new formula is shown below:

$$Pos(B_i) = \min_{1 \le p \le N} Cost(B_i, p) : \sum_{j=1}^{N} \frac{w_{i,j} \cdot D(p, T_j)}{(Max - Blocks(p))}, \qquad (2)$$

where Max is the maximum number of blocks allowed on a core, defined as the total number of blocks divided by N, and $Blocks(p)$ is the number of blocks currently mapped to the core at position p. As more blocks are mapped to a

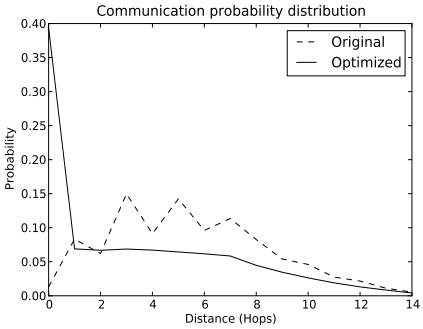

Figure 7: Communication probability distribution of `FFT` before and after optimization.

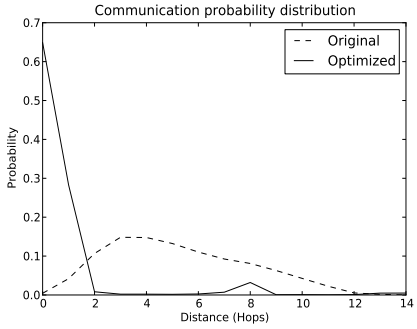

Figure 8: Communication probability distribution of `ocean_non_contiguous` before and after optimization.

given core, the cost function will increase. This ensures that all cores will have the same number of blocks mapped onto them.

One consequence of this modification is that the order in which blocks are assigned to cores affects the end result. We dealt with this issue by sorting the blocks by degree and mapping the high-degree blocks first.

The results from applying the block mapping defined in equation 2 are shown in table 1. The table contains the percentage reduction in energy consumption, runtime, and average packet latency for the applications when running the optimized mapping. The results show a large improvement in energy consumption of up to 81.6% and a moderate improvement in runtime of up to 13.2%. Packet latency also was greatly reduced by up to 77.1%. For all applications, improvements were obtained in all three metrics.

Figure 7 and 8 show the traffic patterns before and after optimization for `FFT` and `ocean_non_contiguous`, respectively. Notice the dramatic change in the traffic patterns towards increased communication locality. This reduction in the average distance traveled by packets is the cause of improved energy, runtime, and latency. In the figures, a distance of zero hop corresponds to blocks that are placed at the same location as the threads that access them.

4.4 Varying the input

The results presented in section 4.3 work as a proof-of-concept in showing that optimizing the location of blocks on the chip is a promising technique. However, to be useful in practice the method should be generalizable to new inputs. In this section, we used the mappings obtained in section 4.3 to run the applications with a different input set of the same size. We then compared the optimized performance with that of the original mapping. The results are shown in table 2.

From table 2, the gains of the optimized mapping are still large and do not change considerably when new input data is used. This means that the communication graph is robust to variations in the input for most applications. For some applications, such as `FMM` (which models an n-body problem) and `radix` (a distributed sorting algorithm), the mapping decayed in performance, though the overall improvements are still significant. This happens because part of the memory accesses of these applications depends on the values of the input data. Consequently, the communication graph could change with new inputs, leading to smaller improvements. Interestingly, in some cases the metrics actually improved with the new input set. For example, `water_spatial` obtained 3.9% improvement in runtime in table 2, compared to only 0.1% in table 1.

The results in this section demonstrate that the method is generalizable across inputs and therefore could be used in practice to perform traffic optimization. Once the block mapping of an application is produced, it can be reused multiple times for new input data with no need to be relearned.

5. DISCUSSION

The analysis above shows that the proposed optimization method is a promising technique with good capacity of generalization and practical applicability. The results show a large improvement in energy consumption and latency for most applications, while only modest improvements in performance were obtained. The fact that reduction in energy and latency is much greater than that of runtime indicates that network bandwidth is not a bottleneck to performance for the analyzed system and applications. It would be interesting to verify whether that holds true for larger systems, in which the traffic volume is higher.

Tables 1 and 2 suggest that energy and runtime improvements are not strongly correlated. For example `FFT` obtained good improvement in energy and poor improvement in runtime. On the other hand, `radix` had a smaller energy improvement in table 2, but the second largest reduction in runtime. More studies need to be performed to understand why these differences arise. It is possible that the network structure of the communication graph could be used to predict improvements from optimization.

In our analysis, we used cache-line granularity for constructing the communication graph and performing the mappings. The reason for this choice is that the cache-line represents the smallest possible granularity and, therefore, allows for maximum locality exploitation. Using this method in practice, however, would require special hardware that is able to implement arbitrary block mappings (instead of the usual approach of identifying directories using a `mod` computation, as explained in section 2). Alternatively, it is possible to use page-level directory granularity, in which case the operating system would handle the mapping of pages to directory nodes using page tables. This would require no extra hardware and very little overhead by the OS. OS management of L2-caches at the page-level granularity has been proposed by [7]. They showed that the operating system can be used to implement several L2-cache policies with

Table 1: Percent improvement in energy, runtime, and latency for the benchmark applications after applying the optimized block mapping. The input data are the same as those used to generate the communication graph.

Application	Energy (%)	Runtime (%)	Latency (%)
barnes	36.2	2.0	23.8
FFT	44.7	0.6	48.7
FMM	19.2	1.1	11.4
LU_contiguous	26.4	0.3	16.1
LU_non_contiguous	52.8	2.2	48.6
ocean_contiguous	75.7	8.1	68.7
ocean_non_contiguous	81.6	13.2	77.1
water_nsquared	26.7	0.3	12.7
water_spatial	25.1	0.1	15.2
radix	56.6	11.8	49.1

Table 2: Percent improvement in energy, runtime, and latency for the benchmark applications after applying the optimized block mapping. The input data are different from those used to generate the communication graph.

Application	Energy (%)	Runtime (%)	Latency (%)
barnes	35.2	1.1	23.8
FFT	44.4	0.5	48.7
FMM	8.1	2.1	3.0
LU_contiguous	26.3	0.3	26.4
LU_non_contiguous	53.9	1.9	48.9
ocean_contiguous	74.0	3.3	68.7
ocean_non_contiguous	80.5	11.2	76.7
water_nsquared	26.7	0.3	12.6
water_spatial	24.8	3.9	16.0
radix	23.5	9.7	21.3

little overhead, and reported good results in performance improvement, although they did not analyze energy consumption. We leave the exploitation of page-level granularity in traffic optimization for future work.

The methodology presented here was designed for static environments, in which applications can be fine-tuned before they are run for multiple iterations, such as in High-Performance Computing (HPC). However, we believe that dynamic optimization is also possible using the same principles, that is, by maximizing traffic locality and load balance. We plan to apply our method for traffic optimization in dynamic environments in the future.

One limitation of our approach is that once a mapping is generated it will only work for inputs of the same size. Because of the way virtual memory is allocated to a process, if the size of the input data changes, the block addresses will no longer be aligned and the quality of the mapping may decay. There are compiler techniques that can be used to circumvent this problem [3], which could be used to extend our work to handle inputs of varying size.

6. CONCLUSIONS

This paper proposed a new method for traffic optimization in many-core systems based on memory-block mapping. The proposed approach works by maximizing traffic locality while maintaining good load balance, and produces mappings that are tuned for individual applications. The results show a large reduction in energy consumption and modest, but significant, improvements in performance for benchmark applications. The technique also has capacity of generalization, which proves its practical applicability. We believe the proposed method for traffic optimization will be a useful technique to increasing the scalability of many-core systems, by reducing power consumption and increasing performance.

7. ACKNOWLEDGEMENTS

S. Forrest acknowledges the support of the National Science Foundation (grants CCF 0621900, CCR-0331580, SHF-0905236), Air Force Office of Scientific Research MURI grant FA9550-07-1-0532, and the Santa Fe Institute. P. Zarkesh-Ha acknowledges the support of the US Department of Energy, Office of Science, under Grant DE-SC0002113. We thank the members of the Adaptive Computation Laboratory and Steve Hofmeyr for insightful discussions and feedback.

8. REFERENCES

[1] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 250–261, 2009.

[2] G. Bezerra, S. Forrest, M. Forrest, A. Davis, and P. Zarkesh-Ha. Modeling noc traffic locality and energy consumption with rent's communication probability distribution. In *Proceedings of the 12th ACM/IEEE international workshop on System level interconnect prediction*, pages 3–8, 2010.

[3] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. *ACM SIGPLAN Notices*, 33(11):139–149, 1998.

[4] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, 2002.

[5] M. Chaudhuri. PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 227–238. IEEE, 2009.

[6] Z. Chishti, M. Powell, and T. Vijaykumar. Optimizing replication, communication, and capacity allocation in CMPs. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 357–368. IEEE, 2005.

[7] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, 2006.

[8] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive nuca: near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 184–195, 2009.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007.

[10] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. Keckler. A NUCA substrate for flexible CMP cache sharing. *IEEE transactions on parallel and distributed systems*, pages 1028–1040, 2007.

[11] A. Kahng, B. Li, L. Peh, and K. Samadi. Orion 2.0: A fast and accurate NOC power and area model for early-stage design space exploration. In *Design, Automation, and Test in Europe*, pages 423–428, 2009.

[12] M. Kandemir, F. Li, M. Irwin, and S. Son. A novel migration-based NUCA design for chip multiprocessors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, 2008.

[13] J. Kelm, M. Johnson, S. Lumetta, and S. Patel. Waypoint: Scaling coherence to 1000-core architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 99–110, 2010.

[14] J. Merino, V. Puente, P. Prieto, and J. Gregorio. Sp-nuca: a cost effective dynamic non-uniform cache architecture. *ACM SIGARCH Computer Architecture News*, 36(2):64–71, 2008.

[15] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, 2010.

[16] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, 1995.

[17] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 336–345. IEEE, 2005.