DR-SNUCA: An Energy-Scalable Dynamically Partitioned Cache

Anshuman Gupta
University of California, San Diego

Jack Sampson
University of California, San Diego

Michael Bedford Taylor
University of California, San Diego

Abstract—**Multicore processors have become ubiquitous across many domains, such as datacenters and smartphones. As the number of processing elements increases within these processors, so does the pressure to share the critical on-chip cache resources, but this must be done energy-efficiently and without sacrificing resource guarantees. We propose a scalable dynamic cache-partitioning scheme, *DR-SNUCA*, which provides an energy-efficient way to reduce resource interference over caches shared among many processing elements. Our results show that DR-SNUCA reduces system energy consumption by 16.3% compared to associatively partitioned caches, such as DNUCA.**

I. Introduction

Multicore processors are ubiquitous today in domains such as datacenters and embedded systems and are witnessing an increasing concurrency, which will soon move us into an era of scalable manycore designs [2]. These domains need processors to satisfy some key requirements: Energy-efficiency, high utilization, and performance guarantees. As a result, manycore processors are under pressures to share critical resources, expecially the on-chip NUCA caches [2], as it allows higher utilization than fixed cache partitioning [6]. However, sharing also leads to resource contention, or *interference*, which causes difficulties in maintaining performance guarantees, and in monitoring and managing resource usage [3].

Previously, cache partitioning designs such as Virtual Private Caches [11] have been proposed, which can be used to create a dynamically partitioned Dynamic NUCA (DNUCA) [8] cache and reduce interference; however, these associatively partitioned caches can place a cache line on any associative way, which is energy-inefficient due to excessive tag-matching, as shown in Figure 1. Mechanisms such as XOR-based way-prediction [12], partial-tag match [7], and cache-block migration [5] reduce the number of ways checked per cache access. However, even with these mechanisms, cache accesses in an associatively partitioned DNUCA consume a progressively larger portion of processor energy as cache size increases. This makes DNUCA based dynamically partitioned caches energy-inefficient as aggregate cache size grows.

On the other hand, Static NUCA (SNUCA) [8] architectures use a fixed indexing function and access a constant number of associative ways for every cache request, which keeps the energy consumption low, as shown in Figure 1. By configuring these index functions, SNUCA cache sets can be partitioned among applications; however, due to fixed hashing, SNUCA cache allocations cannot be dynamically changed, which leads to reduced cache utilization.

Insight: While working sets change, necessitating dynamism in partitioning, they do not change rapidly. Thus, the frequency with which we need to repartition can be low and we should optimize performance and energy for the time *between* allocations.

This work was funded in part by NSF CAREER Award 0846152, NSF Award 1018850, NSF Award 0811794, and NSF Award 1228992.

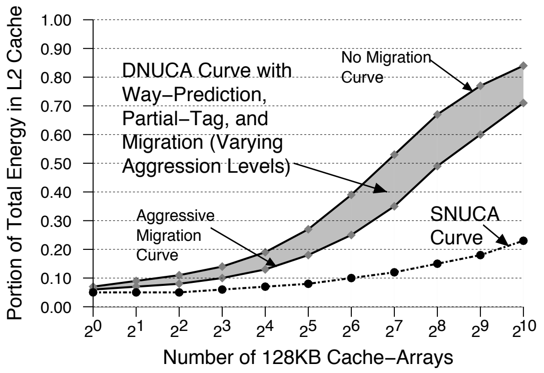

Fig. 1. **Associatively partitioned DNUCA caches are not energy scalable.** The portion of energy spent in L2 caches increases with cache size when using DNUCA, even with XOR-based way-prediction [12], partial-tag match [7] and cache migration [5] (with varying aggression levels as depicted by the shaded region). For every cache size, we chose the partial tag size that minimized the overall energy consumption. For SNUCA, the portion of energy spent in caches remains low, but SNUCA is not dynamically repartitionable.

We introduce *Dynamically Repartitionable Static NUCA*, or DR-SNUCA, a dynamically-repartitionable shared cache with static-mapping during steady-state. DR-SNUCA provides energy efficiency and high cache utilization as well as resource guarantees. DR-SNUCA uses set partitioning i.e. growing or shrinking cache allocations by changing the number of sets allocated to an application while keeping associativity constant. It uses indirect cache addressing to reduce reconfiguration overheads introduced during changes to cache allocations, to enable online reconfiguration. We also introduce *Tag-Duplication* to avoid execution stalls during reconfiguration and keep DR-SNUCA performance comparable to DNUCA.

We evaluate DR-SNUCA against a DNUCA cache on a 32-core manycore processor (similar to Tilera [2]). Our evaluation shows that DR-SNUCA reduces overall system energy by 16.3% on average, while performing within 0.5% of DNUCA. Furthermore, we show that the area and energy overheads of reconfiguration in DR-SNUCA are small.

In summary, the paper makes the following contributions -

- We propose DR-SNUCA, a novel energy-scalable design for dynamically partitioning the shared on-chip caches. It uses indirect cache addressing and set partitioning to provide energy-efficiency.
- We introduce tag-duplication in DR-SNUCA to provide uninterrupted execution during reconfiguration with small area and energy cost.
- We provide a detailed manycore evaluation that shows using DR-SNUCA cache reduces energy consumption by 16.3% while performing within 0.5% when compared to existing DNUCA caches.

II. Dynamically Repartitionable Static NUCA (DR-SNUCA)

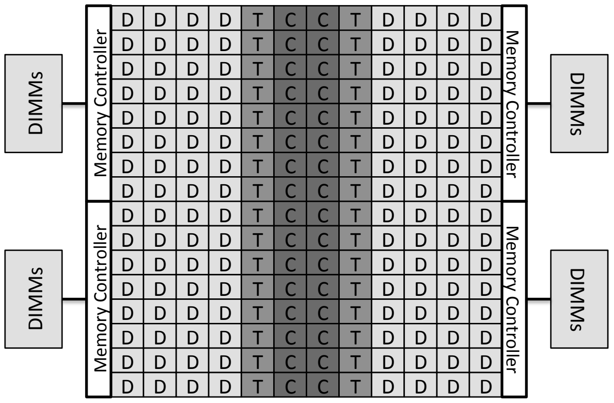

Fig. 2. **Layout of Dynamically Repartitionable Static NUCA, or DR-SNUCA.** DR-SNUCA spatially distributes the last-level cache shared by the cores (C) into cache-arrays(T,D) connected using an on-chip network, or OCN. It also physically separates the tag and data for each cache-array, consolidates them into tag-arrays (T) and data-arrays (D) respectively, and uses indirect cache addressing for cache accesses to reduce the online reconfiguration costs.

DR-SNUCA provides a shared cache for manycore processors that can be dynamically partitioned and is also energy-scalable. DR-SNUCA dynamically partitions the shared cache between applications and each application's cache portion operates as an SNUCA, except during reconfiguration periods. DR-SNUCA has physically separated cache-arrays connected through a point-to-point pipelined on-chip memory network, as shown in Figure 2, which are dynamically allocated to applications. We use the cache allocation algorithm presented in TimeCube [4]. When multiple cache-arrays are allocated to an application, they are merged by increasing the number of cache sets allocated to the application while keeping the number of associative ways constant. DR-SNUCA allocates cache-arrays to applications in powers of two.

Tag-Data Separation. In DR-SNUCA, when the cache allocation changes for an application, its number of cache sets are altered; as a result, its cache hashing changes as well. Naively moving all cache lines to their corresponding

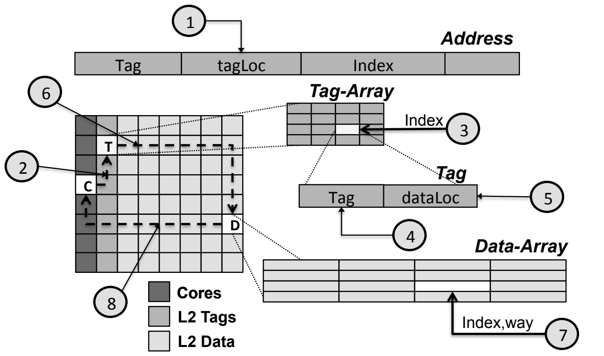

Fig. 3. **Cache access with Indirect Cache Addressing in DR-SNUCA.** DR-SNUCA uses indirect cache addressing to access the physically separated tags and data, and maintains a one-to-one correspondence between all tag and data locations by storing the data location of each tag using *dataLoc* bits. On a cache request, DR-SNUCA finds the tag-array using *tagLoc* bits (1) in the physical address, sends a request to that array (2), and then uses index bits to find the set within that tag-array (3). If some tag matches (4), dataLoc bits in the tag are used to find the data-array location (5). A request is sent to the data-array (6), the cache line is fetched (7) and is sent back to the core (8).

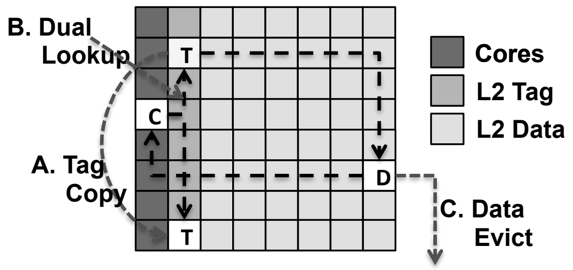

Fig. 4. **Reconfiguration**. If an application's cache allocation changes, its tags are copied (A) to their new locations in the duplicate tag-arrays for the next interval, while maintaining the tag to data location correspondence. Cache accesses are sent to both the new and old tag-arrays (B) during this reconfiguration period. If the new cache allocation is smaller, less recently used lines are evicted, and written back if dirty (C).

new locations would consume excessive energy. Instead, DR-SNUCA physically separates the tags and data for each cache-array into tag-arrays (T) and data-arrays (D), as shown in Figure 2. With this separation, cache reconfiguration moves only the tag and not the data, which significantly reduces the reconfiguration costs, since cache tags are significantly smaller than cache-lines.

Indirect Cache Addressing. DR-SNUCA uses indirect cache addressing [1] to find the cache line corresponding to any physical address on the physically separated tag and data-arrays. The location of the tag-array holding the corresponding cache-line's tag is fixed and is determined based on a hash-map that uses the *tagLoc* bits in the physical address located just above the index bits, as shown in Figure 3. Width of tagLoc equals log_2 of the number of cache-arrays allocated to the application. DR-SNUCA then uses the index bits to determine the appropriate cache set within the tag-array, and looks for a matching tag within that set. However, the corresponding data can be placed on any of the data-arrays. Thus, we store the location of data in each tag using the *dataLoc* bits. The cache set for an address in both tag-arrays and data-arrays is determined based on the index bits. The associative way for both tag and data-arrays is also the same for every cache line. Therefore, DR-SNUCA needs to store the location of only the data-array in dataLoc bits. On determining the data location, cache-line is retrieved for the requesting core. DR-SNUCA maintains a *one-to-one correspondence* between all tag and data locations in the cache at all times. This correspondence is useful in avoiding dead dataLoc references or unreachable data locations, as well as improving the indirect cache access and reconfiguration performance, and can change only during cache reconfiguration.

Reconfiguration. Cache allocations can dynamically change during application execution on interval boundaries, triggered by timer interrupts, typically every 25 million cycles. During this online reconfiguration, a *DR-SNUCA controller* is responsible for shifting tags and evicting data due to a possible change in the hashing scheme, while maintaining the one-to-one correspondence between all tag and data locations. The controller finds new tag locations for cache lines based on their tagLoc bits, and if it is not the same as their current locations, it moves the tags to their new blocks at the same index and way, as shown in Figure 4. The dataLoc bits in the existing tags at these new locations are also copied back to the old locations to maintain the tag and data location correspondence.

If the cache allocation increases, no change is required in the data blocks. However, if the cache allocation shrinks, the controller must select which cache lines to preserve and which to evict. To support this selection, in addition to the associative

Cores	32, x86-64 ISA, 3GHz, superscalar, in-order memory
L1 cache	32KB inclusive, 4 way associative, 8 word line, 1 bank, 3 cycle hit, pipelined, 1 read/write port
L2 cache	16MB: 128 cache-arrays, 1 bank per cache-array, 128KB per bank, 8 word line, 4-way associative pipelined, 1 read/write port
Network	64-wide, mesh, dynamic router, 1-cycle hop
Prefetcher	stream prefetcher, 128 streams, 32 buffers
Memory	4 controllers, bit-interleaved, 4 DIMMs/channel, 4 Ranks/DIMM, 8 Banks/Rank, 64MB/Bank, 16 Banks and 1GB DDR3 per core, 96Gb/s memory bandwidth

TABLE I. **PROCESSOR MODEL** USED FOR DR-SNUCA EVALUATION.

LRU within a set, DR-SNUCA maintains an LRU-vector per application for every equivalent location (same index, way) across all cache-arrays currently owned by the application. We call this the *Application-LRU*, and it is stored on the application's core. On allocation reductions, the controller evicts the application-LRU entries for each equivalent location, as shown in Figure 4. This provides better cache locality, and hence performance, compared to a naive selection, such as preserving all cache lines from a single cache-array. The controller has to proactively evict these lines because if it fails to writeback all the dirty lines, the cache will become incoherent. It writebacks only the dirty lines to save bandwidth. For the cache lines that are to be evicted, it still maintains the dataLoc bits in them in order to preserve the correspondence between tag and data locations.

Tag-Duplication. During cache reconfiguration, it is difficult to handle memory requests for a cache line whose tag is in transit. There are three basic approaches to handling this scenario. First, we could have a protocol to track the tag during its transit and allow intermediate structures to respond, but this is complicated and can cost additional time and energy. Second, we could stall the application execution until the reconfiguration finishes, but this will reduce application performance. Third, *Tag-Duplication*, which maintains two tag block arrays and copies the tags from the arrays allocated for the current interval into the arrays allocated for the next interval, as shown in Figure 4. While the reconfiguration is going on, all tag lookups for an application are sent to the tag blocks allocated to the application for both the current and previous intervals, which guarantees that the tag will be matched if present in the cache. DR-SNUCA uses tag-duplication to prevent application stalling and handle memory requests during cache reconfiguration. Our experiments show that the reconfiguration period is relatively small compared to the interval length, which keeps the dual-lookup costs low.

III. DR-SNUCA EVALUATION

We evaluate DR-SNUCA using a simulated 32-core processor model (Table I) similar to commercial tiled manycores, e.g. Tile64 [2], and a reconfiguration interval of 25 million cycles. 25 million cycles was found to be the interval duration yielding highest performance in our experiments. We use PTLsim [15] and a memory-system emulator to simulate execution of multiple applications on a single many-core chip while sharing last-level cache and off-chip memory accessed through memory controllers, as shown in Figure 2. The emulator internally uses DRAMsim2 [14] for modeling details of the DRAM memory system. We analytically model the area and power consumption using area and energy numbers, static as well as dynamic, obtained from RAW [10] and McPAT [9] scaled to 45nm. To reduce simulation times, we extract application representative phases using SimPoint [13] and then concurrently run SimPoint combinations.

Benchmark	Type	Benchmark	Type
IaaS-IO/webCrwl	slope	IaaS-IO/faceDet	strm
IaaS-IO/fotoBlur	slope	IaaS-IO/dskBkp	slope
FP2000/wupwise	cliff	FP2000/ammp	slope
FP2000/swim	cliff	FP2000/lucas	strm
FP2000/mgrid	cliff	FP2000/fma3d	cliff
FP2000/applu	strm	INT2000/parser	slope
INT2000/vpr	slope	INT2000/bzip2	slope
FP2000/art	strm	INT2000/twolf	slope
FP2006/equake	strm	FP2000/apsi	strm
INT2006/astar	slope	FP2006/namd	slope
INT2006/bwaves	cliff	INT2006/sjeng	slope
FP2006/h264ref	slope	FP2006/soplex	slope
INT2006/hmmer	cliff	INT2006/specrnd	strm

TABLE II. **BENCHMARK CHARACTERISTICS.** WE USE BENCHMARKS THAT PROVIDE A DIVERSE MIX OF MEMORY CHARACTERISTICS SUCH AS MISS RATES IN L1, HIT RATE IN L2, AND CACHE MISS PROFILES [6].

Benchmarks and their Classification We run combinations drawn from 26 benchmarks that span SPEC2K, SPEC2K6, and an internally-developed I/O intensive benchmark suite, as shown in Table II. This selection provides a rich spectrum of cache and memory characteristics [6].

The space of all possible benchmark combinations is very large. Moreover, it provides no intuition about the benchmarks that we have not included in our evaluation. In order to limit the evaluation space as well as incorporate a structure into our evaluation, we classify our benchmarks by memory characteristics into a three-type taxonomy [4], and then examine runs that include different ratios of the three types. The taxonomy is as follows: An application which sees no drop in miss rate with increasing cache size is a *stream* application, an application which sees a sudden drop in miss rate with cache size is a *cliff* application, and an application whose miss rate drops gradually with increasing cache size is a *slope* application. In the applications examined, cache sensitivity was a strong classifier that predicted other characteristics, such as stream applications having good prefetching behavior and high bandwidth requirements. For a workload with high variance within cache sensitivity categories, additional classification axes would be beneficial. We run representatives of these classes to refine our manycore evaluation space and can estimate behavior of similar applications.

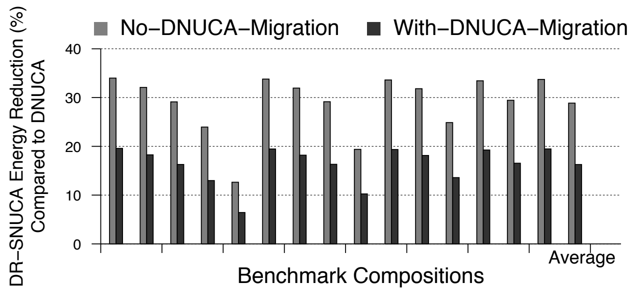

Fig. 5. **DR-SNUCA reduces overall execution energy** by 16.3% on an average when compared to DNUCA even with aggressive migration. This graph shows the energy reduction across 15 benchmark compositions represented as a tuple (streams %, slopes %, cliffs %). They are: (100,0,0); (75,0,25); (75,25,0); (50,0,50); (50,25,25); (50,50,0); (25,0,75); (25,25,50); (25,50,25); (25,75,0); (0,0,100); (0,25,75); (0,50,50); (0,75,25); (0,100,0);

DR-SNUCA is Energy-Scalable. In our experiments with 32 cores using DNUCA associative caches, we observe that a significant portion of energy is consumed in L2 (20.0% on average). In contrast, with DR-SNUCA, average L2 energy consumption is only 2.4%. Figure 5 shows that the greater energy-scalability of DR-SNUCA results in an average overall energy reduction of 16.3% compared to DNUCA.

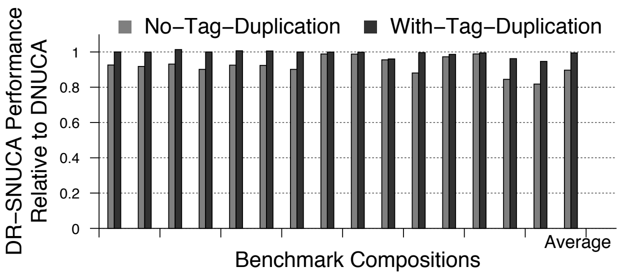

Fig. 6. **DR-SNUCA performance is comparable to the baseline DNUCA.** Without tag-duplication (TD), DR-SNUCA performs 10.3% worse in comparison to the baseline due to reconfiguration stalls. However, after adding tag-duplication, DR-SNUCA is able to perform within 0.5% of the baseline. This graph shows the same compositions as used in Figure 5, and the average.

The execution times for DNUCA and DR-SNUCA are comparable, as shown in Figure 6, due to similar cache hit rates. When running a mix of 32 applications on our prototype architecture, our results show that without tag-duplication we can lose 10.3% of performance. However, with tag-duplication, reconfiguration for DR-SNUCA can be done in parallel with execution with no timing overhead, and we are able to perform within 0.5% of the baseline DNUCA, which in turn performs 14.7% better than SNUCA [8], at the expense of just 6.42% of the overall chip area, as seen in Figure 8. This small penalty is due to indirect cache addressing in DR-SNUCA.

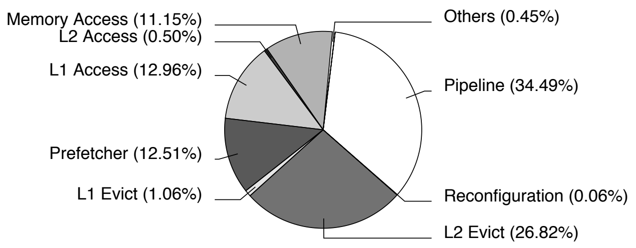

Fig. 7. **Energy distribution with DR-SNUCA.** Energy consumed by cache reconfiguration (0.06%), including tag migration and data eviction, is small.

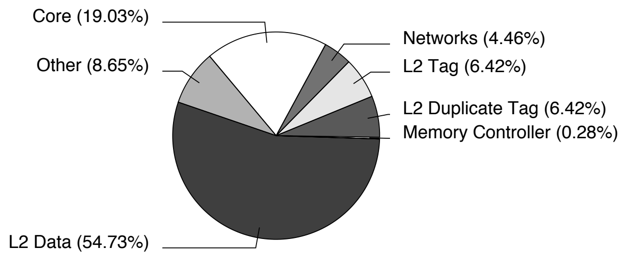

Fig. 8. **Area distribution with DR-SNUCA.** Area required for reconfiguration mechanisms in DR-SNUCA, such as 6.42% for tag-duplication, is low.

Area and Energy Overheads of DR-SNUCA are small. For an example 32 application mix, we observe that with DR-SNUCA we are able to significantly reduce the energy consumption of L2 accesses; as a result, the portion of total energy consumed in L2 access when using DR-SNUCA is low (0.50%), as shown in Figure 7. Most of the energy is consumed in core execution (47.45% including L1 access) and main memory operations (45.36% for access and writeback). Energy consumed for reconfiguration in DR-SNUCA is low: 0.06% of the total energy. The tag-duplication required to support continuous execution during reconfiguration in DR-SNUCA consume only 6.42% of the area, as shown in Figure 8. Overall, the mechanisms used to support reconfiguration in DR-SNUCA are energy and area efficient, making it as efficient as SNUCA while also providing the ability to dynamically repartition.

IV. Related Work

Govindan et al. [3] and others have demonstrated the ill-effects of interference, and dynamic cache partitioning [11] has emerged as the most transparent solution to tackle this problem. Kim et al. [5] proposed NUCA architectures that spatially distribute shared cache to reduce access energy and time. However, the existing NUCA techniques, such as SNUCA and DNUCA, do not satisfy our three key requirements; SNUCA is not dynamically repartitionable and DNUCA is not energy-efficient for large cache sizes required for manycore architectures, even when we use optimization techniques, such as way-prediction [12], partial-tag matching [7] and data migration [5]. Thus, we extend SNUCA with cache-indirection [1] to create DR-SNUCA, which is both dynamically reconfigurable as well as energy-efficient for large cache sizes.

V. Conclusion

DR-SNUCA is an energy-scalable dynamically partitioned cache, which reduces the energy consumption for a 32-core system by 16.3% while performing within 0.5% when compared to DNUCA caches. The area and energy overhead of the reconfiguration mechanisms are low, and thus DR-SNUCA can be used to provide interference-free on-chip caches for manycore processors.

References

[1] A. Agarwal and S. D. Pudar, "Column-associative caches: a technique for reducing the miss rate of direct-mapped caches," in *ISCA*, 1993.

[2] Bell et al., "TILE64 Processor: A 64-Core SoC with Mesh Interconnect," in *ISSCC*, 2008.

[3] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines," in *SOCC*, 2011.

[4] A. Gupta, J. Sampson, and M. B. Taylor, "Timecube: A manycore embedded processor with interference-agnostic progress tracking," in *SAMOS*, 2013.

[5] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A NUCA substrate for flexible CMP cache sharing," in *ICS*, 2005.

[6] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation," in *VSSAD TR*, 2007.

[7] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill, "Inexpensive implementations of set-associativity," in *ISCA*, 1989.

[8] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *ASPLOS*, 2002.

[9] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.

[10] M. B. Taylor et al., "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," in *ISCA*, 2004.

[11] K. J. Nesbit, J. Laudon, and J. E. Smith, "Virtual private caches," in *ISCA*, 2007.

[12] Powell et al., "Reducing set-associative cache energy via way-prediction and selective direct-mapping," in *MICRO*, 2001.

[13] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS*, 2002.

[14] Wang et al., "Dramsim: A memory-system simulator," in *SIGARCH Computer Architecture News, September 2005*.

[15] M. T. Yourst, "Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator." in *ISPASS*, 2007.