Multi-Execution: Multicore Caching for Data-Similar Executions

Susmit Biswas, Diana Franklin, Alan Savage, Ryan Dixon, Timothy Sherwood,
Frederic T. Chong
Department of Computer Science, University of California, Santa Barbara
{susmit, franklin, asavage, rsd, sherwood, chong}@cs.ucsb.edu

ABSTRACT

While microprocessor designers turn to multicore architectures to sustain performance expectations, the dramatic increase in parallelism of such architectures will put substantial demands on off-chip bandwidth and make the memory wall more significant than ever. This paper demonstrates that one profitable application of multicore processors is the execution of many similar instantiations of the same program. We identify that this model of execution is used in several practical scenarios and term it as "multi-execution." Often, each such instance utilizes very similar data. In conventional cache hierarchies, each instance would cache its own data independently. We propose the *Mergeable* cache architecture that detects data similarities and merges cache blocks, resulting in substantial savings in cache storage requirements. This leads to reductions in off-chip memory accesses and overall power usage, and increases in application performance. We present cycle-accurate simulation results of 8 benchmarks (6 from SPEC2000) to demonstrate that our technique provides a scalable solution and leads to significant speedups due to reductions in main memory accesses. For 8 cores running 8 similar executions of the same application and sharing an exclusive 4-MB, 8-way L2 cache, the Mergeable cache shows a speedup in execution by 2.5× on average (ranging from 0.93× to 6.92×), while posing an overhead of only 4.28% on cache area and 5.21% on power when it is used.

Categories and Subject Descriptors

B.3.2 [**MEMORY STRUCTURES**]: Design Styles—*Cache memories*; C.4 [**Computer Systems Organization**]: PERFORMANCE OF SYSTEMS—*Design studies*

General Terms

Design, Management, Measurement, Performance

Keywords

Data Similar Execution, Cache Design, CMP

1. INTRODUCTION

As we move from tens to hundreds of cores on a chip, it is easy to get lost trying to think of all the new ways this raw performance potential can be unleashed on those traditional applications we are all so comfortable with. Programming these chips to be efficient on these traditional workloads is important, but it is also quite tricky. As a single memory stream scales to tens, hundreds, or even thousands of reference streams, the memory system will struggle to service all the requests in a timely manner. Furthermore, even in the embedded markets where these massively parallel cores are already making inroads (e.g. Tilera TILE64[1], Ambric Am2045[2], and Nvidia GeForce GT200[3]), the effort to modify these applications to be both correct and efficient at those levels of parallelism is non-trivial.

One natural, but easily overlooked, way to make use of this raw computational power of hundreds of cores is through the execution of multiple copies of the same program with different input data or parameters. When solving real problems (rather than running benchmarks), this model of parallelism is already common practice. For example, in the machine-learning domain many poorly performing but seemingly independent "learners" can be trained in parallel, and the results can be combined together through a technique called *boosting.* In fact, while these individual learners are independent (in that there are no dependencies between processes), they share something very important that is not typically exploited by the architecture: *the contents of much of their data.* In this paper, we explore this interesting under-explored class of applications, which we term "multi-execution codes."

One of the most interesting behaviors exhibited by this class of application is that there often exists a high degree of similarity across the *data* of different instances of the application. We argue that this fact can be exploited through novel architectural mechanisms to reduce the main memory accesses, thereby speeding up execution (sometimes by a factor of 4 or more). We believe that multi-execution could become a useful model of execution as multicores scale because no software changes are necessary to take advantage of a multicore system and, frankly, programmers (and even non-programmers) already use it in many domains.

We select several applications from simulation, optimization, database, and learning domains, and find that the similarity of multi-execution working sets can be quite high, but careful design is needed to exploit this similarity profitably in a memory system. We present a low-overhead hardware mechanism that can improve caching by merging identical data from different physical-memory regions belonging to different processes, and thereby, reduce required off-chip bandwidth by an order of magnitude and speedup the execution of programs. Implementing a cache architecture capable of merging data dynamically poses interesting challenges to support correctness, physical-memory sharing, process migration etc, which our proposed architecture solves successfully. Note that our mechanisms can be completely bypassed when multi-execution is not utilized, resulting in no performance degradation and a negligible power increase. The main contributions of this paper are summarized in the following items:

1. We introduce the notion of "multi-execution" domain and show that high data similarity exists across multi-execution instances. We also discuss several scenarios where multi-execution is useful.

2. We propose a *Mergeable* cache architecture which increases cache capacity by merging cache lines with identical content used by different processes, and conservatively improves performance by 2.5× on average while incurring a modest increase in area and power when in use.

In order to demonstrate the strength of our approach, we have implemented a cycle-accurate simulation framework based on PolyScalar[4] simulator. In this paper, we present results with 8 applications including 6 benchmarks from the SPEC2000 benchmark suite[5], *libsvm*, and *icsiboost*.

The remainder of the paper is organized as follows. We motivate our approach in Section 2, describe the benchmarks in Section 3, and explain the challenges and techniques of implementing a Mergeable cache architecture in Section 4. Section 5 illustrates the experimental methodology and 6 shows the results. We discuss previous approaches to reducing off-chip memory accesses in 7. Section 8 presents future work, and finally the conclusions are drawn in Section 9.

2. MOTIVATION

The simplest and cheapest way to take advantage of multiple processing cores is by running a separate sequential program on each core in a multi-programmed approach. However, the demand for memory bandwidth increases super-linearly with the number of cores in a chip when running unrelated programs on each core. When multiple cores share a cache, effective cache size per core is reduced, which leads to a super-linear increase in cache misses. Figure 1(a) depicts the average number of L2 misses per 1000 (L1) data memory references for four applications. The number of off-chip accesses decreases non-linearly as the size of the L2 cache increases. Thus, we can expect that even small increases in the effective cache capacity can lead to substantial increases in performance and decreases in total system power (due to DRAM access).

There exists a domain of applications ideally suited to take advantage of a large number of cores and achieve cache compression by executing several instances of the same sequential program with small differences in parameters or input data. We refer to this as the "multi-execution" domain. Within this domain, the various instances of the program have very similar data, but exactly where and when the similarity occurs is not predictable. That is, any given piece of data may go through phases of being identical and not identical across multiple executions of the program. An alternative to multi-execution is to write an explicitly parallel program that takes many instances of an application and explicitly shares redundant data. However, this approach is labor intensive, difficult to get correct, often requires source access to libraries and copyrighted/proprietary codes, and can miss substantial data similarity that can only be discovered with an efficient dynamic mechanism, as we show in our study.

In this paper, several applications from a multi-execution environment are inspected for opportunities to merge identical data. We find that a large amount of identical data exists in cache lines owned by different processes. Four such applications are shown in Figure 1(b). The graph depicts the similarity of the cache contents between two independent executions of these applications on the same input data and with up to 50% variations in parameter values when each process owns a private 1-MB direct mapped cache. Cache snapshots are taken every 10M accesses and results are shown for the first 2 billion accesses only. We observe very high similarity across executions of *300.twolf* and *175.vpr*, whereas *188.ammp* shows high similarity in first 1.2 billion accesses. *255.vortex*, on the other hand, shows very low similarity across executions, and hence, is unable to benefit significantly from cache line merging technique. The high similarity across executions indicates that the effective capacity of on-chip cache can be increased by merging cache lines.

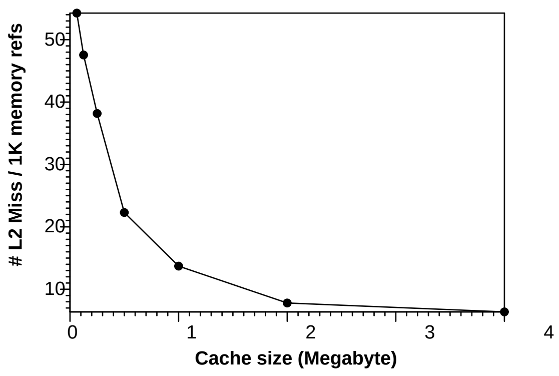
3. APPLICATIONS

Benchmarks are selected from several domains such as simulation, visualization, machine-learning etc. where multi-execution is used in practice. In each of these domains, practical scenarios are used to construct input and parameter variations up to 50%.

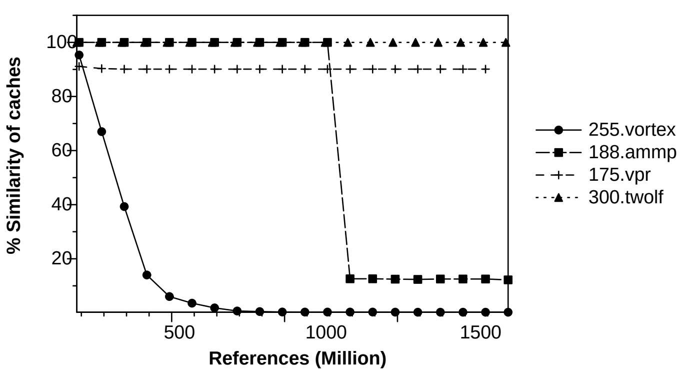
188.ammp is a computational chemistry application in molecular dynamics from SPEC2000fp[5],which computes the energy of the final configuration of a set of atoms in water and protein. We varied the following parameters, which are often varied to synthesize, purify or characterize phenomena more effectively in domains of molecular biology and organic chemistry[16]: *mxdq* works as a threshold to update the full non-bonded list when atomic displacement is greater than the value in angstroms. *bbox* is the bounding box dimension used for computing potential energy. *temp* specifies the simulation temperature. Simulation can be performed with different values to measure sensitivity to temperature. *mmbox* controls the fast multipole algorithm (FMM) for long-range non-bond energy calculation, and when set to a non-zero functions as a factor to compromise between accuracy and speed. *numstep* specifies the number of steps used in the line minimizer.

300.twolf, the TimberWolfSC placement and global routing package is used in the production process of microchips. TimberWolfSC program uses simulated annealing as a heuristic to find solutions for row-based standard cell design. The global router adds extra cells known as *feedthrus* to complete the route if enough space is not available between two adjacent standard cells. The result of placement and routing are dependent on two parameters, *row separation* and *feedthru width*, and several executions are performed in order to discover the "magic numbers" for a design.

175.vpr(*Versatile Place and Route*) is a placement and routing program that automatically synthesizes a technology-



(a) Cache performance vs. size



(b) Data cache similarity

Figure 1: Two factors contribute to the motivation of our approach. (a) **shows the number of average L2 miss of four benchmarks for every 1000 data memory references. Cache performance has a super-linear relationship with cache size. When the L2 cache is shared by multiple cores, effective memory per core is reduced, increasing the miss rate. Merging duplicate cache lines to reduce cache requirements has the potential to improve cache performance substantially, even with small increases in effective cache capacity.** (b) **Similarity of cache contents between two executions of an application run with different input sets or parameters. Cache snapshots are taken at an interval of 10M instructions for all runs while simulating 1 MB direct-mapped cache. Similarity is computed using line-by-line comparison of the caches and taking the ratio of identical lines to total lines.** ***175.vpr*****,** ***300.twolf*** **and** ***188.ammp*** **show good similarity characteristics whereas** ***255.vortex*** **lacks similarity in cache footprint. Therefore, merging identical cache lines is expected to improve the miss rate of L2 cache for all of these applications except** ***255.vortex*****.**

mapped circuit in a Field-Programmable Gate Array using combinatorial optimization. Finding an optimal routing depends on choosing the optimal *routing-channel-width*. In practice, many simulations are required with different values of *routing-channel-width* to find an optimal route[10], so we vary this parameter in our experiments.

183.equake simulates the propagation of elastic waves in large and heterogeneous valleys to find the time history of the ground motion in the valley corresponding to a seismic event. Computations are performed on an unstructured mesh which resolves wavelengths locally using a finite element method. The epicenter and intensity of earthquake are varied. This would be used to explore how a structure or terrain would respond to different potential earthquakes.

181.mcf is designed to find a solution to the single-depot vehicle scheduling problems for a single depot and a homogeneous vehicle fleet occurring in the planning process of public transportation companies. We simulate a scenario where one of the trips is to be deleted due to budget adjustment. To find an optimal solution, several simulations need to be performed with different input sets. We delete one trip randomly and perform simulation on different sets of inputs in parallel.

255.vortex is a single-user object-oriented database transaction benchmark. We simulate random insert, delete and look up queries which are common database operations. However, with randomized queries on the same database the similarity is low. We include vortex in our benchmarks to show how our design performs when applications have little similarity.

Support Vector Machine (libsvm)[12] is an application from machine-learning that requires the user to find the best values for parameters C and γ using cross-validation. C is used as the cost for optimization. It trains on the whole training data set using these parameter values and provided test data. A good model can be generated with careful selection of C and γ values. Therefore, in practice, libsvm is executed multiple times with different parameter values to tune models for better accuracy[12]. We use a standard texture dataset *Satimage* from UCI Repository of Machine Learning Databases[8] as input, and vary these two parameters.

icsiboost is an implementation of Boosting[25], another machine-learning technique. Several weak learners are employed in the process of learning, and rules discovered by them are combined to create a more accurate set of rules by voting. Traditional boosting algorithms are serial in nature, but Lozano et *al.*[19] propose a parallel execution on the same training set with randomized initial weights to achieve significant speed up in the learning process. We have customized icsiboost[6], an open-source boosting package, to implement this technique, and assign random initial weights to learning steps. The *UCI Census Income* dataset[8] is used as input to this benchmark.

We have SPEC2000 train inputs for 300.twolf, 175.vpr, 255.vortex, and the Minnespec[17] inputs for simulating *188. ammp, 181.mcf, 183.equake* as *ref* inputs are too large for cycle-accurate simulation of multiple cores. The standard UCI dataset is used as input for *libsvm* and *icsiboost.* Simulations are run until completion to ensure that results are not skewed due to the initialization phase. Characteristics of these applications are summarized in Table 3.

4. THE MERGEABLE CACHE ARCHITECTURE: CHALLENGES AND SOLUTIONS

Although the data similarity across instances of our multi-execution applications is high, we need to design a cache architecture that can take advantage of this similarity without incurring excessive overhead. Specifically, our goal is to design an efficient hardware technique that dynamically finds identical data among different processes, merges such

Benchmark	Description	Input Modification	Run Length	Footprint (MB) vsz	rsz
175.vpr	FPGA Place and Route	routing-channel-width	3.3 B	2.67	1.35
181.mcf	Combinatorial Optimization	reduced trips	6.99 B	18.94	16.98
183.equake	Seismic Wave Propagation	epicenter and intensity	4.36 B	1.69	0.47
188.ammp	Chemistry	simulation parameters	6.13 B	96.72	39.91
255.vortex	Database	random insert, lookup	5.85 B	15.71	14.38
300.twolf	Place and Route	intercell gaps	4.13 B	11.60	10.35
libsvm	Machine learning	C and γ parameter	5.67 B	6.05	3.38
icsiboost	Ensemble learning	distribution of sample	2.30 B	29.45	18.03

Table 1: Benchmarks and input descriptions. The observed memory footprint sizes reported are average of 20 runs where resident memory size and virtual memory size are abbreviated as *rsz* and *vsz* respectively.

data, and splits them again when the processes write to the data. We focus on data caches and assume that the OS can map identical program code in our separate instances to the same physical page, effectively merging instruction caching by default. There are two significant technical issues which need to be addressed in merging identical data blocks from multiple processes.

1. Finding identical data to merge is an expensive operation if every access to the memory results in comparing data with all valid cache lines. Searches must be minimized while the opportunity for identifying mergeable data is maximized.

2. Merged data needs to be organized in such a way that data is quickly found in a standard cache access.

Efficiently searching for identical data: To solve the first problem, we observe that applications are most likely to have identical data located at the same virtual address (but different physical addresses). Thus, we limit our searches to only data having the same virtual address. To perform this search efficiently, we must arrange to map all the relevant virtual addresses to the same cache set.

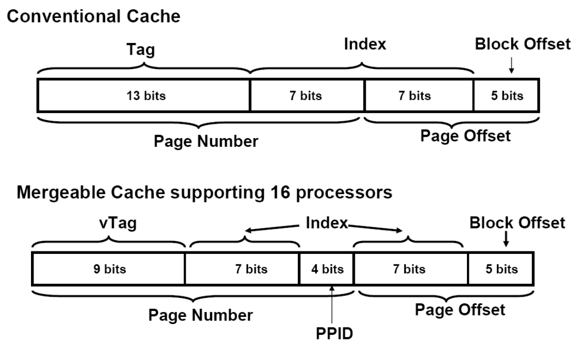


Figure 2: Modification in cache addressing

To achieve this, we use a *page coloring technique*[9][23] to place pages in DRAM. Although we restrict the OS in mapping pages, the attributes of multi-execution workloads are likely to compensate for the negative effects of such restrictions. As all the relevant processes running the same program (with different inputs) have similar working sets in the virtual address space, it is quite space-efficient to assign the same virtual page of all processes in a multi-execution workload to consecutive physical pages for any pages that are candidates for merging. In this way, all bits in the physical address except the lowest ($log_2 P$) bits of the physical page number will be identical for the same virtual address across all processes where P is the number of processors. We call this small set of bits that distinguishes the physical addresses the Process Physical ID (PPID). Figure 2 shows the location of the PPID with respect to the page number and page offset.

This page coloring scheme is chosen because of its flexibility. If the entire memory were partitioned among the multi-execution processes, physical memory space would be wasted for shared pages containing instructions, and it would be more difficult to dynamically share space with the operating system and other processes. The blocks of pages reserved for multi-execution processes are allocated dynamically in sets of P pages and the PPID length is set to $log_2(P)$, where P is the number of *processors*. The amount of merging is also bound to the number of processors. If there are more processes than processors in the system, merging can still occur, but only between subsets of the processes. Those subsets are treated as separate multi-execution workloads. If there are fewer processes than processors, the operating system can place non-merging pages (OS or instruction pages) in the rest of the block of allocated pages or, if there are less than half the number of processes, two sets of virtual pages can fit in a single block. Finally, there is no loss of benefit from spatial locality, because spatial locality is still maintained within a page. Our scheme only stripes the process allocation at a page granularity.

There are two requirements to support merging physical memory owned by different processes - quickly finding data to merge and quickly finding merged data. For the first, addresses from the same virtual page of different processes need to be mapped to the same set in the cache. We accomplish this removing the PPID from the index to the cache sets. Figure 2 shows the new index, which is taken from the bits before and after the PPID. Thus, the same virtual address in different processes (that are part of a potential merging group) will share the same index. For the second, it must be easy to recognize the addresses within the set that are the same virtual address. The new tag is made up of two parts - a subset of the original tag, which we call the *vTag*, and the PPID. For any address that is the same virtual address, the vTag will be identical, but the PPID will differ.

Inside the cache, the vTag and PPID are stored separately (Figure 3). The vTag is stored like a normal tag, but the

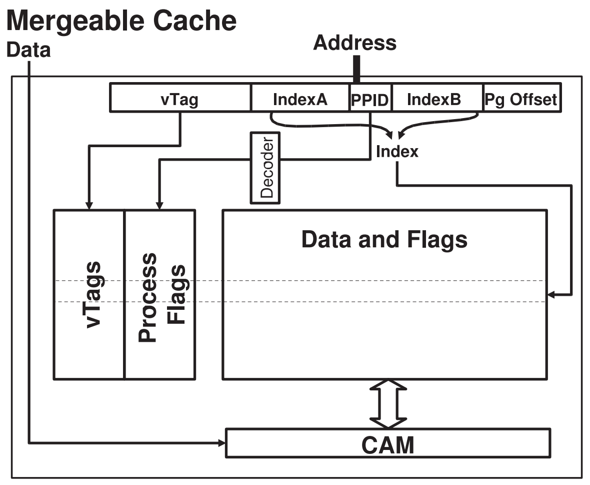


Figure 3: *Mergeable* cache architecture. When a line is evicted from L1D cache and moved to L2, all lines with identical vTags from the mapped cache set are copied in CAM and compared for identical content.

PPID is stored as a bit vector (*process flags*) of length equal to the number of processors. Each PPID corresponds to a separate bit in the flag vector (multiple bits can be set to denote sharers of a merged line). The vTag of the line that is entering the L2 cache is compared with the vTags in its cache set. Data from cache lines with matching vTags are copied to the associative buffer or content-addressable-memory (CAM) where search for a cache line with identical content is performed. If a cache line with identical data and vTags is found, the new line is not given a new slot in the cache. Instead, the bit in the process flags corresponding to the new address' PPID is set. In other words, identical cache blocks share a single cache block while different cache blocks are allocated to lines with non-identical data.

Accessing merged data: When the L2 cache receives a request, the vTag is used to match the tag field, and the PPID is used to check the proper bit in the process flags field. The vTag must be a perfect match, whereas the PPID is expanded into a bit vector with a single high bit, and a bitwise-and operation with a non-zero result indicates a match. An access is considered a *hit* only if both the vTag and process flag match. Because the process flag matching can be performed in parallel to vTag matching, it adds negligible delay along the critical path.

Architectural details: To maximize the on-chip storage capacity of our system and simplify the merging and splitting of lines in the L2 cache, the L1 and L2 caches maintain an exclusive policy. Each line may reside in either the L1 cache or the L2 cache, but never both. This means that the L2 cache only contains items that have been evicted from the L1 cache. This greatly simplifies merging and splitting. A line enters the L2 cache as a result of an eviction from the L1 cache and attempts to merge with a line already residing in the L2 cache. That line is no longer residing in any L1 cache. In order to modify the line, it must be removed from the L2 cache, placed into an L1 cache, and modified. Therefore, no modifications to existing lines are ever made in the L2 cache. This means that lines in the L2 cache are not split up when modified. A merged line to be modified is merely removed, which requires no more than setting the proper process flag to 0. Though our implementation performs merging in the L2 cache, this technique is applicable in all lower level (L2, L3, etc.) shared caches.

The Mergeable cache first splits the address into two parts - the vTag and the PPID which is then expanded to process flags. If the tag in a conventional cache requires $tagsize$ bits per line, the new hardware needs $(tagsize - log_2(P) + P)$ bits per line, where P is the number of processors. Along with this increase of $(P - log_2(P))$ bits per line, the Mergeable cache also needs a CAM for comparing the data of a line entering the cache to the data contained in the its set in the cache. The length of each line in the CAM is equal to the cache's line size, and it contains the same number of lines as the cache associativity. The area and power numbers for the CAM are listed in Table 4.

Rows	Area(mm^2)	Latency(ns)	Power(nW)
2	0.0132	0.507	0.0048
4	0.0140	0.513	0.0055
8	0.0156	0.525	0.0071
16	0.0190	0.549	0.0102

Table 3: Overhead of 256-bit wide CAM obtained using Cacti 4.2 for $45nm$ technology node.

The area of a 8-way CAM is $0.0156mm^2$ whereas the area of a 4-MB, 8-way set associative cache in $45nm$ technology node is found to be $16.84mm^2$ using *Cacti*[27]. Overall, a Mergeable cache poses an overhead of 4.28% in area due to the *process flags* and CAM. Power consumption is increased by 5.21% and cycle time by 0.92%. The 5.21% estimate is conservative, because more power is saved by reading only cache lines whose vTags match into the CAM. For applications with low data similarity, as well as conventional workloads, the data-mergeability could be turned off by using conventional mapping function and disabling the CAM to reduce overhead on power and delay. For simplicity, we assume static decisions in this work.

Finally, an additional optimization from our mergeable cache is that we can augment our memory controller to be aware of merged data. By transmitting the process flags, we can transmit only one copy of the data but have it be written to multiple places in memory. This requires one extra cycle for each write-back of a merged line to transmit the process flags.

5. METHODOLOGY

In this section, we describe our experimental methodology and simulation framework which we have implemented to evaluate the effectiveness of our scheme. The simulation infrastructure is built on the PolyScalar[4] multiprocessor simulator. PolyScalar is a multi-processor version of the Simplescalar[15] simulation tool and uses PISA as the instruction set architecture. The configuration of the simulated system is described in Table 5.

All caches maintain an exclusive policy in order to maximize the number of unique lines that can be stored on-chip. That is, no line can be contained in more than one cache at any time. We experiment with the following two cache architectures in this study.

1. *Shared Cache:* processors share a large conventional L2 cache with an LRU replacement policy.
2. *Mergeable Cache:* processors share a large, Mergeable L2 cache with similar configuration as shared cache,

Cache Type	**Size** (MB)	**Area** (mm^2)	**Access Time** (ns)	**Cycle Time** (ns)	**Read Power** (W)
Conventional	4	15.02	4.98	0.357	0.154
Mergeable	4	16.84	5.22	0.357	0.162

Table 2: Area, delay and power overhead of Mergeable cache obtained using Cacti 4.2 for $45nm$ technology node.

in which cache lines are merged if and only if their contents match.

We simulate both instruction and data memory accesses in the L2 cache but keep only a single copy of instruction cache lines because the text section is shared by all the processes running the same application. So, instruction memory access has the same effect on a conventional L2 and mergeable L2 cache.

6. RESULTS

In this study, we look at several aspects of the Mergeable cache. First, we explore how it changes the off-chip traffic, due to both L2 cache misses and L2 cache writebacks. We then present cycle-accurate performance results. We continue with an analysis of the merging that occurs in the L2 cache. Finally, we discuss a major tradeoff that occurs between the addressing scheme used for merging and conflict misses.

Off-Chip Traffic: The Mergeable cache reduces off-chip traffic in two ways. First, it merges identical data, leading to higher on-chip capacity and to fewer DRAM requests for data. Second, when merged data is evicted from the cache, it can transmit the data only once. An extra cycle is used transferring the process flags bit vector, and the DRAM controller writes the one line of data to multiple locations.

In Figure 4, we show the L2 misses for all of the benchmarks using the dark bar. One miss in the L2 cache affects performance substantially as DRAM access latency is on the order of hundreds of cycles. As the number of cores in processors scales, the miss rate of the L2 cache increases. By merging duplicate cache lines, the Mergeable cache increases cache capacity per core, thereby reducing requests to DRAM.

We can see that the mergeable cache reduces the number of L2 cache misses by up to 9x in *icsiboost*. We also see that, in general, the savings due to merging increases as the number of processes increases. There are two notable anomalies, *vortex* and *vpr*. They illustrate the tradeoff with the Mergeable cache - as the number of processes increases, there is an increase in conflict misses in sets with little merging. This will be discussed in more detail at the end of this section.

Not only does the Mergeable cache decrease the number of DRAM accesses due to L2 misses, when a merged line is evicted from the L2 cache, that data is transmitted to DRAM in a single transaction. In our cycle-accurate results, we incur an extra cycle to transmit the process flags bit vector. Note that without this optimization, although the number of L2 misses might decrease, the Mergeable cache configuration could potentially require a very similar number of writebacks, as each merged eviction is expanded into several writebacks.

The light bar in Figure 4 illustrates the savings due to a decrease in writebacks. Note that as the number of processes increases, for most applications, the percentage of merged writebacks decreases, leading to less savings due to merged writebacks. This may seem counterintuitive, because it would seem that more merging should imply more merged writebacks. As the degree of merging increases, the merged lines get accessed more often, decreasing their chances of being evicted. So it is possible for increases in merging not to directly translate to increases in the eviction of merged lines. In fact, the decrease in writebacks has little relationship to the decrease in L2 misses. Another counterintuitive result is *vpr*. It has fewer L2 misses with the Mergeable cache than with the conventional cache yet it incurs more writebacks. This is due to an increase in L1 cache accesses and misses, both due to differences in the branch mispredictions.

Performance: We begin by showing the effect that increasing the number of processors has on our applications. We measure the effects of increasing the number of processes $(2, 4, 8$; 1 process per processor core) for a fixed amount of cache (4-MB) and associativity (8-way). Figure 5 shows that the Mergeable cache performs better than a conventional cache on average, though the individual results vary depending on the similarity in the application. For applications with high similarity, the Mergeable cache produces significant speedup, as exhibited by *libsvm*, *icsiboost* and *twolf*. In addition, as the number of processes increases, the benefit of the Mergeable cache increases across almost all applications. The increase of DRAM traffic observed earlier in *vpr* is reflected in the speedup graphs. The speedup increases from 2 to 4 processors due to merging, but decreases again at 8 processors due to an increase in L2 cache misses. Finally, in our application with the lowest amount of sharing, the line-merging technique results in a slowdown for *vortex*. Overall, the Mergeable cache outperforms shared conventional cache configurations by a factor of 2.5×, on average.

Merging Analysis: One interesting advantage of the Mergeable cache is that it can dynamically discover mergeable cache blocks which software mechanisms cannot without incurring large overhead for merging data blocks. In particular, the Mergeable cache can merge both *dirty* and *clean* blocks. A technique such as copy-on-write only exploits merging of clean data. Figure 6(a) shows the distribution of the number of times a line is written for every line which gets merged in the L2 cache. Most of the merged data is dirty, meaning a pure copy-on-write-based sharing is not capable of exploiting this phenomenon. Figure 6(b) breaks down the distribution of merged lines which are dirty to demonstrate that in many of the applications, merging is also exploiting more than just data that is initialized only once. Intuitively, similarity in input parameters results in large similarity in both control flow and data values. To get more insight we hand-analyzed four benchmarks to find the regions of code and data structures that lead to merging in L2 cache. We analyzed a few of the top-hitting entries and found that more than 20% of the accessed merged data is dynamic, as shown in the code snippets. Sharing

Processors	2 - 8	**Branch Penalty**	3 Cycles
Issue/Commit Width	8/8	**DRAM Latency**	200 Cycles
I-Fetch Q	8	**Mem Ports**	2
LSQ Size	64	**System Bus Transfer Rate**	8GB/s
RUU Size	128	**L2 Cache**	4MB, 8 way, 32 byte lines
ALU/FPU/Mult/Div	4/4/1/1	**L2 Latency**	6 Cycles
Branch Predictor	2-level, 1024 Entry History Length 10	**L1 I-Cache L1 D-Cache**	32KB + 32 KB, Direct Mapped 32 byte lines
BTB size	2048	**L1 I,D-Cache Ports**	4
RAS entries	8	**L1 Latency**	1 Cycle

Table 4: Configuration of the simulated processors. The L2 cache size is held constant as the number of processors increases.

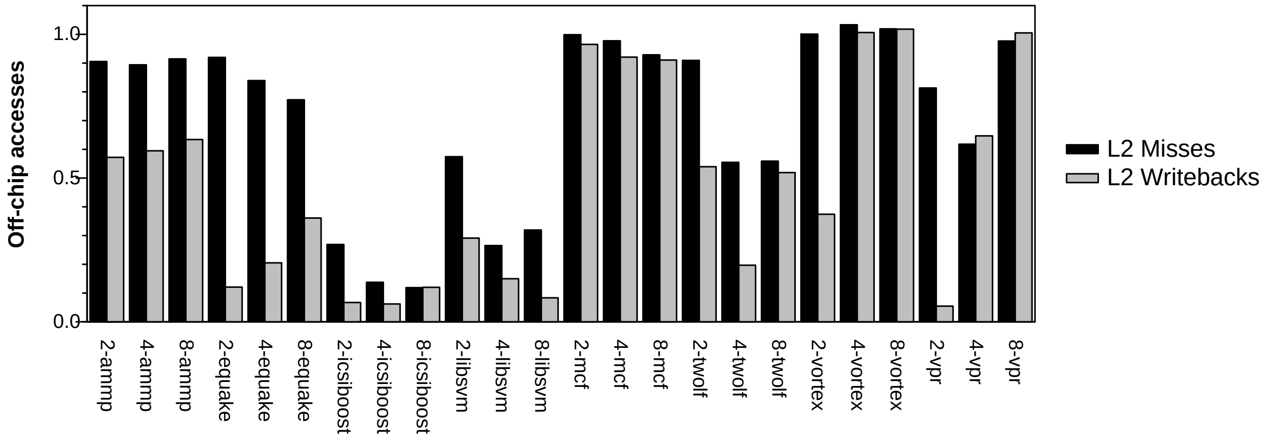


Figure 4: The graph shows the number of DRAM requests (normalized to conventional cache case) for Mergeable caches as the number of processors increases. The *x-axis* shows the number of cores and the benchmark. The amount of off-chip traffic increases with the number of cores per processor, leading to a slowdown in application execution. The dark bar shows that the L2 miss rate is typically reduced by merging identical data. The light bar shows the decrease in writebacks due to the Mergeable cache writing merged lines as a single transaction.

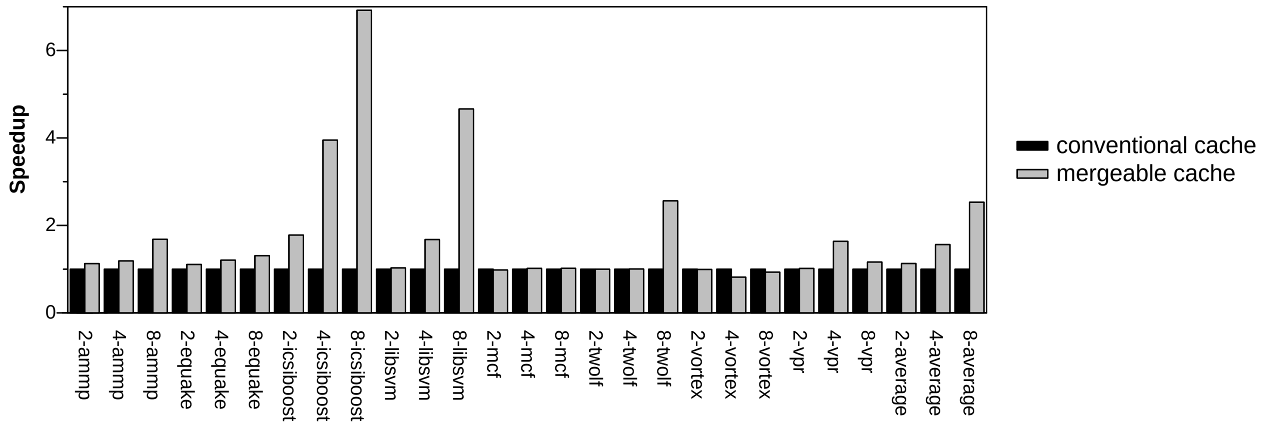


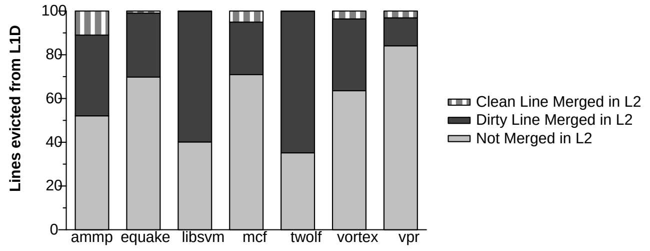
Figure 5: Speedup for all benchmarks simulated with 4-MB, 8 way L2-cache having 32 Byte block while running 8 instances of each application. Mergeable cache show speedup in all benchmarks except in 255.vortex where Mergeable cache suffers from alignment of dissimilar data in the same cache set resulting in increased conflict misses. We analyze this case later in detail.

this data would require substantial synchronization using a shared memory programming model.

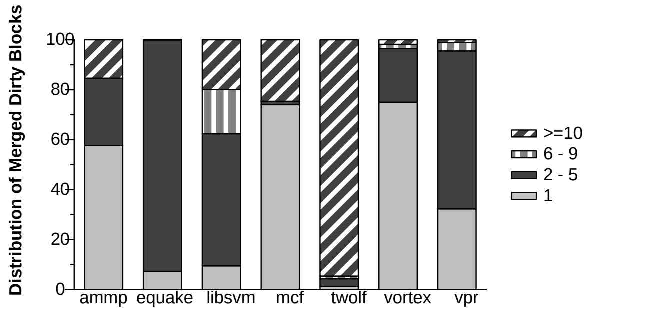
Discussion: The Mergeable cache shows significant potential in identifying and merging identical data in order to reduce off-chip traffic. By reducing the L2 miss rate and merging writebacks to DRAM, the Mergeable cache experiences an average of 2.5x speedup with less than 6% power and area overhead. Despite these benefits, the Mergeable

Bench	Min Dyn. Similarity	Code Snippets
libsvm	60%	svm.cpp:294 $sum+ = px{\rightarrow}value * py{\rightarrow}value;$
mcf	25%	implicit.c:261 $if(tail{\rightarrow}time + arcin{\rightarrow}org\_cost > latest)$ mcfutil:85 $node{\rightarrow}potential = node{\rightarrow}basic\_arc{\rightarrow}cost + node{\rightarrow}pred{\rightarrow}potential;$
vpr	23%	route.c:1082 $if(heap[ito + 1]{\rightarrow}cost < heap[ito]{\rightarrow}cost)$
twolf	28%	dimbox.c:132 $if(netptr{\rightarrow}flag == 1)$ dimbox.c:184,214,257 $ttermptr = termptr{\rightarrow}termptr;$ dimbox.c:185,215,258 $ttermptr{\rightarrow}flag = 1;$

Table 5: We hand-analyzed four applications to find the dynamic data structures that account for more than 20% of the hits in merged cache lines, which are illustrated by these code snippets.



(a) L1D writeback distribution



(b) L2 Merged lines distribution

Figure 6: When a line from L1D cache is evicted, it can be merged in L2 cache. In figure (*a*), we show the distribution of lines being written in L2 cache. In figure (*b*), we break down the dirty merged lines into (*i*)lines that are written only once,(*ii*) lines which are written and merged 2 - 5 times, (*iii*) 6 - 9 times, and (*iv*) lines which are written and merged more than 10 times. It can be observed that the amount of write-once data is quite low which illustrates the need for dynamic data merging capability.

cache has one disadvantage, which is that the addressing scheme, which is the key to tractable merging, can lead to an increase of conflict misses in sets with little data similarity. Because all cache lines with the same virtual address belonging to different processes have been mapped to the same set in the cache, if no sharing occurs, this can lead to an increase in conflict misses for this set. This is especially detrimental because the programs are identical, so it is highly likely that all processes will be using the same virtual addresses at the same time. As the number of processes grows, the pressure on the associativity increases, magnifying this problem. This effect is most obvious in *vortex*, where there is a slight slowdown when the Mergeable cache is used.

Because multi-execution workloads are used to execute the same multi-execution workload many times (i.e. when a new design is completed, or a new data point for an earthquake needs to be generated), we find it reasonable to assume that the user will try the machine with and without merging in order to get the best performance. For vortex, the user would turn off the merging capability and suffer no performance degradation.

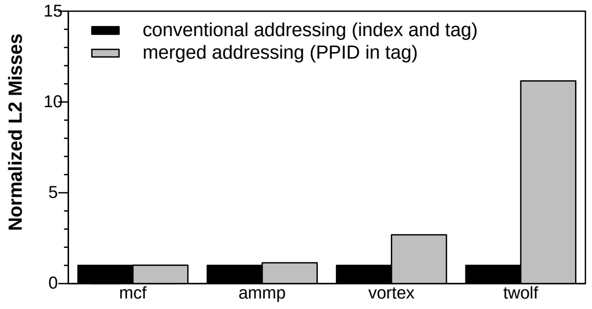


Figure 7: Normalized L2 misses comparing indexing schemes. The conventional cache divides the address between tag, index, and offsets. The new scheme uses bits before and after the PPID for the index, and the PPID and vTag of the tag. No merging is performed to show the effect of only the indexing scheme.

While this solution makes the Mergeable cache usable for all multi-execution workloads, this phenomenon is observed in more than just vortex. An application may have low data similarity either temporally or spatially. Temporally-low data similarity would be phases where a low percentage of the current working set is identical. Spatially-low data similarity would mean that while some sets in the cache have high similarity, another nontrivial number of sets in the cache do not have high similarity, leading to conflict misses in specific sets in the cache. So even applications that benefit overall from the Mergeable cache may have portions of the data that would incur fewer misses in a conventional cache.

Figure 7 shows this phenomenon. In this graph, the two addressing schemes are compared, without the benefit of merging. The graph shows that for *vortex*, the addressing scheme used in the Mergeable cache, in which the PPID is not used in the index, would lead to more than 3× as many L2 cache misses if it were not for some merging. Even more surprising is that in *twolf*, the addressing scheme would lead to 11× more L2 cache misses, yet this is not a poorly performing application using the Mergeable cache. It is only the high degree of merging in *twolf* that leads to speedups. The degradation of benefit in L2 cache misses observed by *vpr* in

Figure 4 is also due to conflicts in non-merged data. Thus, several applications could benefit even more from a hybrid scheme that dynamically divides the cache into two segments - one that performs merging and another that uses conventional addressing. Our initial explorations have shown that the design is non-trivial, with static partitionings giving very mixed results. We will explore such a design in future work.

Using a CAM for data comparisons leads to an increase in design complexity and potential overhead in power usage. However, the Mergeable cache decreases the number of DRAM accesses, thereby reducing the power usage of the bus and DRAM. Using DRAMsim[33], we evaluated the traces of DRAM accesses while running 8 instances of *icsiboost* on a conventional cache and a Mergeable cache. In the presence of a Mergeable cache, a $2GB$ DRAM ($667MHz$) consumes $129.37mW$ on average, whereas $296.15mW$ of power is consumed when a conventional cache is used. In short, a mergeable cache increases processor power usage but reduces overall memory system power usage and demonstrates significant speed up in executing applications from the multi-execution domain.

7. RELATED WORK

Several prior proposals use compiler and architectural support to reduce main memory access and in turn speed up execution. In order to reduce memory stalls, Mahlke et al.[14] propose a profile-guided data partitioning technique. Thread level speculation[11][31] using compiler and architectural support speeds up application execution by spawning speculative threads. Though these techniques speed up execution significantly, with increasing number of cores in a chip, the demand for memory bandwidth is also increased.

In order to reduce memory access, several cache optimization schemes have been proposed. Chang et al. proposed cooperative caching technique[13] in a multiprocessor to reduce off-chip access using a cooperative private cache either by storing a single copy of clean blocks or providing a victim-cache-like, spill-over memory for storing evicted cache lines. An orthogonal study, which has similar motivation as our work, is the data cache compression technique as proposed by Alameldeen et al. [7]. Compressing the L2 data results in reduction of the cache space required to store data. The authors reduce the off-chip accesses and thus save bandwidth. However, compressing and decompressing cache lines add extra overhead in cached data accesses leading to larger access latency.

Kleanthous et al. proposed CATCH[20] to store unique contents in instruction cache by means of hashing, but their proposed system does not support modifications in cached data. In an execution DBI tool such as Valgrid[22], this approach might lead to inconsistencies. Another technique which motivates our approach is the *copy-on-write*[32] mechanism used in virtual machines and operating systems. In the copy-on-write technique, data initially shared by multiple processes become different once one of them writes to it and separated memory regions never merge again. In the VMWare ESX server, content based page searching is performed by using comparison of hashes created from page content. However, data sharing at a page granularity results in low benefit, while increasing the overhead by performing linear search for identical pages. Moreover, VM-based schemes are employed primarily to reduce main memory footprint only by exploiting idle cycles in application execution. In compacting virtual machine memory it is an impactful technique, but reducing memory footprint while running applications not only increases the execution time, but consumes memory bandwidth as well. In our scheme, cache lines are merged at memory write operations and sharing is done at a finer granularity while keeping search latency low. Multiversion Memory[30] stores multiple versions of the data to increase fault tolerance. We take a different approach in this work and propose merging similar data to reduce main memory accesses. Cache line merging can be performed independently from other techniques, and hence can be used as another optimization along with existing techniques.

8. FUTURE WORK

First, we will explore hybrid cache architectures that get the benefit of both the conventional addressing scheme to reduce conflicts and the merging of identical data. We will explore a dynamic cache-partitioning technique which will be able to manage mergeable data more efficiently, adjusting for different amounts of sharing. For applications with low data similarity, dynamic partitioning promises lower system overhead.

Future work will also expand the uses of the multi-execution model. There are several practical scenarios in domains of simulation, visualization, security etc. where multiple instances of the same application are executed with minor variation of parameters or input data *e.g.* the device fabrication process requires many Monte Carlo simulations[21] with minor variation in device parameters to design variation-tolerant devices. In the machine learning domain, ensemble learning[29] technique uses several poor learners to develop finer models. Scientific computing and simulations often require application specific processor[18] designs for faster computations whereas we propose minor modification in cache architecture. More applications of multi-execution exist in the image processing domain as well e.g. the technique of hiding data in images[28] often requires several runs with different parameters to enhance the strength of hiding. It would also be possible to improve performance of redundant multithreading[24][26] based fault tolerant systems using our technique.

Furthermore, the multi-execution paradigm could be further expanded to include simultaneously executing programs that are similar, but not identical. Different programs using many of the same libraries, for example, may be good candidates for multi-execution support. Since we have taken great care to support both low and high-similarity cases in our memory system design, it is our hope that a low-overhead multi-execution system can be used to support a wider scope of applications than explored in this initial study.

9. CONCLUSION

We observe that that a large amount of data is identical across "multi-execution" applications. In this paper, we identify such programs from various domains and propose a Mergeable cache technique which compacts the cache footprint by merging duplicate cache lines owned by different processes and significantly reducing main memory accesses. While running unrelated applications, data merging can be completely bypassed by power gating to reduce power consumption. The Mergeable cache leverages the similarity across "multi-execution" applications to save on cache space and speed up the execution of applications. Specifically, when 8 processors running 8 multi-execution instances share an exclusive 4-MB, 8-way L2 cache, the Mergeable cache speeds up execution up to 6.92× and 2.5× on average, while posing an overhead of only 4.28% in area and 5.21% in power when in use.

10. ACKNOWLEDGEMENTS

The authors would like to thank Mohit Tiwari, Banit Agrawal, Bronis R. de Supinski, Vishwakarma Singh, Krishna Puttaswamy, Ayswarya Sundaram, Darshan Thaker and the anonymous reviewers for providing useful feedback on this paper. This work was supported in part by the National Science Foundation under Grant No. 0855889 and CAREER Grant award MRI-0619911 to Diana Franklin, Grant No. FA9550-07-1-0532 (AFOSR MURI) and NSF 0627749 to Frederic T Chong, Grant No. CCF-0448654, CNS-0524771, CCF-0702798 to Timothy Sherwood.

11. REFERENCES

[1] Tilera TILE Multicore Processors: http://www.tilera.com/products/processors.php.

[2] Ambric Am2000 Family Massively Parallel Processor Array: http://www.ambric.com/products/index.php.

[3] Nvidia GT200 Series: http://www.nvidia.com/object/geforce_gtx_280.html.

[4] PolyScalar: http://users.csc.calpoly.edu/∼franklin/PolyScalar/Home.htm.

[5] SPEC CPU2000: http://www.spec.org/cpu/.

[6] icsiboost: http://code.google.com/p/icsiboost/.

[7] A. R. Alameldeen and D. A. Wood. Adaptive Cache Compression for High-Performance Processors. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 212–223, Washington, DC, USA, 2004. IEEE Computer Society.

[8] A. Asuncion and D. Newman. UCI Machine Learning Repository, University of California, Irvine, School of Information and Computer Sciences, 2007. http://www.ics.uci.edu/∼mlearn/MLRepository.html.

[9] S. Bederman. Cache Management System Using Virtual and Real Tags in The Cache Directory. *IBM Technical Disclosure*, 21(11), April 1979.

[10] V. Betz and J. Rose. VPR: A New Packing, Placement and Routing Tool for FPGA Research. In *FPL '97: Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, pages 213–222, London, UK, 1997. Springer-Verlag.

[11] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August. Revisiting the Sequential Programming Model for Multi-Core. In *Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 69–84, December 2007.

[12] C.-C. Chang and C.-J. Lin. *LIBSVM: a Library for Support Vector Machines*, 2001. Software available at http://www.csie.ntu.edu.tw/∼cjlin/libsvm.

[13] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *ISCA '06: Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 264–276, Washington, DC, USA, 2006. IEEE Computer Society.

[14] M. Chu, R. Ravindran, and S. Mahlke. Data Access Partitioning for Fine-grain Parallelism on Multicore Architectures. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 369–380, Washington, DC, USA, 2007. IEEE Computer Society.

[15] Douglas C. Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.

[16] K. C. Elliott. Varieties of Exploratory Experimentation in Nanotoxicology. *History and Philosophy of the Life Sciences*, 29(3), 2007.

[17] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *IEEE Computer Architecture Letters*, 1(1):7, 2006.

[18] T. Kurihara, E. Kamada, K. Shimada, and T. Shimizu. A RISC Processor for SR8000: Accelerating Large Scale Scientific Computing with SMP. In *IEEE Symposium on High Performance Chips(HOT CHIPS)*, 1999.

[19] F. Lozano and P. Rangel. Algorithms for Parallel Boosting. In *ICMLA '05: Proceedings of the Fourth International Conference on Machine Learning and Applications*, pages 368–373, Washington, DC, USA, 2005. IEEE Computer Society.

[20] M. Kleanthous and Y. Sazeides. CATCH: A Mechanism for Dynamically Detecting Cache-Content-Duplication and its Application to Instruction Caches. In *Design, Automation and Test in Europe, 2008 (DATE '08)*, pages 1426–1431.

[21] M. Nedjalkov, H. Kosina, and S. Selberherr. Monte Carlo Algorithms for Stationary Device Simulations. *Mathematics and Computers in Simulation*, 62(3-6):453–461, 2003.

[22] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, USA.

[23] J.-K. Peir, W. W. Hsu, and A. J. Smith. Functional Implementation Techniques for CPU Cache Memories. *IEEE Transactions on Computers*, 48(2):100–110, 1999.

[24] J. Ray, J. Hoe, and B. Falsafi. Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery. In *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture, (MICRO-34)*, pages 214–224, December 2001.

[25] Robert E. Schapire. The Strength of Weak Learnability. *Machine Learning*, 5:197–227, 1990.

[26] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, pages 84–91, Washington, DC, USA, 1999. IEEE Computer Society.

[27] S. Wilton and N. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.

[28] K. Solanki, N. Jacobsen, S. Chandrasekaran, U. Madhow, and B. S. Manjunath. High-Volume Data Hiding in Images: Introducing Perceptual Criteria into Quantization Based Embedding. In *Proceedings of the IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP)*, volume 4, pages 3485–3488, May 2002.

[29] P. Sollich and A. Krogh. Learning With Ensembles: How Overfitting Can Be Useful. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 190–196. The MIT Press, 1996.

[30] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Fast Checkpoint/Recovery to Support Kilo-Instruction Speculation and Hardware Fault Tolerance. (TR-1420), October 2000.

[31] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede Approach to Thread-Level Speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[32] C. A. Waldspurger. Memory Mesource Management in VMware ESX Server. *SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[33] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMsim: A Memory System Simulator. *ACM SIGARCH Computer Architecture News*, 33(4):100–107, 2005.