

Conflict-Avoidance in Multicore Caching for Data-Similar Executions

Susmit Biswas, Diana Franklin, Timothy Sherwood and Frederic T. Chong

Department of Computer Science
University of California, Santa Barbara,
Santa Barbara, California, USA 93106

{susmit, franklin, sherwood, chong}@cs.ucsb.edu

Abstract—

Power density constraints have affected the scaling of clock speed in processors, but following Moore's law we have entered the multicore domain and we are about to step in the era of manycores. Harnessing the full potential of large number of cores is a challenging problem as shared on-chip resources such as memory subsystem, interconnect networks become the bottlenecks. One easy and popular way of utilizing parallelism in large scale systems is by running multiple instances of the same application as we observe in many domains such as verification, security etc. and we term it as "multiexecution." This model of computation will probably become more popular as the number of cores in a processor grows.

We identify that leveraging the similarity in data across the instances of an application by dynamically merging identical data in a cache can reduce the off-chip traffic and thereby, lead to faster execution. However, dissimilarities in content increase the competition for cache lines as well. In this paper we explore the design space of hybrid mergeable cache architecture that places dissimilar data blocks in a conventional cache and thereby, enables us to exploit data similarity more efficiently by reducing the conflicts. We experiment with benchmarks from various multi-execution domains and show that our hybrid mergeable cache design leads to an average of 9.5% additional speedup over Mergeable cache while running 8 copies of an application, with an overhead of less than 1.34% in area.

I. INTRODUCTION

As the difficulties of uniprocessor performance scaling have proven economically daunting, designers have turned to bandwidth (parallelism), rather than latency, to scale performance in future microprocessors. This approach has led to two significant challenges. First, as a single reference stream scales to tens, hundreds, or even thousands of reference streams, the memory system will struggle to service all of the requests in a timely manner. Second, careful programming will be required to parallelize applications to hundreds of cores. Programming both correct and efficient parallel code is challenging, and too few programmers have the expertise to accomplish both.

One easy though easily overlooked way of using the cores is by running the same application with different input sets or configurations to solve a larger problem. We find that many applications from different domains such as simulations, security etc. are already used in this fashion, but the applications are run either serially or in parallel to exploit the computation power of clusters. We term this model of

computation where same program is run multiple times, but with different input data or parameters, as "multi-execution." We believe that multi-execution could become a useful model of execution as multicores scale, because it is already used in many domains, and because no software changes are necessary to take advantage of a multicore system.

Note that an alternative to multi-execution is to write an explicitly parallel program that takes many instances of an application and explicitly shares redundant data. This approach, however, 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.

We explore the characteristics of several example applications from simulation, optimization, database, and learning domains. We find that the similarity of multi-execution working sets can be quite high, but careful design is needed to profitably exploit this similarity in a memory system. Our previous work used a *Mergeable cache*, which dynamically merges cache lines containing identical data from different instances of the same program running under multi-execution [3]. While highly successful when data is successfully merged, we observed that our approach can lead to substantial cache conflicts when the data remains different and is mapped to the same line. In other words, there is a fundamental tradeoff between mapping data to the same line to find similarity and coping with data that is, in fact, not similar.

In this paper we explore the domain of hybrid Mergeable caches where a pure Mergeable cache is augmented with a conventional victim cache to store dissimilar data blocks. A hybrid Mergeable cache reproduces the benefit lost due to conflicts in allocating cache lines without posing significant benefit. In this paper, we present cycle-accurate simulation results with 3 applications from different domains which suffer from increase in conflict misses.

The remainder of the paper is organized as follows. We motivate our approach in section II, and explain the challenges and techniques of implementing hybrid Mergeable cache architecture in section III. We illustrate the experimental methodology in section IV and show results in section V. In section VI, we discuss previous approaches to reduce memory accesses, and finally conclude in section VII.

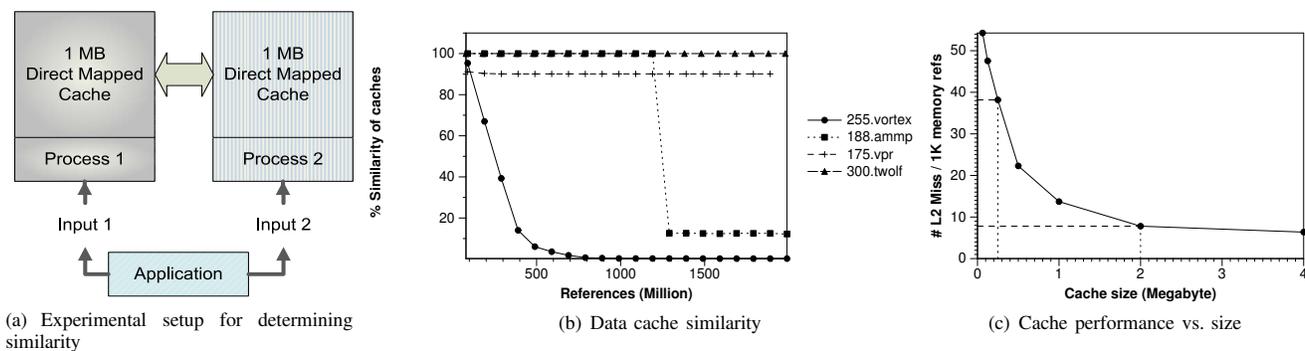


Fig. 1. In order to determine the similarity between two executions we compared cache contents of two cache simulation for two different input sets as shown in (a), and we observed that there exists high similarity across two execution of the same application. Figure 1(b) and 1(c) are borrowed from [3]. In (b) we show the similarity of cache content for four applications where 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. By merging identical cache blocks from different processes we can increase the effective cache capacity per core and thereby, improve cache performance. As cache performance has a non-linear relationship with cache size, merging duplicate cache lines to reduce cache requirements has the potential to improve cache performance substantially, even with small increases in effective cache capacity.

II. MOTIVATION

Today processor manufacturing industry has stepped into the multicore and manycore domain as scaling processor clock speed stumbled against power density wall. One easy way to use these cores effectively without writing efficient and explicitly parallel program, as stated in our prior work [3] is by running multiple instances of the same application with different input sets. This model of computation is already in use in several domains such as simulations, security etc. In a set of experiments, similar to one showed in Figure 1(a), they observed that there exists high data similarity across the executions of the same application (Figure 1(b)). We proposed a Mergeable cache design that identifies identical data blocks dynamically and keeps a single copy of them, increasing the cache space and thereby, improving application performance as shown in Figure 1(c).

The Mergeable cache[3] 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 minor power and area overhead. Despite these benefits, the Mergeable cache has one disadvantage. 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 are mapped to the same set in the cache, in regions of low data similarity, this can lead to an increase in conflict misses for this set as each cache block from different processes will be placed in different cache line, thereby, increasing capacity pressure on the cache. As the processes are run using the same application, it is highly likely that all processes will be using the same virtual addresses at the same time, and 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. 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, and 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.

We illustrate this phenomenon in Figure 2(c). 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 \times 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 \times 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. 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. The conventional segment of the cache is used as a victim cache to Mergeable segment to reduce conflict misses.

III. DESIGN

We observed in Figure 2(c) that dissimilarity in data leads to an increase in conflicts as page coloring enforces mapping blocks with the virtual address to the same cache set. This conflict can be avoided if the same virtual address from all processors are not mapped to same cache set. However, this addressing scheme is required to keep merging candidates in the same cache set as the goal of reducing conflict misses and grouping merging candidates are conflicting.

We resolve this conflict by using a *hybrid Mergeable* cache where a Mergeable cache is assisted by a small conventional victim cache. Data lines which can be merged reside in a Mergeable segment, and lines with dissimilar content

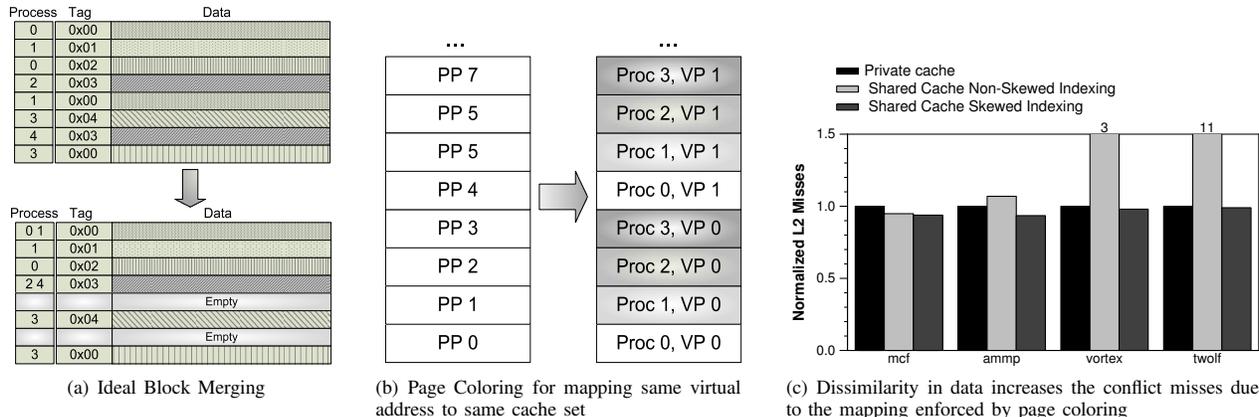


Fig. 2. In order to achieve ideal block merging as shown in (a), a page coloring scheme is used that enforces mapping of same virtual address to the same cache set in a Mergeable cache, but we show in (c) that this page coloring scheme leads to increase in conflict misses. We show the comparison of normalized L2 misses for two cache addressing schemes - Mergeable and conventional. The conventional cache divides the address between tag, index, and offsets. A pure Mergeable cache[3] uses bits before and after the PPID for the index, and the PPID and vTag for the tag. No merging is performed to show the effect of only the indexing scheme.

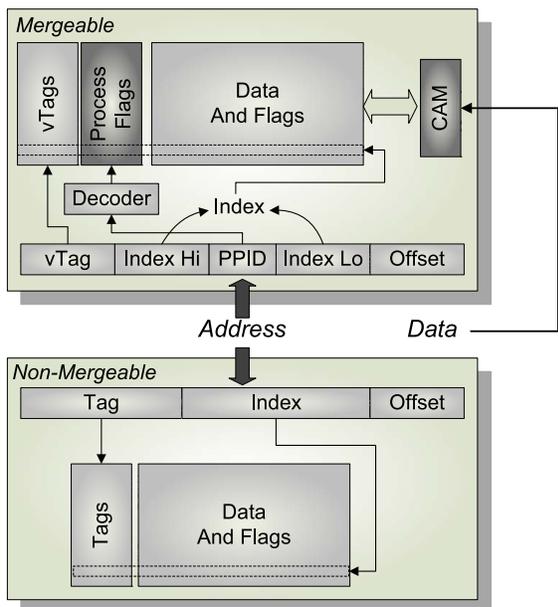


Fig. 3. VA-Mergeable cache architecture. Modifications to traditional cache are indicated as shaded blocks. The cache is partitioned in Mergeable and non-Mergeable(traditional) cache segments. When a line is evicted from L1 and moved to L2, all lines of the cache set which the evicted line gets mapped to, are copied in CAM and compared for identical content.

are moved to a non-Mergeable cache. Using conventional addressing in the non-Mergeable segment enables us to avoid the conflicts induced by page coloring. When a cache line is evicted from an upper-level cache, it is written to the Mergeable segment at first. If a cache line is not able to merge with any pre-existing cache line having the same address and data, it replaces an old line, which is moved to the non-Mergeable segment. We term the hybrid Mergeable cache as VA-Mergeable (Victim Assisted) cache. We experimented with different sizes of the victim cache and found 16-KB, 16 way

cache to be an appropriate size as it provides a good balance between performance and overhead. The LRU replacement policy in the Mergeable cache ensures that lines shared by more processes are not replaced by lines having low sharing unless they have not been accessed recently.

A VA-Mergeable cache provides the advantage of utilizing the full cache when data similarity is low without increasing the cache miss rate significantly, yet storage efficiency is increased by merging identical lines in the Mergeable cache. This benefit comes at a price - the Mergeable cache was able to evict dirty merged lines, allowing a single write-back with an enhanced address to write to all blocks. With the hybrid approach, this benefit is lost, and we will see that this can reduce the benefits of the victim cache approach.

a) *Architectural details*:: In order to support dynamic data merging, minor modifications to the cache architecture are necessary as shown in Figure 3. Though our implementation performs merging in the L2 cache, this technique is applicable in all lower level shared caches. The Mergeable segment uses the similar technique as used in our prior work[3].

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 I
OVERHEAD OF 256-BIT WIDE CAM OBTAINED USING CACTI 4.2 FOR 45nm TECHNOLOGY NODE.

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 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 I.

In this paper we model a set-associative cache with 8-way associativity partitioned into 8 banks similar to the AMD Opteron’s cache. Using Cacti[10] we find that the area of an 8-way CAM is $0.0156mm^2$. Cacti reports cache configurations where three knobs, area, delay and power are tuned to find an optimal configuration that satisfies an objective function. Inspecting different cache configurations we compute the power, delay and area of a Mergeable and VA-Mergeable cache and then compare them to a conventional cache in Table II. We chose the appropriate configurations by finding the design point closest to a conventional cache. In our study we pessimistically assume that accessing victim cache requires two cycles. As the victim cache is accessed only upon a miss in the Mergeable segment, we calculate the power usage of the victim cache (assuming 25% miss rate of Mergeable cache which is found to be the worst in *vortex*) as $1/4^{th}$ of the power calculated using Cacti. Overall, VA-Mergeable cache poses an overhead of 1.43% in area and additional 1.22% power overhead.

IV. METHODOLOGY

In this section, we describe our experimental methodology and simulation framework which is built on the PolyScalar[1] multiprocessor simulator. PolyScalar is a multi-processor version of the Simplescalar[7] simulation tool and uses PISA as the instruction set architecture. The processor configuration is described in Table III.

We simulate multiple cache architectures in this work. In all these caches we assume an exclusive policy in order to maximize the number of unique lines that can be stored on-chip. In other words, no line can be contained in more than one cache at any time. We perform our experiments with the following cache architectures.

- 1) *Shared Cache*: processors share a large L2 cache.
- 2) *Shared Cache with victim cache*: the L2 cache is augmented with a 16 KB, 16 way victim cache. We evaluate this configuration to illustrate that the speedup in a hybrid cache is not a result of the increase in cache capacity for to a victim cache. We restrict the cache size to 16 KB as victim caches have diminishing returns while overheads increase with the size of it.
- 3) *Mergeable Cache*: processors share a large, Mergeable L2 cache in which cache lines are merged if and only if their contents match.
- 4) *VA-Mergeable Cache*: processors share a large L2 cache that is composed of two segments - Mergeable cache and non-Mergeable victim cache which resembles the victim cache in the second configuration.

We simulate both instruction and data memory accesses in L2 cache but keep 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. In our simulations, one core is allocated to a single process. However, process migration can be supported because our hardware uses the physical address based PPID, not the processor ID. In our cache architecture, process migration does not pose any issue as memory pages and L2 cache lines do not need to be moved in case of migration.

We selected benchmarks from several domains such as simulation, visualization, machine-learning etc. where multi-execution is used in practice. For all the applications practical scenarios are used to construct input and parameter variations. In this work we selected applications that suffer from the increase in conflict misses due to constraints in page-coloring.

We selected three benchmarks from the SPEC2000-Cpu suite and use SPEC2000 train inputs for 300.twolf, 175.vpr, 255.vortex. Note that Simpoint based simulations are not feasible in our experiments as warming up the large L2 cache has large impact on the accuracy of the simulation. Instead, we run simulations until completion to ensure that results are not skewed due to the initialization phase. Characteristics of these applications are summarized in Table IV.

V. RESULTS

Figure 4 shows the performance of a conventional and Mergeable cache with and without a victim cache. These results show the tradeoffs between three separate influences. The Mergeable cache provides two benefits that decrease the bandwidth to DRAM - merging identical data to reduce L2 misses and writing back merged data to reduce the number of lines written back. The disadvantage, though, is the increase in conflicts due to the addressing scheme in sets where little merging occurs. In the pure Mergeable case, we can see this phenomenon. The victim cache reduces the number of L2 misses due to conflicts, but it splits the merged data, increasing the number of write-backs occurring. We can see that with twolf and vpr, as the number of processes increases, the opportunities for merging increase. These opportunities in some sets are partially offset by the conflicts in other sets. The victim cache provides substantial improvement, 15% with 8 processes. Vortex, on the other hand, suffers more from the increase in write-backs than the decrease in L2 conflict misses, leading to a slowdown of 1.75%. In the 8-process case, adding a victim cache to the conventional cache increases performance by 8.58%, and a pure Mergeable cache improves performance by 29.79%. The combination of a Mergeable cache and victim cache provides an additional 9.5% speedup beyond the Mergeable cache alone.

VI. RELATED WORK

Several prior proposals use compiler and architectural support to reduce main memory access and in turn speed up execution. In our previous work [3] we proposed Mergeable cache architecture which is unable to address the issue of increase in conflicts due to page coloring based mapping constraints. Our proposal for VA-Mergeable cache addresses the shortcomings and recovers the lost benefit.

In order to reduce memory stalls, Mahlke et al.[6] proposed a profile-guided data partitioning technique. Thread

Type	Banks	Area(mm ²)	Latency(ns)	Dyn Rd Pow(mw)	Dyn Wr Pow(mw)
Conventional	8	33.65	2.62	1173.69	1343.28
Mergeable	8	33.91	2.63	1168.99	1342.81
VA-Mergeable	8+2	33.91+0.19 = 34.10	2.63, 0.65 (victim)	1168.99+ 69.8829/4 = 1186.46	1342.81 + 74.0376/4 = 1361.32

TABLE II

ASSUMING 8 PROCESSORS SHARING A 4MB-8WAY SET ASSOCIATIVE CACHE, WE CALCULATED OVERHEAD OF TAG ARRAYS AND BIT VECTORS IN A MERGEABLE, VA-MERGEABLE AND CONVENTIONAL CACHE USING CACTI 5.3. AS USING PPIID AS BITVECTORS INSTEAD OF BITS IN TAG ARRAY REDUCES THE CAM, THE OVERHEAD IN AREA AND POWER ARE 1.34% AND 1.21% (AVERAGE OF READ AND WRITE POWER) RESPECTIVELY. THESE NUMBERS DO NOT INCLUDE THE OVERHEAD FOR THE CAM USED FOR MERGING CACHE BLOCKS.

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 16 KB, 16 way victim cache
ALU/FPU/Mult/Div	4/4/1/1	L2 Latency	6 Cycles
Branch Predictor	2-level, 1024 Entry History Length 10	L1 I-Cache	32KB + 32 KB, Direct Mapped
BTB size	2048	L1 D-Cache	32 byte lines
RAS entries	8	L1 I,D-Cache Ports	4
		L1 Latency	1 Cycle

TABLE III

CONFIGURATION OF THE SIMULATED PROCESSORS. WE DO NOT INCREASE THE SIZE OF THE L2 CACHE WITH THE NUMBER OF PROCESSORS SHARING IT. THE PARAMETERS USED IN SIMULATIONS ARE CHOSEN JUDICIOUSLY FROM STATE OF THE ART PROCESSORS.

Benchmark	Description	Input Modification	Run Length	Footprint	
				rsz	vsz
175.vpr	FPGA Place and Route	routing-channel-width	3.3 B	2.67	1.35
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

TABLE IV

BENCHMARKS AND INPUT DESCRIPTIONS. THE OBSERVED MEMORY FOOTPRINT SIZES (IN UNITS OF MEGABYTES) REPORTED ARE AVERAGE OF 20 RUNS WHERE RESIDENT MEMORY SIZE AND VIRTUAL MEMORY SIZE ARE ABBREVIATED AS *rsz* AND *vsz* RESPECTIVELY.

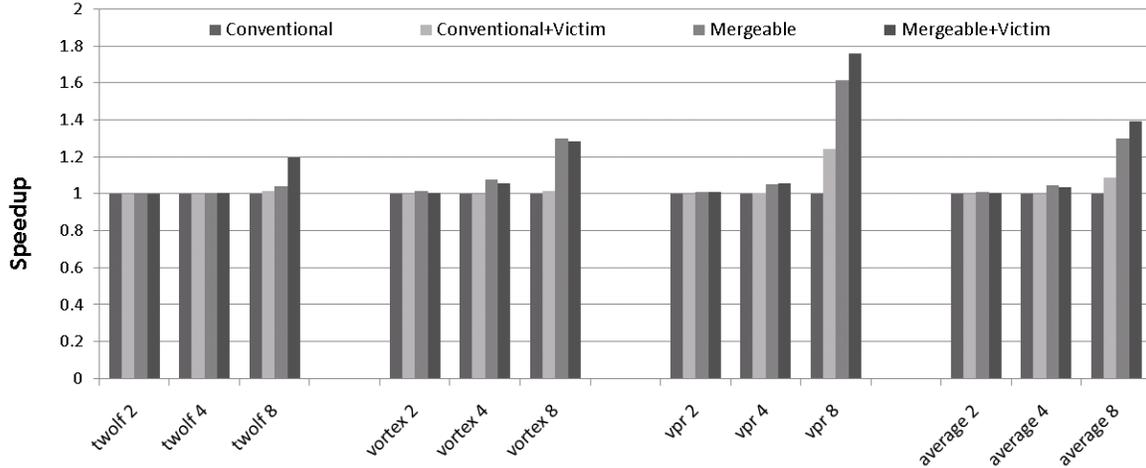


Fig. 4. Speedup for all benchmarks simulated with 4-MB, 8-way L2-cache with 32B blocks running 8 instances of each application. Mergeable cache show speedup in all benchmarks, but suffers from alignment of dissimilar data in the same cache set resulting in increased conflict misses. VA-Mergeable cache addresses this problem and improves performance by 9.5% on average.

level speculation[4][12] 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. Several cache optimization schemes have been proposed for reducing memory access. Chang et al. proposed cooperative caching technique[5] 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. [2]. Compressing the L2 data results in reduction of the cache space required to store data, and also the off-chip accesses and bandwidth. However, compressing and decompressing cache lines add extra overhead in cached data accesses leading to larger access latency.

Kleanthous et al. proposed CATCH[8] 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 Valgrind[9], this approach might lead to inconsistencies. For data caches where block contents change frequently, this technique, being oblivious of the sharing by other processes, leads to inconsistent behavior. Another technique which motivates our approach is the *copy-on-write*[13] 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[11] 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.

VII. CONCLUSION

In our prior work [3], we introduced the notion of “multi-execution” which is used in practice in many domains. Multi-execution refers to the scenario where an application is run with different parameters or inputs to sweep a design space or solve a larger problem. We proposed Mergeable cache that dynamically merges identical cache blocks to increase effective cache capacity per core and improves performance of applications significantly. However, in our study we observed that page coloring based mapping constraint forces two dissimilar blocks to be mapped to same cache set, and in turn increases cache conflicts. In this paper we explore the design space of a hybrid Mergeable cache and show that a conventional victim cache (non-Mergeable) assisted Mergeable (VA-Mergeable)

cache is able to recover the lost benefits. In applications or application-phases with low similarity, VA-Mergeable cache has the capability to improve performance over Mergeable cache by 9.5% in average and up to 15%.

In our experiments we observed that using a conventional cache as the victim cache leads to multiple *writebacks* to DRAM of the identical data, which a pure Mergeable cache can exploit by merging writes. The effect is most prominent in *vortex* where a pure Mergeable cache performs best. In our future work, we aim to address this issue by reducing the number of *writebacks* to DRAM.

VIII. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under CAREER Grant No. 0855889 and 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.

REFERENCES

- [1] PolyScalar: <http://users.csc.calpoly.edu/~franklin/PolyScalar/Home.htm>.
- [2] 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.
- [3] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong. Multi-Execution: Multicore Caching for Data-Similar Executions. In *Proceedings of the International Symposium on Computer Architectures (ISCA'09)*, June 2009.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. *SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.