Improving Data Cache Performance with Integrated Use of Split Caches, Victim Cache and Stream Buffers

Afrin Naz, Mehran Rezaei, Krishna Kavi and Philip Sweany
Department of Computer Science and Engineering
The University of North Texas, Denton, Texas 76206-1366
Email: kavi@cse.unt.edu

Abstract

In our prior work we explored a cache organization providing architectural support for distinguishing between memory references that exhibit spatial and temporal locality and mapping them to separate caches.

The work showed that using separate (data) caches for indexed or stream data and scalar data items could lead to substantial improvements in terms of cache misses. In addition, such a separation allowed for the design of caches that could be tailored to meet the properties exhibited by different data items.

In this paper, we investigate the interaction between three established methods, split cache, victim cache and stream buffer. Since significant amounts of compulsory and conflict misses are avoided, the size of each cache (i.e., array and scalar), as well as the combined cache capacity can be reduced. Our results show that on average 55% reduction in miss rates over the base configuration.

Key Words: Array Cache, Scalar Cache, Victim Cache, Stream Buffer, Memory Access Time.

1. Introduction

In this paper, we investigate methods for improving hit rates in the first level of memory hierarchy and show that the inclusion of a victim cache and a stream buffer together with partitioned cache architectures provides an effective solution for alleviating existing problems in cache designs and enhancing the effective cache memory space for a given cache size and cost.

The success of cache memories has been explained using the concept of locality (either temporal or spatial) of reference [2]. Temporal locality implies that, once a location is referenced, there is a high probability that it will be referenced again soon, and less likely to do so as time passes; spatial locality implies that when a datum is accessed it is very likely that nearby data will be accessed soon. Since cache stores recently used segments of information, the property of locality implies that needed information is also likely to be found in the cache. Computer architects have proposed smart cache control mechanisms and novel cache architectures that can detect program access patterns and can fine-tune some cache policies to improve overall cache utilization and data locality. Among these techniques are associative caches, prefetching mechanisms [3], cache bypassing [4], victim caches [6], column-associative caches [7], stream buffers [6], split caches [8]-[19], and multi-port caches. In his paper Jouppi [6] proposed both victim caches and stream buffers. Victim caches are based on the fact that reducing the cache misses due to line conflicts for data exhibiting temporal locality is an effective way to improve cache performance, whereas stream buffers are oriented towards eliminating cold misses coming from the portion of the code exhibiting spatial locality. A split cache provides architectural support for distinguishing between memory references that exhibit spatial and temporal locality and mapping them to separate caches in order to implement different configurations exploiting different cache parameters selectively and effectively. Each of these approaches has its strengths and works well for the patterns it is designed for. So far, split caches have primarily been used independently of the other two approaches. To date, no split cache has considered the existence of a victim cache or a stream buffer and their interaction on data references. Similarly, a victim cache or stream buffer does not normally consider what optimizations have already been incorporated by locality-enhancing split cache techniques. In this work we use all these three techniques together and study the interaction among them. We propose an integrated scheme that partitions the program into regions, each with its own locality type. Our approach then sends the partitioned memory references to appropriate caches and finally selectively applies either a victim cache for program regions exhibiting temporal locality or a stream buffer for regions with spatial locality, to further enhance the split cache organization.

The rest of the paper is organized as follows. Section 2 discusses related issues and performance metrics in more detail. Section 3 provides a survey and analysis of related research. Section 4 describes the experimental method used in our evaluation while, section 5 presents the results. Section 6 provides a brief

synopsis of our work, drawing conclusions from our experimental results.

2. Concepts

Since our approach combines three different techniques in a single framework, we will first describe how each of them concepts before describing our integrated approach.

2.1. Split Cache and its functionality

Separation of cache is not a new idea. Modern processors rely on split cache architectures, at least at the first cache level, with separate instruction and data caches. Conventional data caches imply no separation based on the nature of the locality exhibited by different data references, handling all memory references in a uniform manner - whenever a reference misses, a new block is brought into cache at the expense of replacing another block. Since not all data items exhibit both spatial and temporal localities, this simple treatment to references makes the data cache inefficient at adapting to the two types of localities. Generally, caches exploit temporal locality by retaining recently referenced data for a long time, and spatial locality by fetching multiple neighboring words as a cache block on a cache miss. If a data item exhibits no temporal locality, bringing it into the cache is useless. Likewise if no spatial locality is exhibited by data items, bringing an entire cache block is needless. Thus traditional treatment of cache misses causes unnecessary movement of data among the levels of the memory hierarchy, causing significant interference between unrelated data inside the cache, leading to the removal of potentially useful data, causing cache pollution and unnecessary increases in miss ratio, memory access time and memory bandwidth.

In order to solve these problems, several split cache architectures have been proposed: Dual cache[9], [12], Split Temporal/Spatial(STS) [10], Split Spatial/Non-Spatial cache (SS/NS) [14], array and scalar cache[13], HP-7200 Assist cache[8], Non-Temporal Streaming (NTS) [11] and Minimax cache[18]. In our prior work [1], we proposed a split cache architecture that grouped memory accesses as scalar or array references according to their inherent locality and each group was subsequently mapped to a dedicated cache partition, equipped with architectural constructs built to exploit that particular locality type selectively and effectively. The “array cache” was a direct mapped cache with larger block sizes to exploit spatial localities more aggressively by prefetching multiple neighboring small blocks on a cache miss. Whereas the “scalar cache” was a 2-way set associative cache with smaller block sizes to exploit temporal locality. In this system, since scalar references and streamed references no longer negatively affected each other, cache interference, thrashing and pollution problems were diminished, delivering better performance. Not only both caches would be designed more optimally according to their specific needs, it would simplify some other general issues and concerns in cache design, such as the associativity, cache block size or cache capacity.

2.2. Victim Cache and its functionality

Victim cache was originally proposed by Jouppi[6] as an approach to reduce the conflict misses of direct mapped caches without affecting its fast access time. Victim cache is a fully associative cache, whose size is typically 4 to 16 cache lines, residing between a direct mapped L1 cache and the next level of memory. On a main cache miss, before going to the next level, the victim cache is checked. If the address hits in the victim cache the desired data is returned to the CPU and also promoted to the main cache by replacing its conflicting competitor. The data evicted from the main cache is transferred to the victim cache. In case of a miss in victim cache the next level of memory is accessed and arriving data fills the line in main cache while moving the current data to victim cache. In this case the replaced entry in the victim cache is discarded and, if dirty, written back to the next level of memory.

The design of a first level cache always involves fundamental tradeoffs between miss rate and access time. Direct mapped caches are simpler, easier to design and require less silicon area than set associative caches. The main disadvantage of a direct mapped cache is the high conflict miss rate -- conflict misses typically account for 40% of all direct-mapped cache misses [6]. Conversely for caches with higher associativity the main advantage is lower miss rate, but they are more expensive and incur longer access times on a hit. The goal of a computer architect is to maximize performance while staying within cost and power constraints. Addition of a victim cache can ease this problem by reducing the conflict miss rate to the same extent as a set associative cache, but at the same time maintaining the critical hit access path of a direct mapped cache. Victim cache temporarily holds data evicted from the cache and, because of its full associativity, it can simultaneously hold many blocks that would conflict in direct mapped cache. If the number of conflicting blocks are small enough to fit in victim cache, both the miss rate to the next memory level and the average access time will be improved due

to relatively low miss penalty for fetching from victim cache.

2.3. Stream Buffer and its functionality

Complementing the cache with a small stream buffer to exploit spatial localities was also first proposed by Jouppi [6]. The stream buffer is a fully associative, FIFO buffer with 4 or 5 entries designed to support direct mapped cache through prefetching. A miss will induce the prefetching of the missed block along with successive blocks that will be stored in the buffer rather than the cache. The intent is to use the stream buffer to avoid cache pollution (premature displacement of data). For data with spatial locality prefetching is always a good solution. Although increasing line size is the simplest way of prefetching, line sizes cannot be made arbitrarily large without increasing miss rates and greatly increasing the amount of data to be transferred [6]. Other conventional prefetching methods also have their deficiencies. The stream buffer not only mitigates traditional problems with larger cache lines and extensive prefetching, it is more effective than other investigated prefetch techniques [6]. The biggest problem with stream buffers is that they need to be flushed at the detection of any non-spatial data. Jouppi's investigation did not explore the stream buffer only for data with spatial localities (such as streams). Rather the buffer was used for all data items.

2.4. Functionality of the integrated approach

So can we design a combined approach that provides even better performance than either applying only one or applying each independently? Until now there has not been significant research investigating the interaction among these three optimizations (viz., split caches, victim cache and stream buffer). We already have shown that using separate (data) caches for indexed or stream data and scalar data items can lead to substantial improvements in terms of cache misses [1]. Although victim caches and stream buffers can reduce miss rates in L1 cache, the reduction achieved depends on the configuration of the cache, as well as the data reference types. Now we will see how a separation of caches can be tailored to meet the requirements of victim cache and stream buffer.

A conflict miss occurs when data with temporal locality is referenced twice but is replaced by another data item in between the references. Victim caching is based on the principle of temporal locality and provides dynamic associativity by allowing up to N+1 conflicting blocks, which belong to the same direct-mapped set, to co-exist in caches simultaneously, where N is the number of block entries in the victim cache. In his original paper, Jouppi implemented a victim cache for a unified data cache. As a result array or stream elements remove scalar data from the victim cache causing expensive victim cache pollution. In our work, as the array references are removed from the scalar cache, the victim cache not only has to deal with fewer references but also without being polluted by stream references. The reduced cost of using small victim cache with direct mapped cache outweighs the performance gains of having a cache with large associativity.

On the other hand, a cold miss occurs when stream or array data are traversed linearly by using the elements only once or very few times during traversals. Stream buffers exploit spatial locality and perform prefetching for stream or array data. Jouppi's analysis [6] also included the stream buffer for a unified data cache and every time scalar data is detected the whole buffer needed to be flushed. In our study because we are removing the contaminating scalar data the performance can increase significantly.

We believe, that while transistors are plentiful in current VLSI designs, it is useful to allocate more resources to allow intelligent control over latency reducing techniques and that it is better to implement multiple smaller dedicated caches because these can be accessed relatively quickly. In our framework, the compiler will separate data references according to their inherent locality type and send them to appropriate cache. In this study the promising aspects of victim cache in keeping conflicting blocks will be used to satisfy the requirements of scalar cache and the prefetching ability of stream buffer will be included with the array cache to exploit its advantages for streaming data. Compiler has a global view of the program that is not visible to hardware, which on the other hands gathers information during runtime. Our objective is to combine runtime and compile time information to take full advantage of both.

3. Related Work

According to our knowledge no work has been reported presenting the integration of these three approaches. For that reason in subsection 3.4 we compared our work with the most closely related work by Johnson et al.[4]

3.1. Split Cache

Valero et al. [9] have proposed a dual data cache, which is composed of two modules, a temporal module

which is a fully associative buffer and spatial module, which is a direct mapped cache targeted to exploit spatial locality. At the compile time memory instructions are tagged as bypass, spatial, or temporal.

In the STS (Split Temporal/Spatial) cache proposed by Milutinovic et al. [10] the temporal part is organized as a two level hierarchy with one word block size, whereas the spatial part is one-level with four 32-bit words and a hardware implemented prefetching mechanism. In a later study Milutinovic et al. [14] proposed a new split cache design, called the Split Spatial/Non-Spatial cache (SS/NS), which used a flag based method for detecting different types of locality.

The NTS (Non-Temporal Streaming) cache proposed by Rivers and Davidson [11] dynamically detects temporal (T) and non-temporal (NT) data and cache them separately. The NTS cache system includes a non-temporal detection unit (NTDU) to monitor the reuse behavior of the blocks. Lee et al. [15] have proposed a split cache system called STAS cache. In this system on every memory access, both modules are accessed simultaneously. Later they proposed an SMI cache [16] that is an extended version of STAS with a prefetching unit. There have been more studies of split cache which include array/scalar cache [13], HP-7200 Assist cache [8].

In the arena of embedded processors, static or dynamic cache partitioning are even more popular. Most prominent works include Minimax cache [18] and Intel's StrongARM SA-1110 [17]. Ranganathan et al. [19] and many others have proposed reconfigurable caches for embedded systems with dynamic cache partitioning.

3.2. Victim Cache

Albera and Bahar [20] combined software code placement and associative-buffer solutions for high performance processors and showed that the buffer can improve performance even more after code layout optimization is applied than when it is used without the code optimization. In a later study Bahr *et al* compared the use of victim caches to more traditional techniques and showed that use of a victim cache is usually a better choice for both power and performance [21].

Espasa and Valero [22] considered the usefulness of adding a victim cache next to the register level of a vector processor and showed that it can provide speedups by allowing a good tolerance of large memory latencies.

Hormdee *et al* proposed an architecture of a self-timed victim cache with a forwarding mechanism suitable for use within an asynchronous environment [23].

While the average performance and energy improvements obtainable using a victim cache are well known for general purpose computers, in the arena of embedded systems where power and time savings are extremely important, the extra one cycle needed to check victim cache may become wasteful and dramatically degrade performance if victim cache hit rate is low. Zhang and Vahid proposed that adding a victim cache as a configurable parameter will be imperative for embedded system designers to fully take advantage of victim cache based on application's specific requirements [24].

Except for the addition of the non-swapping option, no other extension to Jouppi's original victim cache [6] was implemented by any of these above mentioned studies. Bahar *et al* [21] tried to add some extra flavor in their "penalty buffer" but did not gain much improvement. Only one group, Stiliadis *et al* had proposed an improvement of victim caching called "Selective Victim Caching" [25]. In this method a prediction scheme based on each block's past history of utilization is used to selectively place a block either in the main cache or victim cache on a cache miss in either cache and to decide whether to perform swap or not in the case of victim hit.

3.3. Stream Buffer

The most extensive and prominent work with stream buffers is that of McKee et al. [26]. They designed an SMC (stream memory controller), which is a combination of a small buffer and an intelligent scheduling unit for supporting regular cache. Palacharla and Kessler [27] proposed the use of multiple stream buffers to replace big secondary cache.

3.4. Combination of Victim Cache, Stream Buffer and Cache separation techniques

To date, no study has combined the implementation of victim cache and stream buffer with separated data cache approach. Johnson *et al* [4] proposed a method where a single 4–way set associative buffer is used to serve the function of both victim cache and stream buffer on groups of data that have been differentiated based upon the reuse behavior. While we also regroup data by locality analysis and implement both victim cache and stream buffer, our work differs with [4] in several key aspects.

Johnson *et al* [4] presented a method to improve the efficiency of cache by bypassing data that is expected to have little reuse in cache and allowing more frequently accessed data to remain cached longer. The bypassing choices are made by a Memory Address

Table (MAT), which analyzes the usage patterns of the memory locations accessed. In order to characterize memory locations they introduce the notion of macro-block, which is a group of statically defined blocks of memory with uniform size (1k bytes). They used a direct mapped 16k L1 data cache and 256k L2 data cache with fully associative buffers of 8 and 256 entries respectively, which hold bypassing data and are accessed in the same manner as a victim cache. Since fetching the entire cache block for bypassed data with little spatial locality will cause cache pollution and extra traffic, they used small lines (equal to the element size) for the buffers and optionally fill in consecutive blocks when spatial locality is detected. As we can see they are using a single buffer to serve the purpose of both a victim cache (for scalar data) and a prefetch buffer (for stream data). In a later study [5] they extended their scheme by adding an extra structure SLDT (Spatial Locality Detection Table) and extra counter for each MAT entry to detect spatial locality so that the system can adapt to varying spatial locality by dynamically adjusting the amount of data fetched on a cache miss.

The first difference between their work and ours is that rather than using locality types they have used reuse behavior of data as a metric for data separation. Since the MAT keeps the reuse pattern for all data in the whole program, at some point during execution it is possible for an array element to have higher reuse count than scalar data. In that case the MAT scheme will bypass the data which may have otherwise had a few hits before being displaced from the cache. Hence more than one additional miss will be incurred by not caching that data, whereas only one miss is removed by not displacing the more frequently accessed data. Secondly after identifying data as scalar/array we cache both types in separate caches whereas they used bypassing for data with less reuse history. The third and the most significant difference between their work and ours are the number and types of architectural constructs. We not only use two separate caches for two different data types, we also use two separate structures as victim cache and stream buffer to tune the amount of data cached and fetched. This allows us to fully exploit the functionality of a victim cache to reduce conflict misses of scalar data by holding data longer and the usability of stream buffer to reduce the cold miss of array data by prefetching. In their method using 8 lines of buffer as both victim cache and prefetch buffer will negatively affect each's performance.

4. Simulations

The cache architecture proposed in this paper has been evaluated for the following SPECfp2000 benchmarks, art, ammp, mesa, equake, fma3d, mgrid, applu and sixtrack [29]. The number of instructions executed by each application varied from 1 billion to 129 billions. We truncated some of the benchmarks to reduce the number of references. The descriptions of the benchmarks are given in Table 1. We used trace driven simulation as our evaluation methodology. The executables are instrumented using ATOM instrumentation and analysis system [30]. In an actual implementation of split caches, compile time analyses can be used to tag stream data so that they can be directed to array cache, separate from scalar cache.

Table 1: Descriptions of benchmarks used in the experiment

Benchmark name	Description	Name in figure
179.art	ImageRecognition/Neural networks	ar
188.ammp	Computational Chemistry	am
183.equake	SeismicWavePropagation Simulation	eq
177.mesa	3-D Graphics Library	me
172.mgrid	Multi-grid Solver: 3D Potential Field	mg
191.fma3d	Finite-element Crash Simulation	fm
200.sixtrack	NuclearPhysics Accelerator Design	sx
173.applu	ParabolicPartialDifferentialEquation	ap

In an attempt to evaluate the optimal configuration of the integrated approach, a variety cache sizes, block sizes, associativity and replacement methods were examined for each of array, scalar, victim cache and stream buffer. Table 2 presents the optimum configuration for the memory hierarchy that has been implemented in our study.

Table 2. Configuration of Memory hierarchy

Scalar cache configuration	4k, Directmapped,64bytes block
Access time of Scalar cache	1 cycle
Number of lines in victim cache	8 lines, non swapping
Victim cache associativity	Fully associative
Replacement Policy	LRU
Victim cache block size	64-bytes
Access time of Victim cache	1 cycle
Array cache configuration	4k, Directmapped, 64bytes block
Access time of Array cache	1 cycle
Number of stream buffer	4
Number of lines in stream buffer	10
Stream buffer block size	64 bytes
Access time of stream buffer	1 cycle
L2 cache configuration	256k, Directmap,64bytes block
Access time of L2 cache	10 cycle

5. Results

The next three subsections present the selection of cache organizations in the same order as these parameters were described in sections 2 and 3. We compare the effective miss rate of proposed cache against that of conventional unified cache. The results support our view that a complete separation of array and scalar data items with victim cache and stream buffer can be a key to boosting cache performance.

5.1. Results with Split Caches

In our previous work [1], we have shown that the combination of different block sizes and associativities together with partitioned cache architectures reduces compulsory and conflict misses in significant amounts and allows the combined cache capacity to be reduced. Figure 1 shows results of our previous work along with some additional benchmarks. In that work we simulated a partitioned 4k scalar cache while are streams mapped to a 2k array cache. This arrangement proved more efficient than a 16k unified data cache.

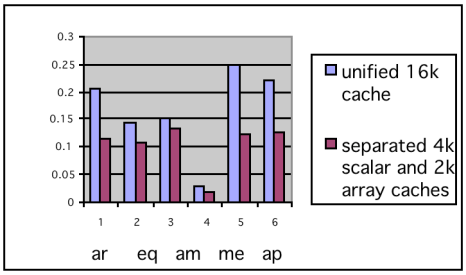

Fig. 1. Reduction in effective miss rate with array and scalar caches

5.2. Results with Victim Cache

Figures 2 and 3 compare the use of victim cache and higher associativity in decreasing cache misses and access time respectively. Figure 2 shows that using a victim cache with a direct mapped scalar cache led to miss rates similar to that of a 2-way set associative cache. Figure 3 shows that the victim cache allowed a significant reduction in access times for scalar data items. Given that access time is a better metric of cache performance than miss rate, our experiments show the significant benefit available with a victim cache.

5.3. Results with Stream Buffer

To evaluate the benefit of stream buffers with the array

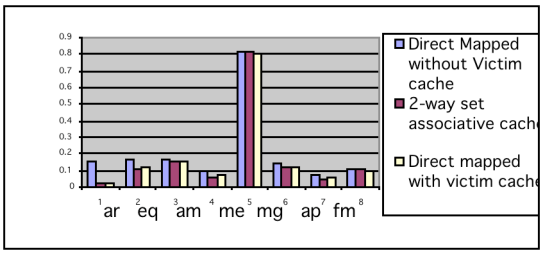

Fig. 2. Comparison of miss rate of 2-way set associative scalar cache with direct mapped scalar cache and victim cache

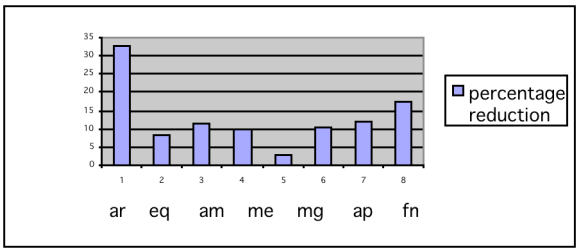

Fig. 3. Percentage reduction in access time by switching from 2-way set associative to direct mapped scalar cache with victim cache

cache, we used multiple (4) stream buffers of 10 elements, following Jouppi [6]. The cache miss rates and access time for each benchmark are plotted in figure 4 and 5 respectively.

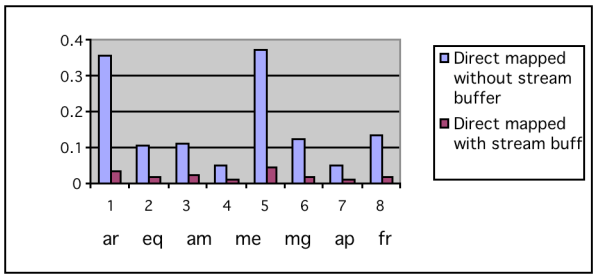

Fig. 4. Reduction in miss rate of array cache with stream buffer

5.4. Results of Combining Victim Cache, Stream Buffer and Split Caches

After the evaluation of optimal configurations for both victim cache and stream buffer, weighted effective miss rate for array and scalar caches are compared against the miss rate of unified 16k data cache. In order to find the effective miss rate we have used the following formula,

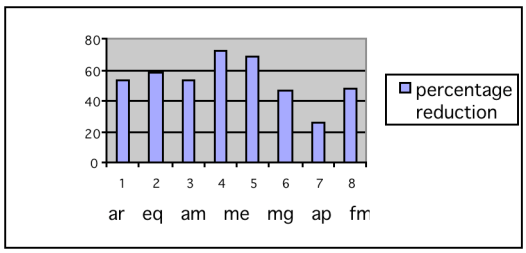

Fig. 5. Percentage reduction in access time by addition of stream buffer

*Effective miss rate = Array miss rate * (Number of Array references/Number of total references) + Scalar miss rate * (Number of Scalar references/Number of total references)*

The results are shown in figure 6. The integrated approach demonstrates uniform superiority over the conventional unified data cache design across all of the benchmarks. For 4k scalar cache and 4k array cache on average 55% improvement is achieved over a 16k unified scalar cache for the benchmark set.

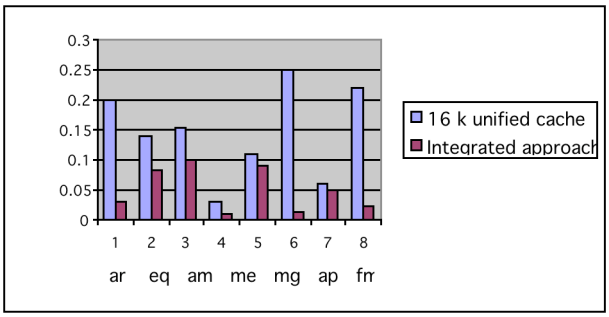

Fig. 6. Reduction in effective miss rate with integrated approach

6. Conclusions

The widening gap between processor and memory speeds makes data locality optimization a very important issue in modern cache systems. Computer architects focus on optimizing data cache locality using intelligent cache management mechanisms. In this paper, we investigated the interaction between three established methods, split cache, victim cache and stream buffer and proposed a strategy to optimize cache locality for scientific applications. Simulation results showed that proposed technique improved miss rates on average 55% with respect to the base configuration, even while using smaller combine cache foot-print. The proposed approach demonstrated how three inherently different approaches could be combined and made to work together by providing further achievement in data locality optimization arena.

7. References

[1] A. Naz, K.M. Kavi, P.H. Sweany and M. Rezaei. "A study of separate array and scalar caches" *Proceedings of the 18th International Symposium on High Performance Computing Systems and Applications (HPCS 2004)*, Winnipeg, Manitoba, Canada, May 16-19, 2004, pp 157-164

[2] A. J. Smith, Cache Memories, *ACM Computing Surveys* 14 (1982) 473-530.

[3] J. L. Baer and T. F. Chen, "An effective on –chip preloading scheme to reduce data access penalty. "In *Proceedings of the Supercomputing'91*, pp. 176-186, 1991

[4] T. L. Johnson and W. W. Hwu. "Run-time adaptive cache hierarchy management via reference analysis". In *Proc. the 24th International Symposium on Computer Architecture,*June 2–4, 1997.

[5] T. L. Johnson, M. C. Merten, and W. W. Hwu "Run-time spatial locality detection and optimization". In *Proc. the 30th International Symposium on Microarchitecture,*December 1–3, 1997.

[6] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers," In *proceedings of the 17th ISCA*, May 1990, pp. 364-373.

[7] A. Agarwal and S. D. Pudar. "Column–associative caches: a technique for reducing the miss rate of direct–mapped caches". In *Proc. 20th Annual International Symposiumon Computer*

[8] G. Kurpanek, K. Chan, J. Zheng, E. DeLano and W. Bryg, "PA7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface", *COMPCON Digest of Papers*, Feb 1994, pp. 375-382.

[9] C. Gonzalez, A. Aliagas, and M. Valero, "Data Cache with Multiple Caching Strategies Tuned to different Types of Locality," In *proceedings of International Conference on Supercomputing '95*, July 1995, pp. 338-347.

[10] V. Milutinovic, M. Tomasevic, B. Markovic, and M. Tremblay, "The Split Temporal/Spatial Cache: Initial Performance Analysis," *SCIzzL-5*, Mar. 1996.

[11] J. A. Rivers and E. S. Davidson, "Reducing Conflicts in Direct-Mapped Caches with a Temporality based Design, *Proc. 1996*

International Conference on Parallel Processing, August 1996.

[12] F. J. Sanchez, A. Gonzalez, and M. Valero, "Software Management of Selective and Dual Data Caches*", IEEE TCCA NEWSLETTERS*, March 97, pp. 3-10.

[13] M. Tomasko, S. Hadjiyiannis, and W. A. Najjar, "Experimental Evaluation of Array Caches", *IEEE TCCA Newslatters*, March 97, pp. 11-16.

[14] V. Milutinovic, M. Prvulovic, D. Marinov, Z. Dimitrijevic, "The Split Spatial/Non-Spatial Cache:A Performance and Complexity Evaluation", in *Newsletter of Technical Committee on Computer Architecture*, IEEE Computer Society, July 1999.

[15] J. H. Lee, J. S. Lee, and S. D. Kim, "A new cache architecture based on temporal and spatial locality," *Journal of Systems Architecture,* Vol. 46, pp. 1451-1467, Sep. 2000.

[16] J. H. Lee, G. H. Park, K. W. Lee, T. D. Han, and S. D. Kim, "A Power Efficient Cache Structure for Embedded Processors Based on the Dual Cache Structure," *In proceedings of the ACM LCTES'2000*, June 2000.

[17] *Intel StrongARM SA-1110 Microprocessor Brief Datasheet*, April 2000.

[18] O. S. Unsal, I. Koren, C. M. Krishna, C. A. Moritz, "The Minimax Cache: An Energy-Efficient Framework for Media Processors," *8th International Symposium on High-Performance Computer Architecture, HPCA8,* Cambridge, MA, February 2002, pp. 131-140.

[19] P. Ranganathan, S. V. Adve, and N. P. Jouppi, "Reconfigurable caches and their application to Media processing," *Proceedings of the 27th International symposium on Computer Architecture*, June 2000, pp. 214-224.

[20] G. Albera, R. I. Bahar, "Power/Performance Advantages of a Victim Buffer in High-Performance Processors", *IEEE Volta International Workshop on Low Power Design*, Como, Italy, March 1999.

[21] R. I. Bahar, D. Grunwald, B. Calder,"A Comparison of software code reordering and victim buffers",*ACM SIGARCH Computer Architecture News*, March 1999.

[22]R. Espasa and M. Valero. "A Victim Cache for Vector Registers". ICS-11. ACM "International Conference on *Supercomputing"*. Vienna, July 1997

[23]D.Hormdee, J.D. Garside, S.B. Furber, "An Asynchronous Victim Cache", *Proceedings of DSD'2002 Dortmund*, September 2002

[24] C. Zhang and F. Vahid, "Using a Victim Buffer in an Application-Specific Memory Hierarchy**",** *Design Automation and Test in Europe Conference (DATE),* February 2004, pp. 220-225.

[25] Dimitrios Stiliadis. "Selective victim caching : A method to improve the performance of direct-mapped caches", In *Proceedings of the 27th Hawaii International Conference on System Sciences*, Los Alamitos, California, 1994. IEEE, Computer Society Press.

[26] S. A. McKee, R. H. Klenke, K. L. Wright, W. A. Wulf, M. H. Salinas, J. H. Aylor, A. P. Barson, "Smarter Memory: Improving Bandwidth for Streamed References," in *IEEE Computer*. July 1998. p. 54-63.

[27]S.Palacharla and R. E Kessler. "Evaluating Stream Buffers as a Secondary Cache Replacement," *In Proceedings of the 21th International Symposium on Computer Architecture*, Chicago, IL, Apr. 1994, pp. 24--33.

[28] J. L. Hennessy and D. A. Patterson, Computer Architecture A Quantitative Approach, Morgan Kaufmann Publishers, Third Edition 2003, pp 423-430.

[29] L. Henning. "SPEC CPU2000: Measuring CPU Performance in the New Millennium", *IEEE Computer*, 33(7), pp. 28-35, July 2000.

[30] A. Eustance and A. Srivastava. "ATOM: A flexible interface for building high performance program analysis tools", Western Research Laboratory, TN-44, 1994.

Acknowledgement. This work is supported in part by a NSF grant ITR-0081214 (subcontract from Washington University in St. Louis).