A Study of Separate Array and Scalar Caches

Afrin Naz, Krishna Kavi, Philip Sweany and Mehran Rezaei

***Abstract*— In our prior work we explored the use of a separate cache for I-structure memories within the context of dataflow based multithreaded systems. I-structure memories in dataflow systems are used to store arrays and other indexed or stream data items. This 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 explore a similar cache organization providing architectural support for distinguishing between memory references that exhibit spatial and temporal locality and mapping them to separate caches. 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. According to the results of our simulations a partitioned 4k scalar cache with the streams (or arrays) mapped to a 2k array cache can be more efficient than a 16k unified data cache.

I. Introduction

Conventional caches imply no separation of data 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 minded treatment to the 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 whenever a cache miss occurs. 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 leads to wastage. Thus traditional treatment of cache misses not only causes unnecessary movement of data between the various levels of the memory hierarchy, it may lead to premature displacement of blocks that are likely to be re-referenced (i.e., cache pollution). This can become very costly if the newly loaded data tends to be non-temporal. In any case the result is an unnecessary increase in miss ratios, memory access times and memory bandwidths.

In our prior work we explored the use of a separate cache for I-structure memories within the context of dataflow based multithreaded systems [1]-[3]. I-structure memories in dataflow systems are used to store arrays and other indexed or stream data items. This 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 explore a similar cache organization providing architectural support for distinguishing between memory references that exhibit spatial and temporal locality and mapping them to separate caches. The selection of proper block size or associativity to maximize performance while staying within the cost are the hardest choices in designing cache memories. By partitioning the cache as we propose, our cache system can implement different configurations exploiting different cache parameters more selectively and effectively. The “array cache” is a direct mapped cache with larger block sizes to exploit spatial localities more aggressively by (pre)fetching multiple neighboring small blocks on a cache miss. Whereas the “scalar cache” is a 2-way (or 4-way) set associative cache with smaller block sizes to exploit temporal locality. The combination of different block sizes and associativities together with partitioned cache architectures provides an effective solution for alleviating the existing problems in cache designs and maximizes the effective cache memory space for any given cache size and cost. 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 thus be reduced. According to the results of our simulations a partitioned 4k scalar cache with streams (or arrays) mapped to a 2 k array cache can be more efficient than a 16 k unified data cache.

The rest of the paper is organized as follows. Section 2 discusses related issues and performance metrics in more detail to motivate the reader. Section 3 provides a survey and analysis of related research. Section 4 describes benchmarks and experimental set up used in our evaluation while, section 5 presents the results. The main conclusions are drawn in section 6 with a brief synopsis of the future work.

II. Cache, its data, parameters and deficiencies

Although caching dates back to Von Neumann's classic 1946 paper that laid the foundation for modern practical computing, it became vital in the performance of a processor since the beginning of 1990’s as the gap between the processor cycle and memory latency times increased dramatically. Caches are typically placed between a large, relatively slow and inexpensive source of information (the lower level of memory) and a much faster consumer of that information, the processor. The success of cache memories has been explained by the property of locality of reference [4], which is a property exhibited by most programs. A cache exploits this property to improve the effective access time to data and reducing the cost of accessing main memory. The property of locality has two aspects, temporal and spatial. 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 the time passes; spatial locality implies that when an instruction or datum is accessed it is very likely that nearby instructions or data will be accessed soon. Since cache buffers recently used segments of information, the property of locality implies that needed information is also likely to be found in the cache.

As the CPU speed has outstripped the rest of the system by many orders of magnitude and the memory bandwidth problem keeps growing, some deficiencies of conventional caching are becoming evident. Existing cache organization suffers from the inability to distinguish different types of localities and non-selectively cache all data rather than making any attempt to take special advantage of the locality type. This causes unnecessary movement of data among the levels of the memory hierarchy, significant interference between unrelated data inside the cache, removal of potentially useful data causing cache pollution and unnecessary increases in miss ratio, memory access time and memory bandwidth. The references can be easily divided into two groups according to the types of localities exhibited by the program -- the scalar and streamed (strided) references. Conventional cache techniques are acceptable for general-purpose scalar references with high temporal locality. But the picture is totally opposite for stream references, which do not reuse data soon or often enough to derive much benefit from caching. Since arrays and streams exhibit only spatial localities and the data sizes are often too large for caches, computations with streams and array access patterns cause mostly compulsory misses (rather than conflict misses) and perform extremely poorly in terms of cache behavior. In order to solve this problem, our proposed architecture groups the memory accesses as scalar or array references according to their inherent locality and each group subsequently mapped to a dedicated cache partition, equipped with architectural constructs built to exploit that particular locality type. In this system, since the scalar references and streamed references are no longer negatively affecting each other, cache interference, thrashing and pollution problems will be diminished, delivering better performance. Not only both caches would be designed more optimally according to their specific needs, it will simplify some other general issues and concerns in cache design, such as the associativity, cache block size or cache capacity. Even if the program displays only a small percentage of scalar references (in case of scientific applications) or very few arrays or streams we feel that it is better to use separate scalar and stream caches. In the following subsections we will talk about issues in general cache design.

A. Cache block size

The selection of block size depends on both the latency and bandwidth of the lower-level memory [5]. It is possible to achieve higher memory bandwidths on modern memories that are supported by wider buses, multiple banks, more pins, integrated circuit properties of DRAMs (for example on-chip memory), the newer Rambus and Synchronous DRAM [6]. However, high memory latency is still an issue that must be addressed. Although high latency and high bandwidth both encourage larger block sizes since the cache gets many more bytes per miss for a small increase in miss penalty, not all applications can benefit from larger blocks. Increasing block size to reduce the impact of memory latency also implies prefetching of data for applications exhibiting greater spatial localities, such as the applications using streams. On the other hand applications that exhibit very little spatial but greater temporal localities (as exhibited by scalar data items) cannot benefit from preteching or larger cache blocks. In fact, for scalar references, it is better to have smaller cache block sizes and more cache lines to eliminate conflict misses and even capacity misses when smaller caches are used [5]. Our work is motivated by the observation that it is not possible to design a single cache that works well for different types of localities and data types. We propose multiple data caches designed with different block sizes to meet the needs of the different data types.

B. Cache capacity

Increasing cache size will obviously reduce capacity misses; however as cache size increases, a capacity miss will become a conflict miss [5]. On the other hand if the number of capacity misses is small, increasing the capacity of the cache will not benefit the application. Jouppi et. al. [7] reported that increasing cache capacity actually increases cold-start or compulsory misses and these misses are more likely to be sequential in nature. This is particularly the case with stream data types. Thus for stream references it is not necessary to have a large cache. A larger cache can benefit applications that access several sets of data, but not for applications that access a single stream [7]. Similar results have been reported for media processing workloads [8].

C. Associativity

For a cache of given size, its set-associativity is dictated by a number of criteria, which include implementation cost, access time (both on hit and miss) and miss rate. 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. Because of lack of associativity, conflict misses typically account for 40% of all direct-mapped cache misses [7]. Conversely for caches with higher associativity the main advantage is lower miss rate, but they are more expensive and incur longer access times on hit. The goal of a computer architect is to maximize performance while staying within the cost and power constraints. A more desirable cache design would reduce the conflict miss rate to the same extent as a set associative cache, but at the same time it would maintain the critical hit access path of the direct mapped cache. Because of the lack of temporal locality, the stream references will cause more compulsory misses than conflict misses and direct mapping will be the better option for an array cache. Whereas for a scalar cache, increasing associativity will lead to a reduction of conflict misses and exploitation of temporal locality.

D. Streams

In this paper we focus on the class of computations that involve access to stream references. Streams are sequential, structured data or collection of successive elements with a known, fixed displacement (called stride) between elements. Streams are traversed linearly - during read these return successive elements, and during write these accept successive elements and store them sequentially. Typically the elements are operated on iteratively. And the elements are used once or very few times during the traversals. Thus streams exhibit high degree of spatial locality with very little temporal locality. Another characteristic is their transparent and completely predictable reference pattern. Stream accesses are typically generated in loops. At the beginning of the loop, only by knowing the base address (the address of the first element in the stream), stride and the length of the stream, the entire reference pattern of this specific type of structure can be predicted. Examples of computations using streams include vector (scientific) computations, string processing, multimedia applications, compression and decompression, encryption, signal processing, image processing, text searching, and DNA sequence matching.

Since caches that rely on temporal locality are ineffective for streams, memory bandwidth is rapidly becoming limiting for these streaming computations. The presence of higher spatial locality of the stream accesses makes them a better candidate for prefetching by increasing the block size. Since most modern DRAM components support modes that make it possible to perform some access sequences faster than others, the predictability of the stream accesses makes it possible to reorder them to get better memory performance [6].

III. RELATED WORKS

Complementing the cache with a small extra module to exploit temporal and spatial localities was first proposed by Jouppi [7]. The Stream buffer is a fully associative, FIFO buffer with 4 or 5 entries designed to support the 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 to avoid cache pollution(premature displacement of data). The Stream buffer will not only mitigate traditional problems with larger cache lines and extensive prefetching, it is more effective than other investigated prefetch techniques [9]. The biggest problem with Stream buffer is the it needs 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), the buffer was used for all data items. Subsequently, two different approaches have emerged; the first approach retains Jouppi’s original idea and supplements the regular cache with a small buffer for prefetching all data items regardless of the nature of locality exhibited by the data; the second approach is real cache partitioning to exploit different data localities exhibited by different data types. Partitioning of the cache can be either static or dynamic.

The most extensive and prominent work belonging to the first trend is done by Mckee et. al. [6]. They designed a SMC (stream memory controller), which is a combination of a small buffer and an intelligent scheduling unit for supporting the regular cache. When the program enters a loop that accesses one or more streams, compiler-generated code provides the scheduling unit with the base addresses, the number of elements, and the strides for any streams accessed in the loop body. The Memory Scheduling Unit (MSU) uses this information to reorder the requests so that even though the processor still issues requests to the Stream Buffer Unit (SBU) in the natural order, the order in which associated requests are made to memory will maximize the use of its bandwidth. Since the stream accesses no longer affects the cache, the cache can be designed more optimally for the remaining requests. Palacharla et. al. [10] proposed to use multiple stream buffers to replace the big secondary cache.

Sanchez et. al. [11] have proposed a dual data cache, which is composed of two modules, temporal module which is a fully associative buffer, built to exploit just temporal locality and spatial module, which is a direct mapped cache targeted to exploit spatial locality. The former module has only 16 very short blocks (each 64-bits) and the later has larger blocks (32 bytes per block). At the compile time memory instructions are tagged as bypass (data that do not exhibit any type of locality), spatial, or temporal. For misses both modules are checked in parallel to find the required data. References tagged as bypass are sent to CPU directly, rather than bringing them into cache. If a reference with a spatial or temporal tag misses in both modules, a new block is brought into the module indicated by the tag. Previously they proposed a similar architecture [12], where instead of compile time annotations, the memory references were tagged at execution time using an additional hardware unit called *locality prediction table*.

Tomasko et. al. [13] reported on a preliminary experimental evaluation of an architecture with separate array and scalar caches to observe the potential benefits to design a cache organization to a specific type of locality. In their experiment they assumed a model where the tagging of data as array or scalar to be allocated in one or the other caches would be done statically by compiler and the model does not assume any extensive analysis of references to determine the nature of the locality of access; rather it allocates the data only on the basis of the data type declaration. The main difference between their work and the one reported here is not using different configurations for exploiting associativity and implementing different sizes of array and scalar caches

STS (Split Temporal/Spatial) cache proposed by Milutinovic et. al. [14] is a little different than other proposed split caches. Since for "temporal" data hierarchy is needed (to reduce miss penalty for subsequent misses) but fetching the entire block is unnecessary, the temporal part is organized as a two level hierarchy with one word block size. The spatial part is one-level with four 32-bit words with a hardware implemented prefetching mechanism. The STS cache has four variants STS1, STS2, STS4a STS4b, each with same sized temporal module but larger spatial modules. Both modules are 4-way set associative with LRU replacement. Initially all data blocks are regarded as "spatial"; a data block may be changed to "temporal" and re-allocated through optimization of relevant parameters (using different counters to detect locality) during profiling or during runtime by means of a monitoring hardware unit.

In order to avoid the problem of determining counter thresholds present in STS method and also the problem of complicated memory hierarchies for each module, Milutinovic et. al. [15] proposed a simple method of detecting useful spatial locality which is tested by incorporating it into a new split cache design, called the Split Spatial/Non-Spatial cache (SS/NS). For detecting different types of locality the design used a flag based method, requiring fewer cache bits than counter implementation of STS. For exploiting the locality the system has two separate modules with same associativity and equal hierarchy. Prefetching is used if spatial locality is too large to be exploited by larger cache block.

The HP-7200 Assist Cache [16]-[17] design tries to avoid both cache conflict and cache pollution due to prefetching. The primary direct-mapped cache is coupled with a small fully-associative buffer (the Assist buffer), with a one cycle lookup in both units. The direct-mapped primary cache and the buffer units are designed with equal sized blocks. Until a block is identified as temporal, if it is requested either by a cache miss or a prefetch, the block is first loaded into the Assist buffer. It is promoted into the direct-mapped unit only when it exhibits temporal reuse. Spatial-only data, especially array data, may bypass the direct-mapped cache entirely, moving back to memory in FIFO fashion from the Assist buffer. In this system dynamic associativity is provided by allowing up to N+1 conflicting blocks which belong to the same direct-mapped set, to co-exist in the cache simultaneously, where N is the number of block entries in the Assist buffer. Since only a uni-directional communication exists between the direct-mapped unit and the Assist buffer; no swapping between the two units is allowed.

The NTS (Non-Temporal Streaming) Cache proposed by River et. al. [18] dynamically detects temporal (T) and non-temporal (NT) data and cache them separately. The NTS Cache consists of a data storing unit (DSU) which is a conventional direct mapped cache supplemented with a small fully-associative buffer (NT buffer) and a non-temporal detection unit (NTDU), which is a hardware bit-map structure attached to the main cache in order to monitor the reuse behavior of the blocks. Cache block size is uniform across the direct-mapped cache primary cache, the buffer units and the next level of memory. The strategies adopted in the NTS cache for detecting and caching temporal and non-temporal references are very similar to those implemented in HP-7200 Assist Cache [16]-[17]. Since the NTS cache did not handle the compulsory misses, Rivers et. al. [18] evaluated a group of caching strategies that integrate various combinations of temporal locality based caching and tagged prefetching.

Lee et. al. [19]-[20] have proposed a cache system called STAS (Selective Temporal and Aggressive Spatial) cache. Although they claimed to have two separate caches for temporal and spatial references with different block sizes, the module for spatial locality is actually a buffer rather than a cache. In this system on every memory access, both modules are accessed simultaneously. If a miss occurs at both places the block is brought to the spatial buffer. However a write back of dirty block in spatial buffer cannot occur directly - the dirty block is always placed in the direct mapped cache before being replaced. Later they have extended the STAS cache into another cache structure SMI (Selective Mode Intelligent) cache which consists of three parts: a direct mapped cache with a small block size, a fully associative spatial buffer with large block size and a hardware prefetching unit.

In the arena of media/embedded processors, static or dynamic cache partitionings are even more popular. Unsal et. al. [21] have proposed minimax cache which has a 8k 2-way set associative cache for non scalar data while the scalar data are directed to a 512 byte fully associative minicache. The system also has a secondary cache and the block size is the same across the buffer and caches. Intel's StrongARM SA-1110 [22], a low-power processor for embedded system, has a 8k data cache with 32-way set associativity and 512 byte fully associative mini data cache to enhance caching performance when dealing with temporal references. This system does not contain a L-2 cache.

Ranganathan et. al. [8], Petrov et. al. [23] and many others have proposed reconfigurable caches for embedded systems with dynamic cache partitioning. In their customizable partitioned cache, Petrov et. al. have used same block size and associativity for the partitions. The reconfigurable cache proposed by Ranganathan et. al. allows the cache array to be divided dynamically into two or more partitions that can be utilized by the processor for various purposes.

To contrast our approach with the designs summarized in his section, we propose a very simple design by providing two separate caches, named array and scalar, with individual design parameters optimized to meet the needs of different data types. The scalar cache will exploit temporal locality for some data items, while the array cache will be used to exploit spatial locality exhibited by other types of data items. Among the approaches mentioned above, SMC, Dual data cache, Assist cache, NTS, STAS, SMI are using multiple stream buffers to supplement the single data cache whereas STS, SS/NS, Array/Scalar, minimax cache, StrongARM are real split cache architectures. Our simple design principles of cache allows one to build correspondingly simple hardware controller. We believe that rather than using a multiple streamed FIFO buffer it is more practical to use a cheaper, faster, well-established architectural construct like cache. The performance of the split caches can be improved with compile time analysis and direct memory accesses to appropriate cache.

Our architecture permits the use of different block sizes and different associativities within a single CPU design. STS cache, SS/NS cache, Array/Scalar cache systems use the same associativity with different block sizes. Victim caches, Assist cache, the NTS cache system, the minimax cache, Intel StrongARM SA-1110 use different associativities but the same block size. None of these designs permit both different blocks sizes and different associativies in a truly split cache model. For stream references with spatial locality which causes more compulsory misses, we use direct mapped array cache with larger block sizes to benefit from prefetching. For scalar references which causes more conflict misses we use a 2-way set associative cache with smaller block sizes and more blocks in cache to avoid the high conflict and thrashing effect of direct mapped caches.

IV. Evaluation Methodology

In order to evaluate our ideas we developed a simulation environment. In this section, we briefly describe the characteristics of our benchmark codes and the simulation environment.

A. Benchmarks

The cache architecture proposed in this paper has been evaluated for the following SPECfp2000 benchmarks, art, ammp, mesa and equake [24]. Each of the program is written in C. We used gcc compiler version 2.3. The percentages of array and scalar references in the benchmarks are shown in Table 1. From the table it can be seen that the percentage of array references ranged from a low of 6.58% in mesa to a high of 26.92% in art. It should be mentioned that we used the exact benchmark codes and did not modify them to make efficient use of split cache (such as reordering array references or including prefetching hints). We traced complete program runs and did not limit only to array and in-loop references. Non-array references, especially scalar and stack variables contribute most to temporal reuse and are the main victims of cache pollution (premature replacement by array references). Hence excluding them from traces will not provide the true picture of program’s memory reference behavior. The number of instructions executed by each application varied from 1 billion to 129 billions.

TABLE I

Descriptions of benchmarks used in the experiment

Benchmark	Function	Total references	% of Array reference	% of Scalar referenc
179.art	Image Recognition/ Neural Networks	1244504516	26.92	73.08
183.equake	Seismic Wave Propagation Simulation	93304349658	18.63	81.37
188.ammp	Computation al Chemistry	2338560511	14.38	85.62
177.mesa	3-D Graphics Library	129910909077	6.58	93.42

B. Simulation Environment

We used trace driven simulation as our evaluation methodology. The executables of the benchmarks are instrumented using ATOM, a performance measurement tool [25]. ATOM instrumentation routine produces a new executable file a.out.atom. When this file is executed in the same manner and same input as the original program, a highly compressed trace file of every load and store reference made by the program is produced which is fed to the analysis routine of ATOM to simulate[1] different cache organizations for split array and scalar cache to generate miss rate and other relevant statistics for the program. 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.

In an attempt to evaluate the optimal configuration of the split cache, a variety of array and scalar cache sizes, block sizes and associativity were examined. We simulated three cache sizes for array cache (1k, 2k and 4k) and four cache sizes for scalar cache (4k, 8k, 16k, 32k and 64k) and block sizes ranging from 32 bytes to 128 bytes for both array and scalar caches. We have chosen two common approaches,

[1] We simulated caches directly inside ATOM analysis instead of collecting address traces and then using a separate cache simulator such as the Dinero V.

the direct mapped cache and 2-way set associative cache for both array and scalar cache. For 2-way set associativity we used both least recently used (LRU) and random replacement policies. We also simulated a conventional cache with corresponding configurations for comparison.

V. Results

of our experiments. By changing the block size, cache capacity and associativity, attempt is made to obtain the best configurations for array and scalar caches. The next three subsections present the selection of a cache parameter in the same order as these parameters were described in section 2. Finally we compare the effective miss rate of split-cache against that of conventional unified cache (for both stream and scalar data types), which support our view that a complete separation of array and scalar data items can be a key to boosting cache performance.

A. Selection of Block size

Approaches to exploit both types of locality in a unified data cache contradicts each other because in a fixed sized cache increasing block size will result in decreased number

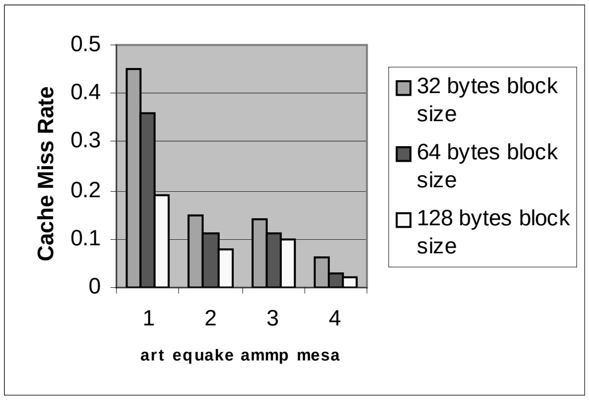

Fig. 1.Changes in miss rate with increase in block size of 4k array cache

of lines. Hence in a single data cache, it is not possible to achieve a balance between these two forms of accesses.

Figure 1 shows the decrease in miss rate with increasing block sizes in a 4k-array cache. Whereas for scalar cache as

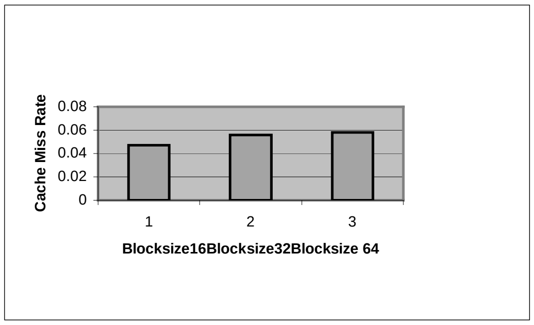

Fig. 2. Increase in cache miss rate with increase in block size of scalar cache for benchmark 177.mesa

shown in figure 2, for benchmark mesa, increasing block size actually causes an increase in miss rate. Similar results have been found for the other three benchmarks. Hence in our proposed architecture we will take advantage of both techniques-- by using larger cache blocks for array caches and smaller block sizes for scalar cache.

B. Selection of Cache size

As mentioned in section 2.2, an important criterion for selecting cache size is the frequency of capacity misses. We expect that when separate scalar and array caches are used,

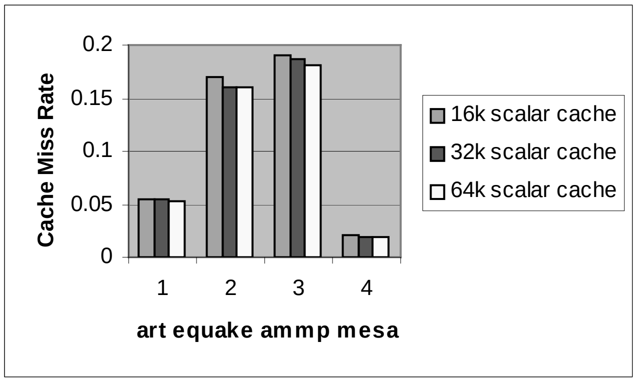

Fig. 3. Changes in cache miss rate with increase in cache size of scalar cache

the scalar cache can be very small (say 4K level 1) since the number of capacity misses is small with scalar data items. As figure 3 shows for scalar cache almost no improvement is achieved even after doubling or quadrupling the cache size. For this reason we decided to

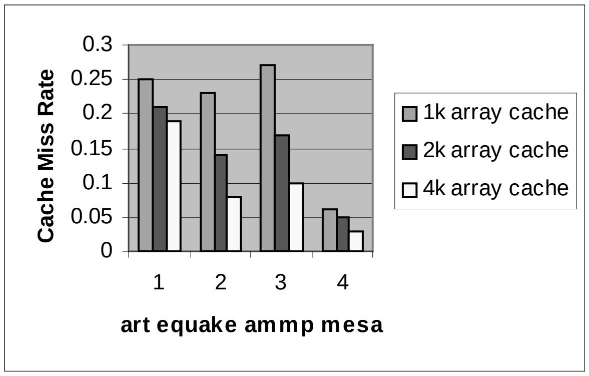

Fig. 4. Changes in cache miss rate with increase in cache size of array cache

use a small 4k or 8k scalar cache.

Figure 4 shows that for array cache increasing cache size with increasing block size reduces the miss rate. But we did not repeat our experiments with larger array caches than 4k because it has already been demonstrated that for stream references miss rate increases with cache size [7]-[8], unless even larger block sizes are used.

C. *Selection of associativity*

Several experiments are performed to determine the optimum associativity for each cache type and the cache miss rates for each benchmark are plotted. From figure 5

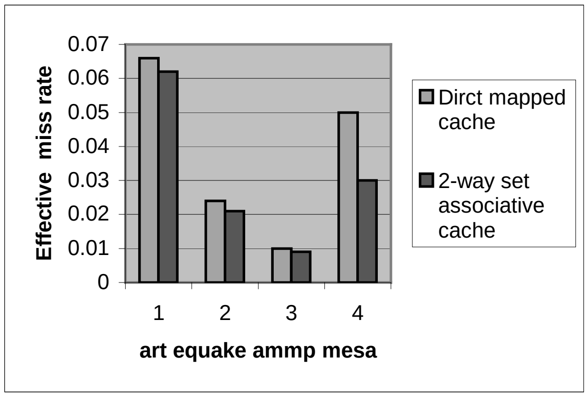

Fig. 5. Changes in cache miss rate for array cache with different associativity

we can see that for array cache, except benchmark mesa, increasing associativity will not be worth its cost.

This observation is consistent with our initial expectations. In our test suite, the percentage of capacity misses is very low and after removing streamed references,

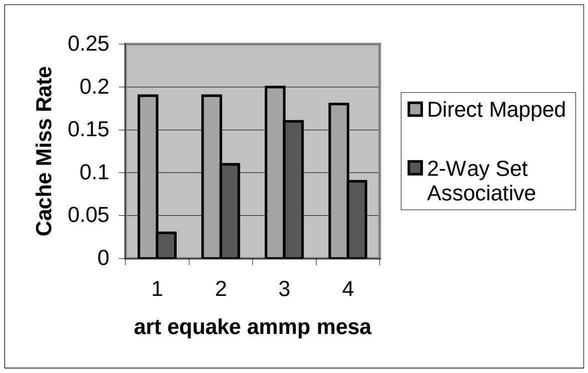

Fig. 6. Changes in cache miss rate for scalar cache with different associativity

for our scalar cache, conflict misses are the main concern. Whereas the lack of capacity misses first directed us to use a small scalar cache, this decision and the higher conflict miss rate then convinced us to select 2-way set associativity, for the scalar cache. It is obvious from figure 6 that for scalar cache increasing associativity improves the performance.

D. *Comparison of split array and scalar data cache with conventional unified data cache*

After the evaluation of optimal configurations for both array and scalar caches, weighted effective miss rate for array and scalar caches of all four benchmarks are compared against the miss rate of unified 16k data cache. In

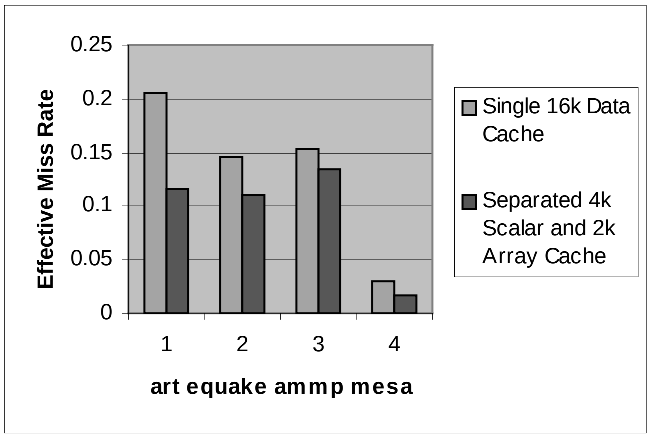

Fig. 7. Reduction in effective miss rate with separate array and scalar caches

order to find the effective miss rate we have used the following formula,

*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)*

Figure 7 shows the effectiveness of cache splitting across the benchmark suite. The split array and scalar cache demonstrate uniform superiority over the conventional unified data cache design across all of the benchmarks. For 4k 32 bytes scalar cache and 2k 128 bytes array cache 43.41%, 24.14%, 11.76% and 43.33% improvement is achieved over a 16k 64 bytes unified scalar cache for art, equake, ammp and mesa benchmarks respectively.

VI. CONCLUSION

In this paper we have presented the initial evaluation of a split array and scalar data cache. Existing cache memory architectures exploit locality of reference in both data and instruction address streams. Separation of cache is not a new idea. Modern processors rely on split cache architecture, at least on the first cache level, with separate instruction and data caches. The locality within the data address stream is also not uniform. Hence it seems worthwhile to exploit the two types of localities in data intensive applications, specifically in large-scale scientific computations. Several papers have been published on separate data caches. The main difference between this work and those is the complete independence of the two caches in terms of block size and associativity. We have simulated a direct mapped array cache with larger block size for stream references exhibiting spatial locality in order to permit prefetching and reduce the compulsory misses. Whereas our 2-way set associative scalar cache with more number of smaller blocks overcomes the structural drawbacks of direct mapped cache like high conflict and thrashing effects to hold blocks longer and exploit the reuse behavior of temporal locality. The achievements of these goals has been confirmed by extensive experimental results using SPEC 2000 benchmarks. Since this benefit may be very effective in increasing cache performance, which is expected to be an important limitation in future, we believe that our split cache architecture will find its way into the future microprocessors and the multiprocessor/multicomputer systems.

In a related research [26]-[30], we have been investigating techniques for off-loading memory management and other memory intensive operations to separate hardware logic either embedded within a DRAM or in the memory controller. When such intelligent memories are available, they can also be designed to utilize split data caches more effectively.

REFERENCES

[1] J. Arul, K.M. Kavi and S. Hanief, "Cache Performance of Scheduled Dataflow Architecture", Proc. of the 4th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP2000), Hong Kong, Dec. 11-14, 2000.

[2] K. M. Kavi, A.R. Hurson, P. Patadia, E. Abraham and P. Shanmugam. "Design of cache memories for multi-threaded dataflow architecture", Proceedings of the 22nd Intl. Symp. on Computer Architecture (ISCA-22), June 1995, St. Margherita Ligure, Italy, pp. 253-264

[3] K.M. Kavi and A.R. Hurson. "Performance of cache memories in dataflow architectures", *Euromicoro Journal on Systems Architecture*, Vol. 44, No. 9-10, June 1998, pp 657-674.

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

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

[6] S. A. McKee, R. H. Klenke, K. L. Wright, KL, 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] C. Gonzalez, A. Aliagas, and M. Mateo, “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.

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

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

[15] 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.

[16] 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.

[17] E. Rashid, A CMOS RISC CPU with On-Chip Parallel Cache, *ISSCC Digest of Papers*, Feb 1994, pp. 210-211.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] P. Petrov, A. Orailoglu, "Towards Effective Embedded Processors in Codesigns: Customizable Partitioned Caches", in *International Symposium on Hardware/Software Codesign (CODES)*, pp. 79-84, April, 2001.

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

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

[26] S.M. Donahue, M.P. Hampton, M. Deters, J.M. Nye, R.K. Cytron and K.M. Kavi. “Storage Allocation for real-time, embedded systems”, *Proceedings of the First International Workshop on Embedded Software (EMSOFT 2001)* (October 2001), Springer Verlag, pp 131-147

[27] S.M. Donahue, M.P. Hampton, R. Cytron, M. Franklin and K.M. Kavi. “Hardware support for fast and bounded time storage allocation”, *Proceedings of the Workshop on Memory Processor Interfaces (WMPI),* in conjunction with the International Symposium on Computer Architecture, May 2002, Anchorage, Alaska

[28] L.M. Fox, C.R. Hill, R.K. Cytron and K.M. Kavi. “Optimization of storage-referencing gestures”, *Proceedings of the Workshop on Compilers and Tools for Constrained Embedded Systems (CTES-2003*), held in conjunction with Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES-2003), Oct. 29, 2003, San Jose, CA.

[29] M.Rezaei and K.M. Kavi. “Utilization of Separate Caches to Eliminate Cache Pollution Caused By Memory Management Functions”, *Proceedings of the 16th International Conference on Parallel and Distributed Computing Systems* (PDCS-2003, sponsored by the International Society for Computers and their Applications, ISCA), Aug. 3-15, 2003, Reno, Nevada, USA.

[30] M. Rezaei. “Intelligent memory manager: Towards improving the locality behavior of allocation intensive applications”, PhD Dissertation, University of North Texas, May 2004.