Computer Science
Technical Report

Colorado State University

Evaluation of a Split Scalar/Array Cache Architecture

Michelle Tomasko, Simos Hadjiyiannis and Walid A. Najjar*
Department of Computer Science
Colorado State University
Ft. Collins, CO 80523
najjar@cs.colostate.edu

Technical Report CS-97-105

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

*This work was supported in part by DARPA Contract DABT63-95-0093

Evaluation of a Split Scalar/Array Cache Architecture

Michelle Tomasko, Simos Hadjiyiannis and Walid A. Najjar*
Department of Computer Science
Colorado State University
Ft. Collins, CO 80523
najjar@cs.colostate.edu

Abstract

The widening gap between the processor clock speed and the memory latency puts an added pressure on the performance of cache memories. This problem is amplified by the increase in instruction issue per cycle. This paper reports on the initial evaluation of a split scalar and array data cache. This scheme allows an efficient exploitation of both temporal and spatial locality by having a different organization and block size for each of the data caches. Initial experimental results show very significant improvements in hit rates on some Spec95fp and NAS benchmarks.

1 Introduction

As the gap between the CPU speed and the memory access latency keeps widening, as argued by Wulf and McKee [1], the performance of the first level cache architecture becomes even more critical to the overall performance of the processor.

The objective of existing cache memory architectures is to exploit the locality of reference in both the data and instruction address streams. Modern processors rely on a split cache architecture, at least on the first cache level (L1), with separate instruction and data caches. The locality within the data address stream is not uniform. Some accesses are more *spatially* local while others are *temporally* local. In particular, array accesses tend to be more spatial in nature and therefore benefit from larger cache lines (blocks) while scalar accesses are more temporal and benefit from smaller cache lines. For a fixed size cache, the locality in array accesses is best exploited by a small number of large cache lines while that in scalar accesses by a large number of small cache lines.

The performance of the cache memory is made even more critical by the increase in the instruction issue rate: the larger the number of instructions issued per cycle the higher the cost of a cache miss in potential instructions executed. Note that those programs that exhibit a large degree of instruction level locality, namely scientific type codes, are also the ones that operate on large arrays thereby exhibiting a large degree of spatial locality. The performance of such programs is very likely to be limited by the cache capacity.

This paper describes the evaluation of a split scalar and array cache. The selection between these two caches would be done statically at compile time by issuing different `load` and `store` instruction op-codes for scalars and arrays. A schematic of a possible implementation is shown in Figure 1.

This paper is organized as follows: Section 2 describes the experimental set-up used in the evaluation and Section 3 reports on the obtained results. An analytical evaluation of the cost and benefits of a split data cache is presented in Section 4. Related work is briefly discussed in Section 5.

2 Experimental Set-Up

The benchmarks used in this evaluation include eight of the SPEC95fp and four NAS benchmark codes.

*This work was supported in part by DARPA Contract DABT63-95-0093

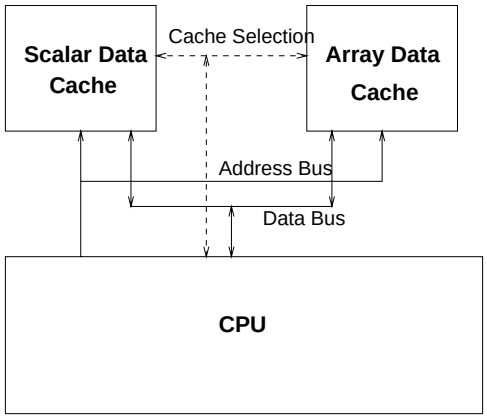

Figure 1: Split *scalar* and *array* L1 data caches

Benchmark	*Total refs.*	*% Scalar*	*% Array*
CFP95 (FORTRAN)			
APPLU	309818634	63.26	36.74
APSI	438022078	70.42	29.58
FPPPP	672945890	81.39	18.61
HYDRO2D	399672591	60.04	39.06
MGRID	243087010	72.88	27.12
SU2COR	209392603	46.27	53.73
SWIM	243023561	32.46	67.54
TOMCATV	203607050	62.69	37.31
NAS (FORTRAN)			
APPSP	347077931	69.09	30.91
BUK	51225081	88.84	11.16
CGM	594734959	48.54	51.46
EMBAR	665704522	96.62	3.37

Table 1: SPEC and NAS benchmarks used in the experimental evaluation

The executables of these benchmarks were processed by QPT2 [2, 3], a fast program profiling and tracing system. QPT2 instruments a binary executable file by inserting code to trace its execution, producing a new executable file `a.out.qpt`. When this file is executed, in the same manner and with the same input as the original program, a highly compressed trace file of every instruction and data reference made by the program is produced. QPT2 also produces a trace regeneration program that reads the compressed trace file and produces a full program trace.

In order to differentiate, during the cache simulation, a scalar from an array data memory reference, it is necessary to know the locations that the array data occupies during program execution, and their respective sizes.

In FORTRAN codes arrays are statically allocated: there is an explicit declaration of a global variable in the source code. In order to determine the location and size of such an array, a program was developed which examines the symbol and string tables of an executable which has been compiled with debugging information. This program scans the symbol table for all globally declared arrays, determines their run-time location, and by decoding the array data type also determines their size. Thus it is possible to determine for each data reference whether it is an array data reference or a scalar data reference by checking to see if it lies within the array memory ranges. The detection program also handles COMMON block arrays in FORTRAN.

The trace regeneration program source code is then compiled with a driver program (`din.c`) which reads the trace file and writes the address trace in a format suitable for the dineroIII cache simulator [4, 5] For a split cache it is necessary to distinguish between an array and a scalar reference in the address trace. The `din` driver program has been modified to check each data reference against the list of array memory ranges that it has available. If the data reference is within the memory space of an array, the reference is sent to the array cache, else it is sent to the scalar cache.

3 Experimental Evaluation

The percentages of array references in the benchmarks is shown in Table 1. These range from a low of 3.37% in EMBAR to a high of 67.54% in SWIM. The distribution is somewhat uniform in that the benchmarks are not biased one way or another.

The following experiments have four objectives. The first objective is to determine if miss rates can be reduced by splitting the cache and increasing the block size of the array. Secondly, if improvements are found, do they come from the split–cache configuration, or the increased block size alone? The third goal is to provide hints as to the optimal ratio of array cache size to scalar cache size. Finally, the optimal block and cache size configuration for the split data cache architecture is investigated.

3.1 Variable Array Cache Block Size

Here, the miss ratio of the split cache is compared against that of the unified cache to determine if any gains can be made. For this experiment, three unified cache size, 32KB–128KB, and five block sizes for the array cache, 64B–512B, were chosen. Both the unified cache and scalar cache have a constant block size of 32K throughout these experiments. The total effective size of the split data cache is equivalent to the size of the unified cache it is being compared to, with 25% of the size dedicated to the scalar cache and 75% to the array cache. The plots of the results are given in Figures 2 and 3.

The initial results for the split cache are quite encouraging. For eight out of the twelve benchmarks tested, improvement is found across all cache and cache block sizes. The exceptions are APSI, SU2COR,

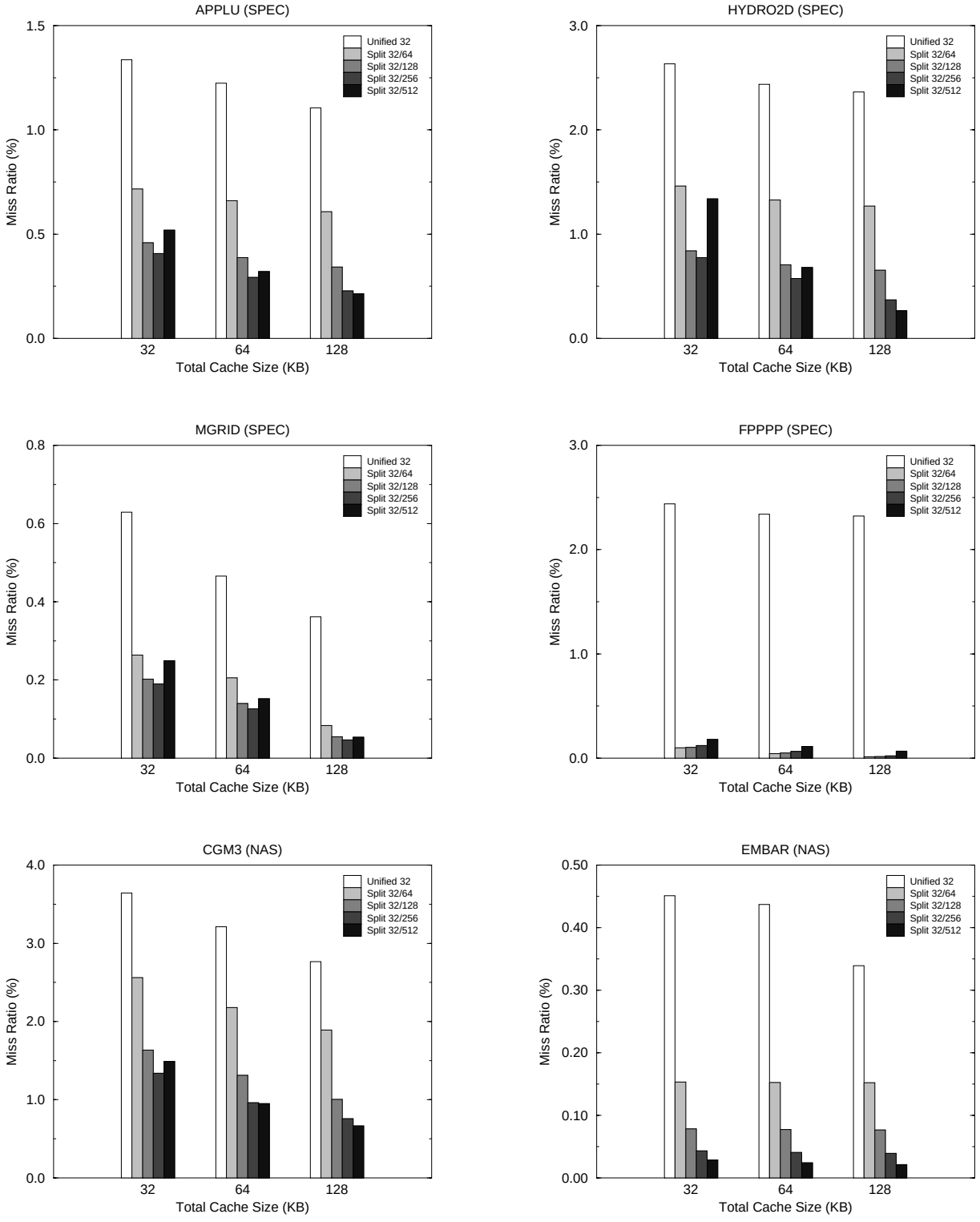

Figure 2: **25/75 Split Cache Performance.** The 25/75 scalar/array split cache is compared against the unified cache. The unified cache and the scalar cache of the split cache have a block size of 32B for all cache sizes. The block size of the array cache of the split cache varies from 64B–512B for each cache size. (*Continued, next page.*)

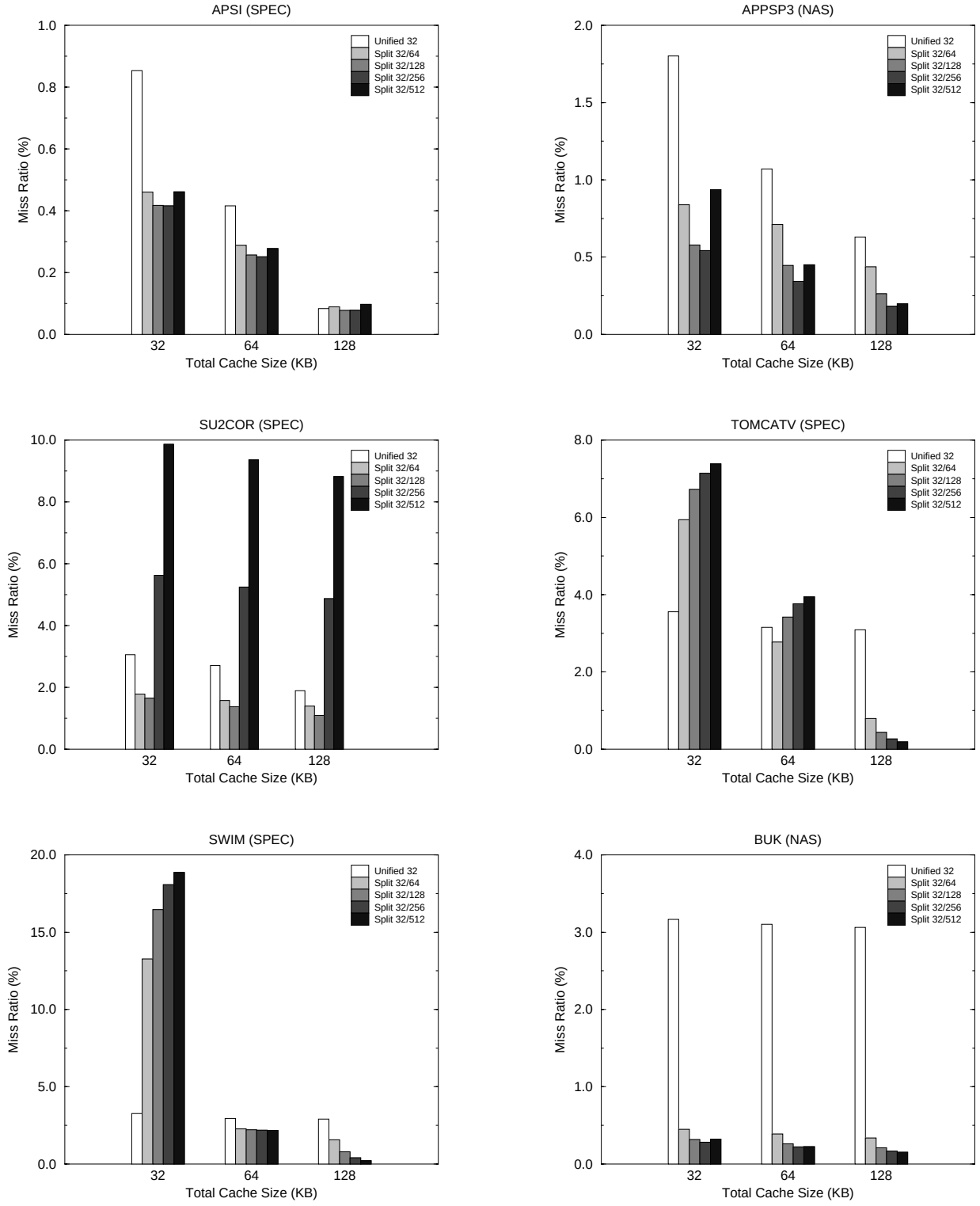

Figure 3: **25/75 Split Cache Performance.** The 25/75 scalar/array split cache is compared against the unified cache. The unified cache and the scalar cache of the split cache have a block size of 32B for all cache sizes. The block size of the array cache of the split cache varies from 64B–512B for each cache size. (*Continued from previous page.*)

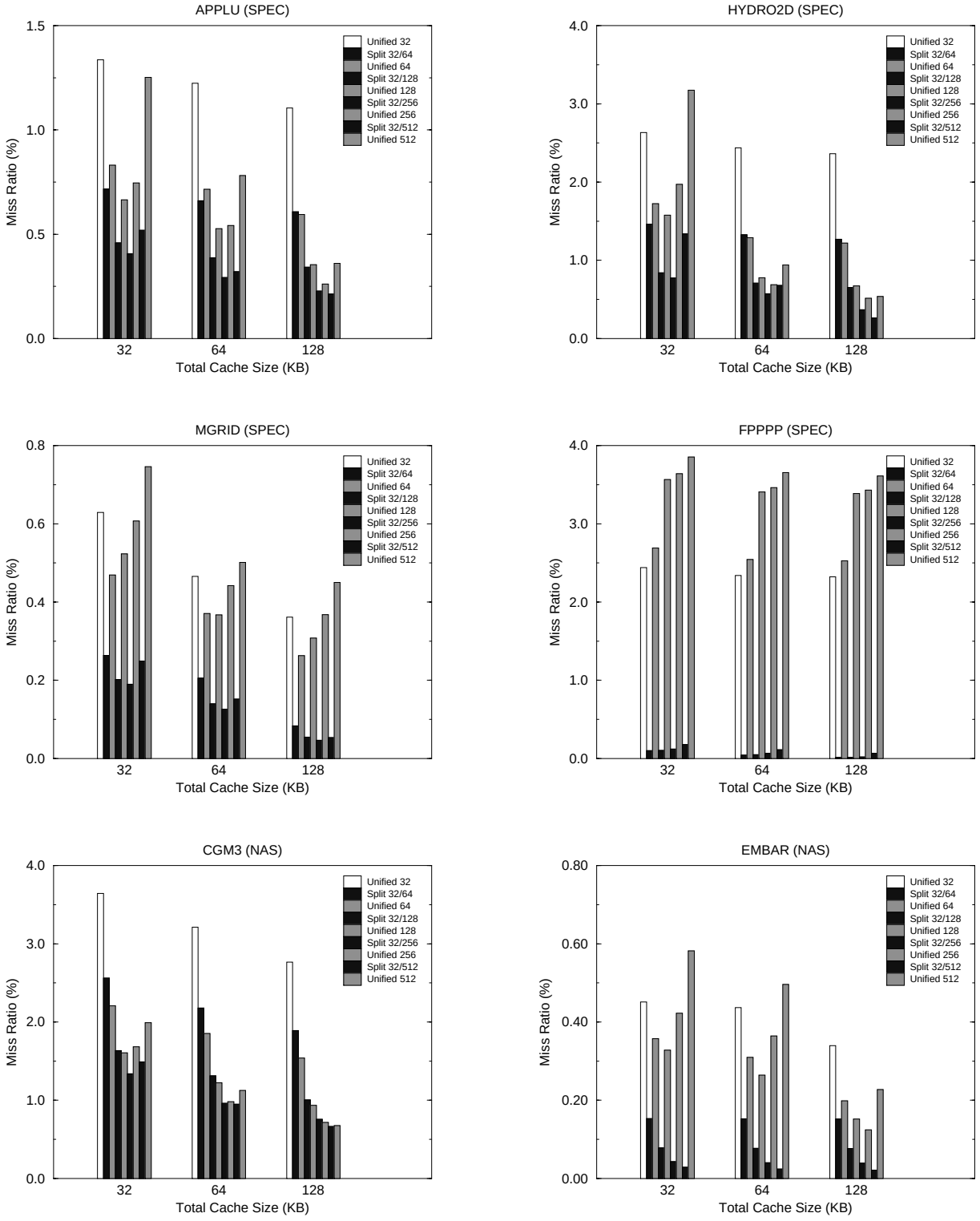

Figure 4: **Larger Block Sizes in the Unified Cache.** The 25/75 scalar/array split cache is compared to the unified cache with larger block sizes. The unifed cache miss ratio is plotted for block sizes 322B–512B for all cache sizes. The split cache miss ratio is plotted for constant scalar block size of 32B and array block sizes 64B–512B. (*Continued, next page.*)

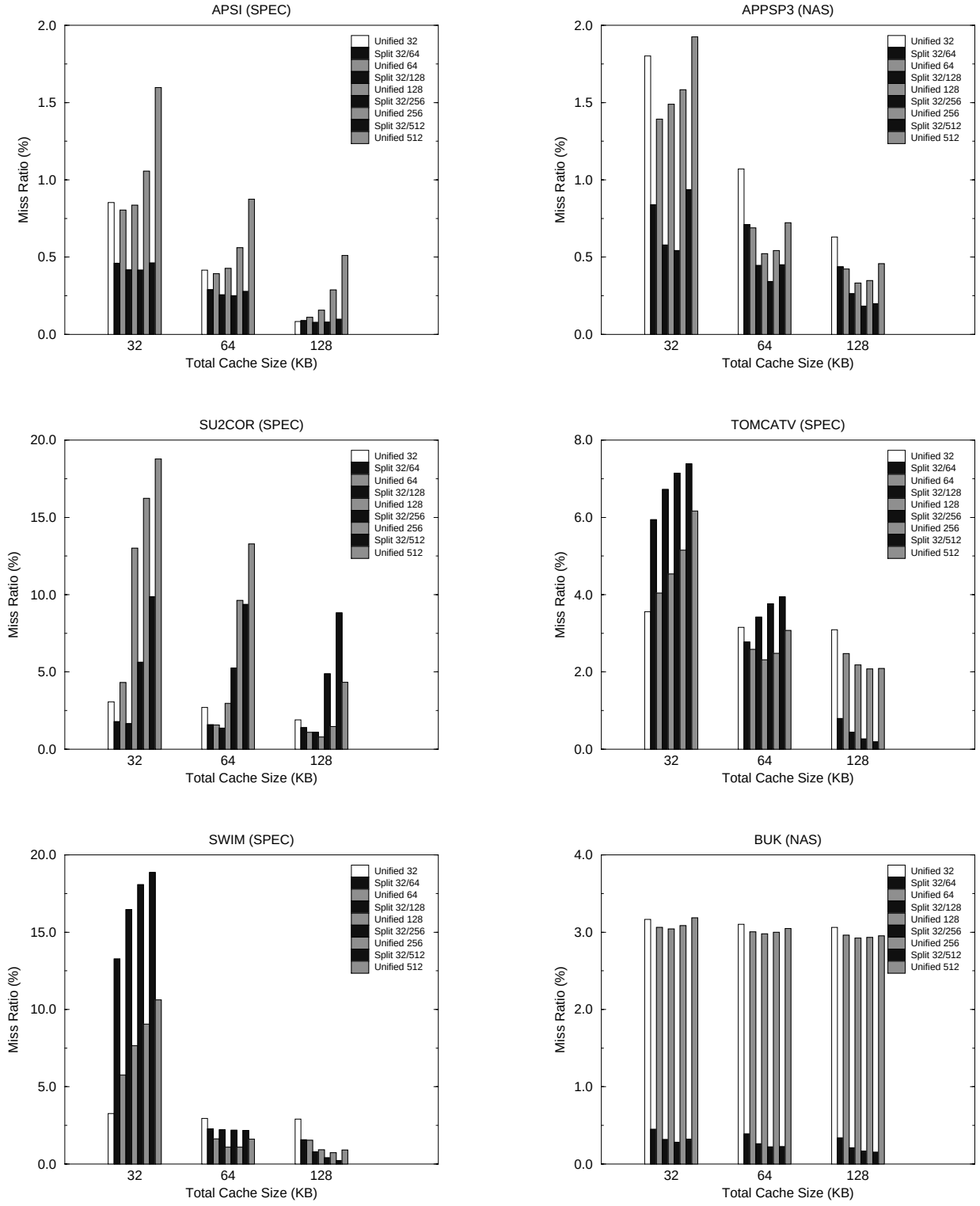

Figure 5: **Larger Block Sizes in the Unified Cache.** The 25/75 scalar/array split cache is compared to the unified cache with larger block sizes. The unifed cache miss ratio is plotted for block sizes 322B–512B for all cache sizes. The split cache miss ratio is plotted for constant scalar block size of 32B and array block sizes 64B–512B. (*Continued from previous page.*)

SWIM, and TOMCATV. APSI shows improvement for all cache sizes except 128K. For this cache size, the miss ratio of APSI is quite close to that of the unified cache. SU2COR improves miss rate at only the small block sizes and degrades in performance considerably for block sizes 256B and 512B. SWIM and TOMCATV, benefit the most at the larger cache sizes. Note that the plots given in Figures 2 and 3 are not on the same scale.

The degradation in preformance found in TOMCATV, SWIM, and SU2COR can be speculated to be due to one of two causes. One affect of splitting the cache can be the introduction of capacity misses in the array cache due to the smaller effective size. The other possible cause for increased miss ratio is in the nature of the array accesses. If the strides are very long, loading larger block sizes results in wasted effort and increased misses. The latter cause for degradation can be solved with a more intelligent algorithm for choosing which data should reside in the array cache and which should instead be sent to the scalar cache.

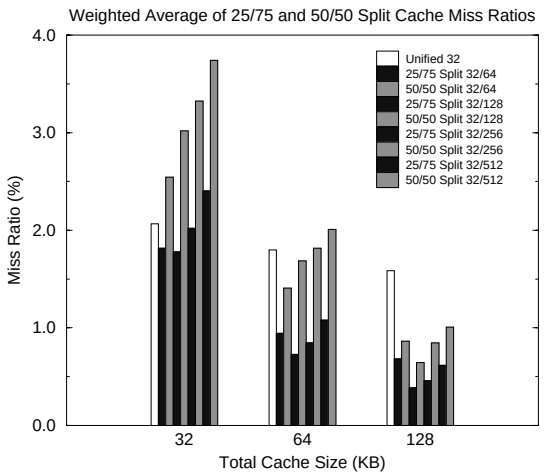

Figure 6: **25/75 Split versus 50/50 Split.** A 50/50 split of total cache size is compared the a 25/75 scalar/array split. For each cache size, the miss ratio of the 32B block unified cache is plotted. Both split cache configurations have a 32B scalar block size and an array cache block size varied between 64B and 512B.

3.2 Increasing Block Size in the Unified Cache

By increasing the block size of the unified cache, the gains made by the split data cache architecture is shown to be due to the combination of splitting the cache and increasing the data cache block size and not from only the increase in block size. Figures 4 and 5 show the results. The benchmarks that consistently benefit more from the split cache are APSI, BUK, EMBAR, FPPPP, and MGRID. APPLU, APPSP3, CGM3, and HYDRO2D have lower miss ratios for the split cache for a most of cache and block size configurations, but not for a cache block size of 64KB in the larger caches. TOMCATV and SWIM split caches only improve over the unified with larger block sizes for a cache size of 128KB. Conversely, SU2COR improves only at the 32KB and 64KB cache sizes.

3.3 Changing the Scalar/Array Cache Size Ratio

By changing the ratio of scalar cache size and array cache size in the split cache, attempt is made to provide a hint as to the best configuration. In this experiment, a 25/75 scalar/array cache split is compared against a 50/50 split. The results show that the 25/75 split outperforms the 50/50 split in nearly all cases. The weighted average plot summarizing the results is in Figure 6. Although these results fail to pinpoint the exact optimal scalar/array cache size ratio, they show that allocating a larger percentage of space to the array cache is better.

3.4 Optimal Block and Cache Size

To hint at the optimal configuration of the split data cache architecture, the weighted average of all the results of the benchmarks in this study is plotted. Figure 7 plots the average miss ratio of the unified cache for block sizes 32B–512B and split cache for array cache block sizes of 64K–512K. All tests were performed on total effective cache sizes 32KB–128KB. The split cache is a 25/75 split.

For all cache sizes, the optimal block size is 128B. The cache size that gives the best performance improvement is the 128KB cache. The 32KB, 64KB, and 128KB split caches improve the miss rate by 1.16, 2.47, and 4.12 times, respectively. Note that, on average, the split cache improves on the unified, 32B block size for all configurations except the 32KB cache with array block size of 512B. Furthermore, the average shows the split cache clearly performs better than the unified cache with larger block sizes.

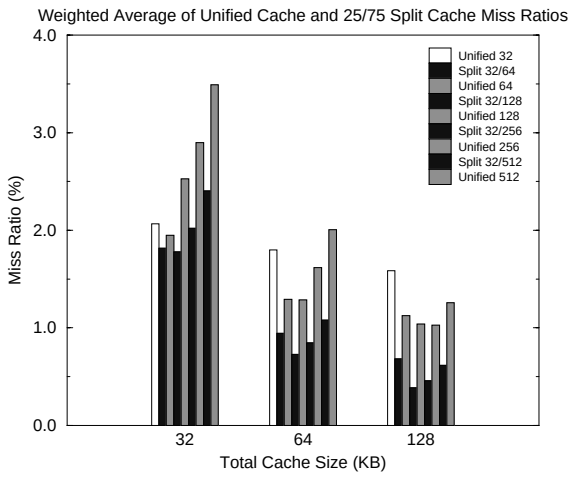

Figure 7: **Weighted Average of 25/75 Split Cache Performance.** The 25/75 scalar/array split cache is compared against the unified cache. The unified cache and the scalar cache of the split cache have a block size of 32B for all cache sizes. The block size of the array cache of the split cache varies from 64B–512B for each cache size.

4 Analytic Evaluation of Cost & Benefits

The implementation of separate array and scalar caches has implications on the whole processor architecture. Among these a possible increase in the clock cycle time which could defeat the reduction in overall miss rates. The purpose of this section is to analyze the cost trade-offs of a split data cache architecture based on the experimental data presented so far.

The expression for the CPU time including the cache performance [6] is:

$$\begin{aligned} CPUtime &= IC \times Clock\ Cycle\ time \times [CPI\ + \\ &\quad (\frac{Mem\ access}{Instruction} \times \\ &\quad Miss\ rate \times Miss\ penalty)] \end{aligned}$$

(where IC is the instruction count and CPI is the cycles per instruction). Since the clock cycle time and the miss rate are the two parameters of relevance here, expression can be summarized as:

$$CPUtime = C \times [CPI\ +\ P \times m]$$

where $C = Clock\ Cycle\ time \times IC$ and $P = \frac{Mem\ access}{Instruction} \times Miss\ penalty$.

Let C_u, C_s, m_u and m_s denote the parameters for the unified and split cache architectures. Let k denote the expected degradation in clock cycle time of the split caches ($C_s = (1\ +\ k)C_u$) and α the reduction in miss rates in the split architecture ($m_u = m_s\alpha$). Therefore the speed-up of the split caches over the unified one can be expressed as:

$$\begin{aligned} Speed-Up &= \frac{C_u}{C_s}\ \frac{CPI\ +\ Pm_u}{CPI\ +\ Pm_s} \\ &= \frac{1}{1\ +\ k}\ \frac{CPI\ +\ Pm_s\alpha}{CPI\ +\ Pm_s} \end{aligned}$$

This expression assumes that the CPI for both the split and unified architectures are the same. Depending on how the split architecture is actually implemented the CPI for the split cache design can actually be smaller since it would be expected that the CPU can potentially issue multiple simultaneous load instructions to the two caches. In this analysis no change in the CPI is assumed. In modern superscalar processors the CPI is often expressed as IPC (instructions per cycle) where $IPC = 1/CPI$. Using IPC instead of CPI the speed-up expression can be rewritten as:

$$Speed-Up = \frac{1}{1\ +\ k}\ \frac{1\ +\ IPC\ \times\ Pm_s\alpha}{1\ +\ IPC\ \times\ Pm_s}$$

The relation between k and α can be derived from the condition $Speed-Up \geq 1$ as:

$$\alpha \geq 1\ +\ k(1\ +\ \frac{1}{IPC\ \times\ P\ \times\ m_s})$$

Considering that the fraction of memory reference instructions is roughly one third, the expression can be rewritten as:

$$\alpha \geq 1\ +\ k(1\ +\ \frac{3}{IPC\ \times\ (Miss\ penalty)\ \times\ m_s})$$

In evaluating a numerical range of values for a *break-even* α the following ranges of parameters is considered:

- A 5% to 15% increase in clock cycle time because of the split cache architecture ($0.05 \leq k \leq 0.15$).
- A two to four issue superscalar architecture ($2 \leq IPC \leq 4$). Note that such rates are common today, future CPUs would most likely have much higher instruction issue rates.
- A miss penalty between 15 and 45 cycles. These values are relatively low by today's standards, as the clock cycle time of CPU shrinks in relation to the memory latency these values are expected to be even larger in the future.

- A unified cache miss rate ranging between 4% and 10%.

These parameters result in break-even values for α ranging from 1.06 to 1.525 which are within the ranges observed in the experimental results for those benchmark where the split cache architecture did improve performance. Note that in Figure 7 the miss rate for the 25/75 split cache with a 128 byte block in the array cache is 1.16, 2.47, and 4.12 times better miss ratio than the unified cache for the 32KB, 64KB, and 128KB cache sizes, respectively.

Note that this model assumes a wrapped memory fetch where the accessed word of the block, in a cache miss, is delivered to the processor first and the rest of the block is fetched to the cache in successive memory cycles.

5 Related Work

The problem of a split data cache along temporal and spatial components has been independently addressed in two prior works.

The concept of a dual data cache was first introduced by Gonzáles *et al.* in [7]. The dual data cache consists of a spatial cache designed to exploit mostly spatial locality and some temporal locality and a temporal cache designed exclusively for temporal locality. In this scheme the type of access is determined dynamically using a *locality prediction table* which is similar in nature to the schemes proposed in [8, 9, 10]. Furthermore, the temporal and spatial caches are not exclusive in that a data element could be present in both. The main difference between this work and the one reported here is the dynamic categorization of data accesses as temporal or spatial and the non-exclusivity of the two caches. The results reported in [7], however, are very similar to the ones reported here.

A split temporal/spatial cache architecture relying on a compile time tagging of the data is described and analyzed in [11, 12, 13]. This scheme includes a second level temporal cache as well as a mechanism for moving data between the temporal and spatial caches at run time. The performance of four variants of the split cache architecture are reported for the ATUM traces. The authors claim to achieve 40% better performance for the same complexity as a conventional cache. This approach is very similar to the one described in the present paper. It differs in the use of a temporal L2 cache and the dynamic moving of data between the two caches.

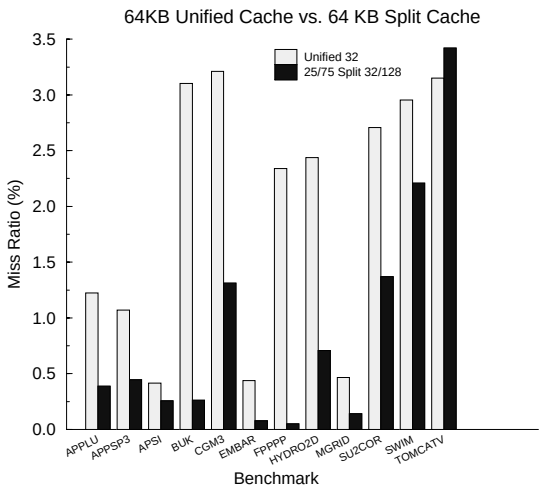

Figure 8: **Performance of the 25/75 Split Cache for all Benchmarks** This plot summerizes the miss ratio improvements for the 64KB cache using 32B blocks in the scalar cache and 128B blocks in the array cache. The unified cache uses 32B blocks.

6 Conclusion

This paper reports on the initial evaluation of a split scalar and array data cache. The tagging of data for allocation in one or the other data caches would be done statically by the compiler.

The initial performance evaluation is done using eight Spec CFP95 and four NAS programs. The split cache configuration evaluated is a 25/75 split of the data cache between scalar and array caches. The results indicate a significant improvement in the miss rates of eight of the twelve benchmark codes for all of the cache configurations evaluated. Of the others, one shows improvement for all cache sizes except 128KB, one only shows improvement using relatively small cache block sizes in the array cache, and two improve at the larger cache sizes. Figure 8 is a summary plot of the miss ratio improvements of the 25/75 split cache over the unifed cache for all the benchmarks using a cache size of 64KB and an array cache block size of 128KB. The block size for the unified and the scalar caches is 32B.

Comparing the split cache to the unified with larger block sizes shows that, on average, the split cache benefits the miss ratio more than merely increasing the block size of the unified cache.

In an attempt to determine the optimal configuration of the split cache, the different split ratios between the scalar and array caches were examined and a variety of cache and array cache block sizes

were tested. From the results, the 25/75 scalar/array split performed better than the 50/50 split, indicating that a larger array cache is better. For all the cache sizes, a 128B block size in the array cache was optimal. The cache size that saw the best performance improvement was the largest, 128K.

Finally,the cost benefit tradeoffs of the split cache organization is evaluated analytically. It is based on the assumption of an increase in the clock cycle time because of the selection between two first level data caches. The results show that the reduction in miss rates could easily offset any potential increase in clock cycle time. The split cache organization can therefore provide a significant reduction in overall execution time mostly for large scientific programs.

This paper reports on the preliminary results of the evaluation of the split data cache. Several other aspects of this cache architecture are currently being evaluated.

References

[1] Wm. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, December 1994.

[2] J. R. Larus. Abstract execution: A techniques for efficiently tracing programs. *Software Practice and Experience*, 20(12):1241–1258, December 1990.

[3] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Trans. on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[4] M. D. Hill. Test driving your next cache. *Magazine of Intelligent Personal Systems*, pages 84–92, August 1989.

[5] M. D. Hill and A. J. Smith. Experimental evaluation of on-chip microprocessor cache memories. In *Proc. Int. Symp. on Computer Architecture*, pages 158–174, Ann Arbor, MI, June 1984.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1995.

[7] A. Gonzales, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proc. Int. Conf. on Supercomputing*, pages 338–347, 1995.

[8] J-L. Baer and T-F Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proc. Supercomputing'91 Conf.*, pages 176–186, 1991.

[9] J. W. Fu, J. H. Patel, and B. L. Jansen. Stride directed prefetching in scalar processors. In *Proc. Int. Symp. on Microarchitecture (MICRO-25)*, pages 102–110, 1992.

[10] Y. Jegou and O. Temam. Speculative prefetching. In *Proc. Int. Conf. on Supercomputing*, pages 57–66, 1993.

[11] V. Milutinovic, M. Tomasevic, B. Markovic, and M. Tremblay. A new cache architecture concept: The split temporal/spatial cache. In *Proc. of the IEEE/MELECON-96*, 1996.

[12] V. Milutinovic, B. Markovic, M. Tomasevic, and M. Tremblay. The split temporal/spatial cache: Initial performance analysis. In *Proc. of the SCIzzL-5*, March 1996.

[13] V. Milutinovic, B. Markovic, M. Tomasevic, and M. Tremblay. The split temporal/spatial cache: A complexity analysis. In *Proc. of the SCIzzL-6*, September 1996.