SECTORED CACHE PERFORMANCE EVALUATION:
A case study on the KSR-1 data subcache

Jude A. Rivers and Edward S. Davidson
Advanced Computer Architecture Lab
EECS Dept., The University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109
jrivers@eecs.umich.edu

Abstract

Sectoring is a cache design and management technique that is re-emerging as cache sizes get larger and computer designers strive to exploit the possible gains from using large block (line) sizes due to spatial locality. Sectoring allows for a small tag array size to suffice retaining address tags only for the large blocks, but still avoids huge miss penalties by utilizing a smaller transfer size between the cache and the next higher level of memory. With this caching strategy comes the need for a new approach for evaluating cache performance, especially relating to cache space and its best use, bus traffic and so forth.

In this study, we give a broad overview of the technique of sectoring in caches. We have introduced a new set of metrics for cache performance evaluation, stressing cache block and bus traffic usage. We use these set of superfluity metrics to investigate the behavior of real scientific applications, and also to help determine adequate and appropriate cache design parameters. We show an example of how these metrics can help point to the spatial locality problems in a given application code, thereby indicating code optimization techniques which can most significantly improve the code's performance.

INTRODUCTION

There is no argument that the efforts of the past couple of decades focused on increasing processor speeds have begun to pay off. The result has been the emergence of very high-speed GFLOPS processors, which are very necessary for today's large scale scientific computing demands. However, these gains in processor speeds have not been well matched by mass memory speeds. Memory manufacturers have rather been on an upward swing in increasing the size of memories. The consequence of these trends has been an upward increase in the average number of processor cycles required to satisfy a memory access [16] . With the ever increasing gap between the main-memory access time and the frequency of instruction issue, the performance of a processing system has become highly dependent on the behavior of high-speed caches. Previous studies have shown how significant a cache is as a performance factor in avoiding expensive main-memory accesses in high-performance computer systems.

As SRAM and DRAM costs continue on a downward turn, we see an emerging trend in memory design within the computer architectural community. The most obvious is multi-level cache design, where the first (*primary*) level is normally kept small and direct-mapped, with subsequent levels being larger in both size and associativity. Also, cache sizes are getting larger (in the range of 256K and 2M bytes) for high-performance systems, and in order to reduce the implementation costs of large tag stores, most designers prefer to use large block sizes. However, Hill [1] and Goodman [3] have shown, separately, that for various applications, cache sizes and organizations, the minimum cache miss ratio is obtained with block sizes in the range of 16 to 64 bytes. They go on further to show that, for optimal memory traffic, this byte range is also the most acceptable for data transfer size. In addition, Eggers and Katz established in [4] that for a shared memory multiprocessor and many parallel applications, larger block sizes lead to more false sharing and coherence misses. Hence, in order to employ block sizes in this recommended range (16-64 bytes) while keeping a small tag array size, many designers have begun to revisit sectored caches. Sectoring is a cache management technique that has been in existence since the advent of caches [5] . A sectored cache design allows a cache block with a single address tag (the allocation unit in the cache) to be sub-divided into several sectors or subblocks. Each subblock (as we prefer to call them) has its own validity bit, and is the minimum unit of data transfer between the respective cache and the next higher level of memory.

Due to their significance in system performance, cache design issues have been studied extensively [6] [7] . Effects of associativity [1] , cache size or block size [8] have all

*This work used resources of the University of Michigan Center for Parallel Computing, partially funded by NSF grant CDA-92-14296.

been the focus of many studies. However, with some few exceptions [3] [9] , not much performance evaluation has been done with respect to sectored caches. A systematic focus of previous efforts in evaluating designs has been on effects of cache organization on miss rates and memory traffic ratios. These popular metrics are related to total execution time, but are insufficient for sectored cache design. Interestingly enough, balancing performance, cost, area and power still remain the goals and constraints within which the computer designer must operate. Therefore any design that leads to a lot of waste and unused space in the cache adversely affects system performance. Sectored cache design introduces a significant metric for cache performance evaluation that can no longer be overlooked as *on-chip* and *companion-chip* processor caches get larger. This new metric is about space, and how much of it is being wasted in caches. It is also about how much unnecessary memory traffic is being generated in the form of data items transported in and out of the cache but never used. We introduce the *superfluity coefficient*, and use this metric to measure inefficient cache utilization due to *superfluity* and *pollution*. The significance of this metric could be very obvious in coherent bus-based shared memory multiprocessor systems where very large cache blocks can lead to *false sharing* [10] (i.e. a situation wherein two data items that are not being shared happen to reside in the same cache block). False sharing increases the number of invalidations and, subsequently, coherence misses [4] , reducing the cache performance. Also, we believe this metric can be helpful in identifying the *spatial* and *temporal* data locality problems of large scientific application programs, thereby assisting in tuning or restructuring efforts for better cache performance.

In this study, we examine the performance impact of sectored cache design on high-performance computer systems. For our baseline system, we choose the *KSR1* [15] from Kendall Square Research. The KSR1 is a cache-based shared-memory multiprocessor system that belongs to the Cache-Only Memory Architecture (COMA) family. The memory associated with each processor is managed as a 2-level sectored cache. The primary cache per processor (called the *subcache*) is split into separate instruction and data caches, with each consisting of 128 blocks of thirty-two 64-byte subblocks. The subcache is organized as 64 sets, with 2-way associativity and a random replacement write-back policy. The secondary cache (called the *local cache*) is a 32 MB cache, organized as a 128 set, 16-way set associative cache with a least recently used (LRU) replacement policy. For this work, we limit our study to one single processor and its primary cache and treat the secondary cache as main memory. A pertinent question we seek to answer also is: How good is this *all-cache* memory design on a real scientific application?

The rest of this paper is organized as follows. In the next section, we introduce the idea and principles behind sectored caches. We show the relevance of this cache design with respect to maintaining a low tag implementation cost while providing large cache blocks for better exploitation of spatial locality in scientific application programs. In section 3, we discuss the methodology behind this work and the trace gathering methods used. The KSR1 subcache structure and its principles of operation are presented. We also discuss our simulation approach, the application programs and the performance metrics used in this study. Section 4 presents trace-driven simulation results across various cache sizes with varying subblock sizes. Meaningful comparisons and tradeoffs are shown concerning the various metrics. In section 5 we introduce code optimization ideas; and show reductions in superfluity, miss rate, memory traffic and pollution for an optimized version of one of our test applications. Section 6 presents conclusions to this work.

SECTORED CACHES

Sectoring is a cache management technique that was developed to help ease two major problems in caches [22]:

- the tag storage can create a significant space overhead for a cache design, and
- the transfer of large cache blocks results in long *miss penalties*.

Tag Storage

Reducing tag implementation cost in a cache is an important design issue that requires much attention, even as memory gets cheaper and caches get larger. For traditional caches, every cache block has a tag word associated with it. A tag word normally comprises of an address tag and other status bits. An address tag makes it possible to retrieve the effective address (which can be either virtual or physical) of the data stored in the cache block. Virtual and physical addresses are getting wider (approaching 64 bits for the former and 36 bits for the latter [11] [12]) and this has resulted in the tag word occupying a significant fraction of the width of the cache block itself. For example, the MIPS R4000 has a cache block size of 16 bytes maintained by a 3 byte (i.e. 24 bits) address tag, not counting the other status bits. This is an indication of how many tag bits are needed to maintain a cache block. Besides this space overhead problem, keeping the tag bits to a minimum is an important performance issue for most microprocessor systems since the tag array may need to service two accesses per cycle: a snooping transaction on the bus for maintaining cache coherence and a transaction from the processor for accessing the cache [9] . This calls for the tag array to be double-ported while the data array may remain single ported. Thus, for off-chip caches, the tag array cannot be easily built with the same RAM chips as the data array. In most designs today [13] [14] [15] , an attractive solution has been to include the tag array and the whole control logic for the cache in a single companion-chip. Hence the upper bound on size for the tag array is limited by that chip's integration density.

Large Block Sizes

The choice of a cache block size (the amount of data stored in a single cache block) still remains a debatable issue in cache design. Since there is no single, optimum block size for all machine and cache designs, the specific design goals for the machine pretty much determines the block size used. A cache improves system performance by exploiting the two types of locality of reference: *temporal* and *spatial*. Temporal locality is the property that programs are likely to reuse recently referenced items. Spatial locality, on the other hand, suggests that programs are likely to reference items that are *near* recently referenced items. This would lead us to believe that large cache blocks improve application performance when there is substantial spatial locality. This might not, however, be the case for some parallel applications and coherent cache-based multiprocessing systems as described below.

Several studies have shown that the performance of coherent caches depends on the relationship between the granularity of sharing and locality exhibited by a program and the cache block size [4] [17] . If cache blocks are smaller than the data objects used on a per-processor basis, then accessing a single object can result in many cache block references, leading to more misses in order to acquire the needed data objects and possibly more conflict misses. Conversely, if cache blocks are too large, then there is the likelihood of false sharing, which also increases the number of invalidations.

Reducing the implementation cost of tags is a major reason for designers to choose a large block size; but research findings have generally favored small to medium cache blocks:

- Hill [1] has shown that, for a large number of applications and for many cache sizes and organizations, the minimum cache miss ratio is obtained for line sizes in the 16-64 byte range.
- Goodman showed in [3] that as the minimum data transfer size (cache block, in many traditional caches) between memory and the processors increases, traffic between memory and the processors increases even with miss ratios remaining constant
- Another evidence of this is the findings by Eggers and Katz [4] : for many parallel applications in a coherent shared memory environment, increasing the block size does increase the amount of false sharing and consequently, coherence communication.

To exploit the gains of large block sizes, while only paying for a small tag array size, and at the same time take advantage of small to medium block size for fast data transfer, revisiting sectoring might be an excellent option for cache designers.

Principles of Sectoring

In a sectored cache, each cache block contains several subblocks with each subblock having its own validity and coherency bits. The important property here is that all the subblocks in a block share a single address tag. Any two or more subblocks that are valid in a particular block in the cache belong to the same block in memory (i.e. their addresses differ only by the block offset).

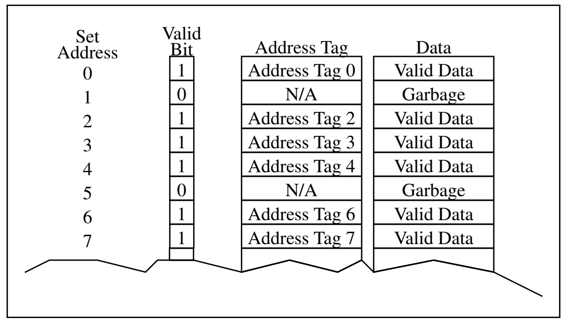

FIGURE 1. Tag and data organization of a non-sectored cache. Block size = 1 word

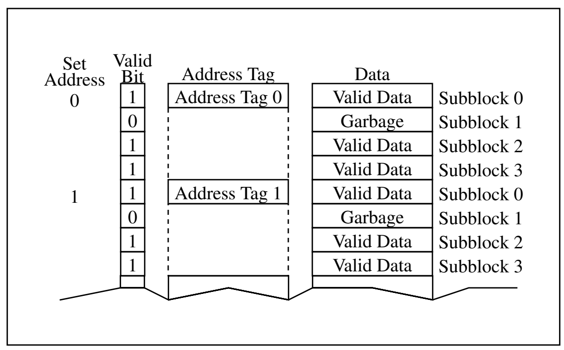

FIGURE 2. Tag and data organization for a sectored cache. Block size = 4 words, subblock size = 1 word.

The size of the tag array in a sectored cache is significantly smaller than the size of the tag array in a non-sectored cache using the same transfer unit size. We illustrate this with the figures above. Figure 1 shows the tag and data organization of a non-sectored cache where the block size is one word long. The same cache is shown in Figure 2 organized as a sectored cache with a block size of 4 words and a subblock size of one word. From Figure 2, we see that the cache data and the valid bits for every word (i.e. the subblock size in this case) have been kept, but the address tag store has only one-fourth as many entries. This obvious savings in the tag array size explains why the sectored organization is being reconsidered to save silicon in integrated cache controllers, whose address tag storage is on the same die as the cache controller. An example is Intel's 82385 controller for the 386 microprocessor.

Cache Misses : The transfer granularity from memory to the processor on a sectored cache is a subblock. Goodman [3] showed that this low granularity of data transfer reduces bus traffic. Two kinds of misses are distinguishable in the sectored cache environment:

- *Block misses*: the subblock that contains the referenced word is missing and no subblock in the same memory block is alive in the cache (i.e. the referenced block is not allocated). This is the same as the actual misses that would have occurred if sectoring was not implemented, with block size remaining the same.
- *Subblock misses:* the subblock that contains the referenced word is missing, but some other subblock(s) of the same memory block is(are) alive in the cache. That is to say, the referenced block is allocated in the cache but the referenced subblock in it is invalid.

When a cache miss occurs, the block that must be replaced is invalidated (if the miss was as a result of a block miss and there is a block to replace); a new block is allocated if necessary; and the subblock of the requested word is loaded into the cache. The valid bit of this subblock is then set; and those of the other subblocks in the block are reset, for a newly allocated block. Good spatial locality would lead us to believe that words in nearby subblocks may soon be requested by the CPU. When any of those subblocks is indeed referenced, the cache controller loads the subblock into the cache and sets the valid bit for that subblock within the block. Only the subblocks that contain words that the CPU actually requested would be brought into the cache from main memory, a positive measure in controlling the memory-to-cache bus traffic.

For systems with bus architectures capable of burst read cycles, an ideal subblock size need to be more than a single word in order to take advantage of spatial locality. However in a coherent shared memory multiprocessor, the bus may easily become a performance bottleneck if the system gets caught in the *ping-pong* phenomena (a thrashing problem among private coherent caches due to false sharing). The good news is that, using small cache blocks can help alleviate this performance problem [4] , and this can well be achieved by using sectoring.

Examples of Real Sectored Caches: The following are a few examples of sectored caches that are commercially available today:

- The KSR1 [15] has 2-levels of sectored caches associated with each processor. The *subcache* (as the primary level is called) consists of 128 blocks; each block has 32 subblocks of size 64 bytes. The *local cache* (i.e. the secondary cache) comprises of pages, each of size 16 Kbytes; each page contains one hundred twenty eight 128 byte subpages.
- The Motorola MC68030 primary cache has 4K blocks of 16 bytes each. There are four 4 byte subblocks in each block.
- The TI SuperSparc instruction cache has 64 byte blocks each with two 32 byte subblocks.
- The Power601 on-chip unified cache also has 64 byte blocks, each with two 32 byte subblocks.

METHODOLOGY

We use trace-driven simulations to study the trade-offs in the performance of different sectored cache configurations. In this section, we give a brief description of our simulation environment and the specific architecture on which it is based, the trace generation process, the characteristics of the applications used, and the metrics used for performance evaluation.

The KSR1 Processor Cache

Each KSR1 node contains a 64-bit custom processor with a 20 MHz clock. Even though the basic architecture is a load/store RISC, an added enhancement allows a 2-instruction issue per clock cycle: one address calculation, branch, or memory instruction and one integer or floating-point calculation instruction. The processor is connected to memory by two 64-bit wide buses, one for instructions and the other for data. The data bus has the capability of providing an 8-byte word to the processor per clock cycle.

A KSR1 node has two levels of private cache. The first level consists of a 0.25MB data subcache and a 0.25MB instruction subcache. The subcache is a companion-chip cache; it is 2-way set associative and uses random replacement, write back policy. Each processor has four cache control units (CCU) that manage the subcache and local cache. Space allocation in the subcache is done by the CCU on a block basis. However the unit of data transfer from the second level cache to the subcache is a subblock. A block is 2048 bytes long and contains 32 subblocks, each of which is 64 bytes.

The second level cache, called local cache, is 32MB in size, organized as 128-sets, 16-way set associative, with LRU replacement. Our present study considers only the first level data subcache, and views the local cache as main memory. It is our assumption that the instruction subcache in every configuration we consider is big enough for all instruction references to hit in the cache. Figure 3 shows the KSR1 memory structure.

Windheiser [19] et al. conducted a series of timing experiments to determine memory access latencies for various levels of the KSR1 memory hierarchy. We provide these estimates in Table 1 for the subcache and local cache.

Simulation Environment

Our trace-driven simulation environment is modeled around two tools: K-Trace and *dineroSf* for generating and inter-

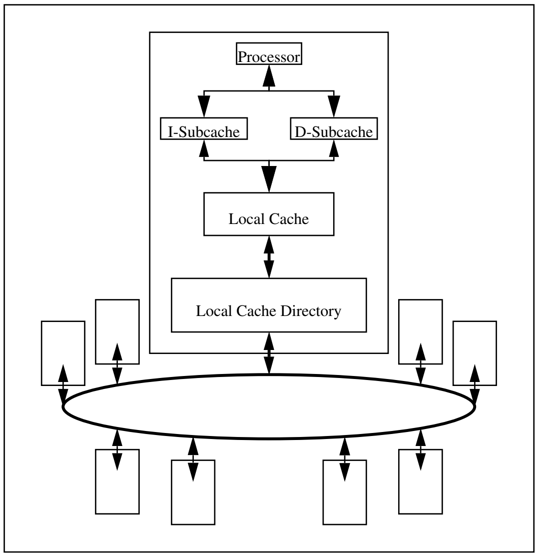

FIGURE 3. KSR1 Processor and Cache Structure (for one processor)

TABLE 1. KSR1 cache typical access times.

Memory Component	Memory Size (MBytes)	Memory Access (Cycles)
Each Subcache	0.25	2 (1 per clock)
Local Cache	32.0	
(Allocated block)		23.4
(Unallocated block)		49.2

preting memory references respectively. The overall simulation environment is as shown in Figure 4. K-Trace[a], a trace generator for memory references on the KSR1, was used to generate address traces for the application programs.

The performance simulator, *dineroSf*, is a modified version of *dineroIII,* originally developed by Mark Hill [2] . *dineroIII* is a trace-driven sectored cache simulator. Simulation results are determined by the input trace and the set of given cache parameters. It uses the priority stack method of

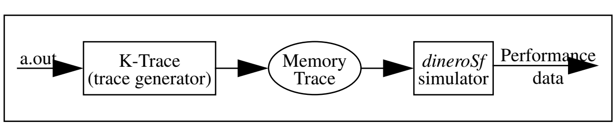

FIGURE 4. The simulation environment (tracing and simulating program *a.out*).

[a] K-Trace was developed by Shih-Hao Hung of the PPP project at the University of Michigan.

memory hierarchy to increase flexibility and improve simulator performance in highly associative caches. The modifications that we added to produce *dineroSf* incorporate additional cache parameters and provide the added capability of measuring superfluity and pollution in the cache. In addition to the capability of marking a subblock valid or invalid, reference bits are associated with individual words within a subblock that can be set to mark word usage. For each memory reference, the tag store gets checked first. On a hit (a reference to a word in a valid subblock) if the reference bit for the word has not been previously set, it is done at this time. We deal with two types of misses as explained earlier. Missing on an absent block forces a new block to be allocated for the reference, and this calls for invalidating all the valid subblocks of the block being replaced. All valid subblocks of this evicted block that are marked dirty need to be written-back to memory. We also take statistics for the number of subblocks that were marked valid during the block's lifecycle in the cache, enabling us to record the number of words that were actually referenced in each subblock. The tag store is then updated, and the respective subblock of the block being fetched is marked valid and the data loaded. At this time, a bit for the actual referenced word is also set. Misses to subblocks that belong to blocks already allocated in the cache carry less penalty; the subblock is fetched, its valid bit turned on and the referenced word is marked valid. At the end of the simulation run, we collect validity statistics on all blocks still resident in the cache.

Applications Overview

For performance simulation, we selected two major scientific application programs. We avoided benchmark suites since they generally do not exercise the cache sufficiently, and we are also interested in the performance of the KSR1 data subcache on a real application. We believe that understanding memory performance on real applications is a major tool for deducing restructuring techniques to improve such applications. We study two main scientific application programs in this work.

FEMC is a radiation modeling application developed at the University of Michigan [18] . It determines the frequency response obtained from broadcasting a pulse of electromagnetic radiation at a solid object, where the object and the surrounding space is modeled as a mesh of finite elements. The discretization of the Maxwell equations leads to a system of complex linear equations which is solved using a diagonal-preconditioned symmetric biconjugate gradient method. The method iteratively refines an approximate solution of the given linear system until convergence. We used a version of FEMC that we refer to as FEM3K. It was run with a 3KB dataset for 90 time-steps on a single KSR1 processor, generating 36.5 million memory references.

The other program is also a finite element application we call FEA. FEA follows a common parallel program structure [21] that consists of a series of parallel sections, each enclosed by a parallel construct that enforces barriers at the beginning and end of each section. The parallel sections may be separated by serial sections. The entire sequence is enclosed in an iteration loop where an iteration corresponds to a time step. For our purposes, we run FEA on a single processor and collect traces for two time-steps, totalling 7 million references.

Cache designs must be evaluated in an accurate and time-efficient manner. To obtain accurate and typical cache behavior, trace-driven simulations require programs that are realistic and representative of typical programs, such as the applications we have described above. However, traces that are long enough to properly evaluate many types of systems are almost impossible to obtain or even store. Ideas like *trace sampling* [20] and *time sampling* have been suggested to address this problem. To limit our trace size and still have traces long enough for proper evaluation, we use *selective tracing*. In most scientific applications, memory reference behavior is very similar across time-steps of the same loop. It is therefore our conjecture that a substantial percent of the time-steps is highly representative of the main procedure, or loop, memory behavior. For example, the memory reference behavior of a finite element procedure or loop that requires 300 time-steps to converge can very well be captured by tracing the first 50 time-steps. It is important that more than one time-steps are run to reduce the impact of possible initialization and cold start effects.

PERFORMANCE METRICS, EVALUATION AND ANALYSIS

Miss Ratios

Cache miss ratio is the total number of cache misses divided by the total number of memory references throughout the execution of a program. Two kinds of cache misses constitute the total misses in sectored caches: block misses and subblock misses. Non-sectored caches only have to consider block misses and we can always expect those misses to be significantly fewer than the total misses in a corresponding sectored cache with the same cache size and block size. Quite important, however, is the respective cost paid per cache miss. We compare miss ratios using *relative miss ratio* defined as:

Definition 1: (Relative Miss Ratio): *Let C denote a non-sectored cache with size CS, block size B, and associativity degree A. Let C_1 denote a sectored cache with size CS, block size B, and associativity degree A. We define relative miss ratio (RMR) of C_1 to C as the miss ratio on C_1 divided by the miss ratio on C.*

For a fairly simple machine organization, it is easy to use the miss ratio for predicting more direct measures like memory access latency and program execution time. However, as machine organization gets more complex a major factor that cannot be ignored for performance analysis is the particular penalty associated with various kinds of misses. If there are misses for which a lesser cost is paid as against those requiring substantially higher costs, then we can employ design techniques that do increase these inexpensive misses as long as the overall (average) miss penalty is reduced. Hence, it can be argued that a relatively higher miss ratio, by itself, is no longer a sufficient metric for comparing relative performance between a sectored and a non-sectored cache even if they have the same block configuration and management policies since, after all is said and done, the goal of a memory hierarchy is to reduce program execution time, not cache misses.

Miss Penalty

Miss penalty is the cost incurred in both time and space in servicing a cache miss. Characterizing this metric for a sectored cache design is becoming increasingly difficult due to the non-uniformity of cache block invalidation and the rate at which these invalidations are necessary, especially in coherent shared cache environments.

The miss penalty is highly dependent on the data transfer unit size of the cache, the memory latency and the transfer rate. A sectored cache miss results in one of three penalties:

> Case 1: A subblock miss, where the referenced block is already allocated in the cache: Let L_m be the memory latency to the first word, B_c be the time per word (8-bytes) transfer and W_{sb} be the subblock size in words, then the miss penalty is given by: $L_m + B_c \times (W_{sb} - 1)$
>
> Case 2: A block miss, where the miss results in a cache block replacement and the block that is being replaced is clean. Hence the only added overhead is the cost for allocating the new block. Let this cost be A_c, then the miss penalty is: $A_c + L_m + B_c \times (W_{sb} - 1)$
>
> Case 3: A miss that results in a cache block replacement and the block being replaced is dirty. The cost of allocating the new block in this situation includes the write-back time. Suppose the write-back time is *WB*, then the miss penalty is given as: $WB + A_c + L_m + B_c \times (W_{sb} - 1)$

There is no clear distinction between cases 2 and 3, since quantities A_c and *WB* are very design dependent. Invalidating a cache block can be a costly operation, requiring at the very least a write to the address tag store to invalidate the block. If the block being invalidated has data marked dirty then the overhead may be larger. However, this might not be the case with many designs. Some systems have the capability of hiding this write-back latency. A common technique is to dump the stale data in a write buffer and proceed to service the cache miss while the write completes in the background.

Due to the difficulty in determining accurate miss penalties, we analyze our applications by counting events like miss rate and words transferred, while ignoring the time factor at this time.

Memory Traffic

High memory bandwidth requirements can lead to degraded performance in a system. A cache miss results in a full data transfer (block in non-sectored caches, subblock in sectored caches) from the next higher level of memory to the cache. If some of the words in the block/subblock are unused before the block is replaced then some bus bandwidth and transfer time was wasted in fetching the unused portion of the block. Obvious causes that can be cited for fetches of unused data include isolated scalar references and non-unit strides, and isolated gather/scatter references to data structures. We expect this effect to be higher for smaller caches where there is less chance that words in a block will be used before the block has to be replaced.

Definition 2: (Memory Traffic Ratio)*: Memory Traffic Ratio (MTR) is defined as the number of words transferred between main memory and a data cache divided by the total number of memory references.*

MTR can be viewed as the average number of words transferred per memory reference. The MTR of a cacheless system is therefore equal to 1. MTR grows with data transfer unit size and shrinks with data reuse, and will exceed 1 for large transfer unit size and poor reuse. However, even an MTR greater than 1 for a cache system is no indication that a corresponding cacheless system will show better performance. As we saw earlier, miss penalty is linear in, but not proportional to transfer unit size. Hence the cumulative miss penalty for a program executed in the cacheless system could far exceed that for the same program in a cache system.

Superfluity

Superfluity addresses the cost of unused cache space. We introduce two metrics for measuring superfluity in a cache: *Block-size superfluity coefficient* (Sf_c) and *Fetch-size superfluity coefficient* (Wf_c). Both Sf_c and Wf_c are in the range $0 <= [Sf_c, Wf_c] < 1$. Sf_c equals 0 if every subblock in a loaded block in the cache is referenced at least once before the block is replaced. It approaches 1 for very poor usage of cache space. Wf_c is 0 if every word loaded into the cache is referenced by the processor and approaches 1 if there are a lot of unused words. Both Sf_c and Wf_c are needed for proper superfluity evaluation of a sectored cache. Non-sectored cache superfluity analysis requires only Wf_c since fetch size is the same as block size, implying $Sf_c = 0$.

Block-size Superfluity Coefficient: Sf_c is a measure of space wasted in a cache block. In particular, it is the fraction of the subblocks in allocated blocks that are still invalid when the block is replaced, averaged over all blocks loaded into the cache. It thus measures the average fraction of subblocks that are never referenced during a block's allocated lifetime in the cache. A more detailed picture of subblock usage is a histogram FREQ = $<FREQ_1, FREQ_2, ..., FREQ_s>$, where $FREQ_i$ is the number of block replacements that contain i valid (i.e. referenced) subblocks, and s is the number of subblocks in a block. The accuracy of $FREQ_i$ is further improved by including data for the blocks that are still resident in the cache at the end of the run. We obtain the block-size superfluity measure as:

$$Sf_c = 1 - \frac{\sum_{i=1}^{s} i \times FREQ_i}{s \times \sum_{i=1}^{s} FREQ_i} \qquad \text{(EQ 1)}$$

Fetch-size Superfluity Coefficient: Wf_c has implications for both space in the cache and cache-memory traffic. Wf_c is a measure of spatial locality, superfluous data and redundant traffic. It gives an indication of the ratio of average words not used per valid (i.e. referenced) subblock in the cache. Since on each miss, one subblock of data is fetched into the cache, we can calculate the portion of the MTR due to superfluous data. For average word usage per subblock, consider the histogram freq = $<freq_1, freq_2, ..., freq_{W_{sb}}>$, where $freq_i$ is measured when each block is replaced and at the end of the run and is equal to the number of valid subblocks (summed over all such blocks) that contain i accessed words, and W_{sb} is the number of words in a subblock. Wf_c is calculated as follows:

$$Wf_c = 1 - \frac{\sum_{i=1}^{W_{sb}} i \times freq_i}{W_{sb} \times \sum_{i=1}^{W_{sb}} freq_i} \qquad \text{(EQ 2)}$$

A superfluity coefficient near 1 indicates that a smaller block or subblock, or restructuring the application code or its data structures may achieve better exploitation of locality in the cache. This is because a high measure of superfluity is an indication of poor spatial locality in the cache; and may be due to a poorly structured application code and/or excessively large cache block.

Definition 3: (Traffic Superfluity Ratio)*: We define Traffic Superfluity Ratio (TSR) as the portion of the Memory Traffic Ratio* *that is superfluous. TSR is calculated as:*

$$TSR = Wf_c \times MTR \qquad \text{(EQ 3)}$$

TSR is thus the average number of superfluous words transferred per memory reference. In some sense, TSR may be thought of as the pollution ratio.

PERFORMANCE EVALUATION AND ANALYSIS

In this section, we present our simulation results and analysis for the KSR1 data subcache. We omitted the instruction subcache in this study mainly for the reason that all the configurations that we consider are big enough for our application working set to fit in the cache.

The behavior of sectored caches was measured on the data and compute intensive programs described in Section 3.3. Besides determining how the miss ratio and relative miss ratio change based on particular cache structures, we computed superfluity coefficients (Sf_c and Wf_c), the average traffic ratio between the cache and the main memory (MTR) and the average superfluous traffic ratio (TSR). For testing the effect of a particular cache parameter, a base-line cache was defined and individual parameters were varied. We looked at cache sizes successively doubling from 64K to 1M. For each cache size, we fixed the subblocks per block at 32 and varied subblock sizes by successively doubling from 1W (8 bytes) to 32W. Every doubling of the subblock size also implies the doubling of the block size. The base-line cache was defined as 2-way, random replacement with write-back policy and no prefetching.

Cache Size

Effects of varying cache size on miss ratio have been studied extensively [7] but never in conjunction with superfluity analysis. Sectoring miss rates do not exactly follow the commonly accepted folk wisdom of traditional cache miss rate behavior. For example, doubling cache size does not necessarily reduce sectored cache misses by 30% as suggested by Smith [8] .

Table 2 gives the average miss rates for FEM3K with varying cache sizes. We have included the miss rate behavior for cache sizes 64K, 128K, 256K, 512K and 1M using different subblock sizes. Generally, as cache size doubles we see a gradual reduction in miss rate for FEM3K except that the gain is not substantial enough.

TABLE 2. Average miss rates for FEM3K per various subblock/block and cache sizes

	Words per Subblock / Bytes per Block					
Cache Size	**1W/.25K**	**2W/.5K**	**4W/1K**	**8W/2K**	**16W/4K**	**32W/8K**
64K	58.97%	29.55%	15.14%	7.97%	4.93%	7.16%
128K	55.71%	27.90%	14.18%	7.28%	3.99%	3.09%
256K	53.18%	26.61%	13.44%	6.82%	3.54%	2.01%
512K	47.28%	23.69%	11.96%	6.05%	3.12%	1.75%
1024K	27.19%	13.73%	6.99%	3.57%	1.88%	1.01%

TABLE 3. Relative Miss Ratio for FEM3K for various subblock/block and cache sizes

Cache Size	Words per Subblock / Bytes per Block					
	1W/ .25K	2W/ .5K	4W/ 1K	8W/ 2K	16W/ 4K	32W/ 8K
64K	28.49	26.15	22.68	14.76	4.88	1.48
128K	29.63	28.18	25.78	21.41	10.23	2.66
256K	30.00	30.00	28.00	25.26	16.86	7.18
512K	31.31	31.17	29.90	26.30	18.35	7.95
1024K	31.61	31.20	30.39	29.75	31.33	25.25

As expected, the miss rates for various cache configurations is generally much higher than non-sectoring with the same block size. Table 3 shows the RMRs (relative to a non-sectored cache with the same block size and total cache size) for FEM3K. There is a correlation between RMR and the block-size superfluity of the application. The closer RMR is to the maximum number of subblocks in a block (thirty two, in our case), the better the block usage and the lower the block-size superfluity coefficients (Sf_c). Our explanation is that when FEM3K has a high RMR, the vast majority of the misses are subblock, not block misses and refer to already allocated blocks. This translates to good block usage (lower superfluity) and less penalty per miss.

Figure 5 plots block-size superfluity for FEM3K for all the cache sizes. Figure 5 gives a clear indication that as cache size doubles (also implying doubling the number of sets) there is an accompanying sharp drop in Sf_c, an indication of good cache space usage. All cache sizes show low Sf_c (below 0.3 for most cases) up to the 2K block size. For a cache size of 1024K, the very low Sf_c pattern strongly suggests that with its higher set size, the configuration is large

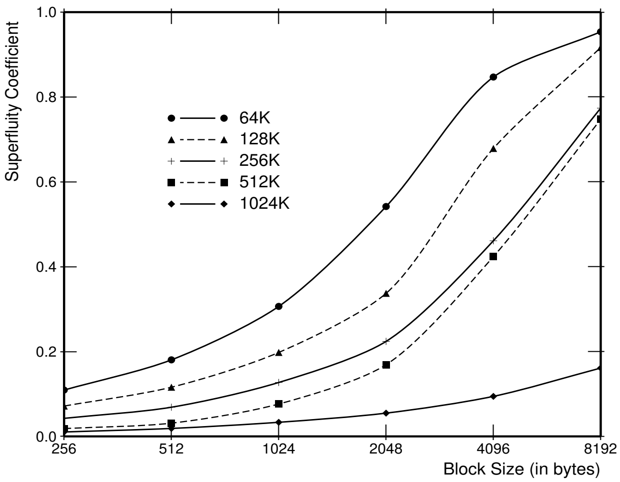

FIGURE 5. Block-size Superfluity Measure Sf_c for FEM3K. Cache sizes 64K, 128K, 256K, 512K, 1024K

enough to allow blocks to stay sufficiently long in the cache to be almost fully filled with active subblocks. Conversely, the relatively high Sf_c pattern for the 64K cache size can be blamed on the relatively small cache. For small cache sizes, pathological block conflicts are prominent resulting in blocks getting replaced faster before being fully utilized.

Similar analysis on FEA, which is a far more complex code, portrays a different picture, although the trends are similar. Table 4 and Table 5 show the average miss rate and RMR, respectively. We chose to exclude both extreme cache sizes (64K and 1024K) in order to focus first on the center of the table. The miss rates are higher than FEM3K for each cache size as expected. The RMR numbers, however, are very low given that there are thirty two subblocks per block. This indicates that the miss rates are high for non-sectored caches as well. Again, Table 5 indicates that as cache sizes double, FEA shows a gradual improvement in cache space usage. Unlike FEM3K, FEA exhibits a lot of misses that do not map into already allocated blocks, and the block-size superfluity remains high as shown in Figure 6. FEA shows poor usage of space in the cache since on the average blocks do not stay in cache long enough for full utilization. Overall, FEA exhibits very poor subblock utilization, as the Sf_c numbers indicate. Sf_c numbers approaching 1 indicate high superfluous behavior.

TABLE 4. Average miss rates for FEA per various subblock/block sizes

Cache Size	Words per Subblock / Bytes per Block		
	4W/1K	**8W/2K**	**16W/4K**
128K	27.72%	20.74%	19.80%
256K	24.56%	15.22%	11.03%
512K	23.50%	13.56%	8.63%

TABLE 5. Relative Miss Ratio for FEA per subblock/ block sizes

Cache Size	Words per Subblock / Bytes per Block		
	4W/1K	**8W/2K**	**16W/4K**
128K	4.76	2.71	1.72
256K	9.86	6.34	3.66
512K	13.91	10.11	6.30

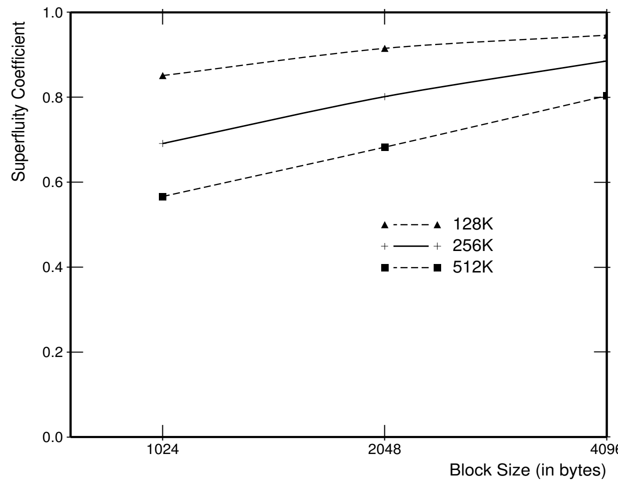

FIGURE 6. Block-size Superfluity Measure Sf_c for FEA. Cache sizes 128K, 256K and 512K

Fetch (Subblock and Block) Size Effects

Increasing the block size often helps cache performance because of spatial locality [8] but can result in a huge overhead in fetching the data during a cache miss. This overhead is what sectoring attempts to address.

A cache miss results in a fetch into the cache. Automatic prefetching is brought about if the fetch size is more than one word long. This can create a memory traffic amplification problem if the application has poor spatial locality. Also, if the size of the cache is small enough for significant mapping/conflict misses to be prominent, this can increase memory traffic. The same can be said about a given cache if block sizes are excessively long so that only few sets/blocks can reside in the cache. To evaluate this, we measured the effects of various fetch sizes while keeping cache size constant.

Table 6 gives the fetch-size superfluity measure Wf_c for FEM3K. The very low coefficients indicate that, almost all (except for large subblocks in small caches) words in a fetched subblock are utilized during the subblock's period of occupancy in the cache. Consequently, little superfluous data is transmitted to the cache for this application. The only cases under FEM3K that do not give near optimal space usage in the cache are: 1) using a 32W subblock with 64K and 128K size caches; and 2) using a 16W subblock with a 64K cache.

We expect the memory traffic to be generally low for FEM3K since, as Figure 8 shows, superfluous traffic is almost non-existent, except for the three cases we have pointed out. These three cases confirm our assertion that small caches with large blocks suffer a great deal from mapping/conflict/replacement misses. MTR explicitly for FEM3K for cache sizes 64K, 128K and 512K are shown in Figure 7. MTR numbers give the average words transferred between the cache and the main memory per memory reference; hence where this number is less than one the reuse effect overcomes the superfluity effect and the cache is reducing total memory traffic in addition to streaming it in subblock long bursts.

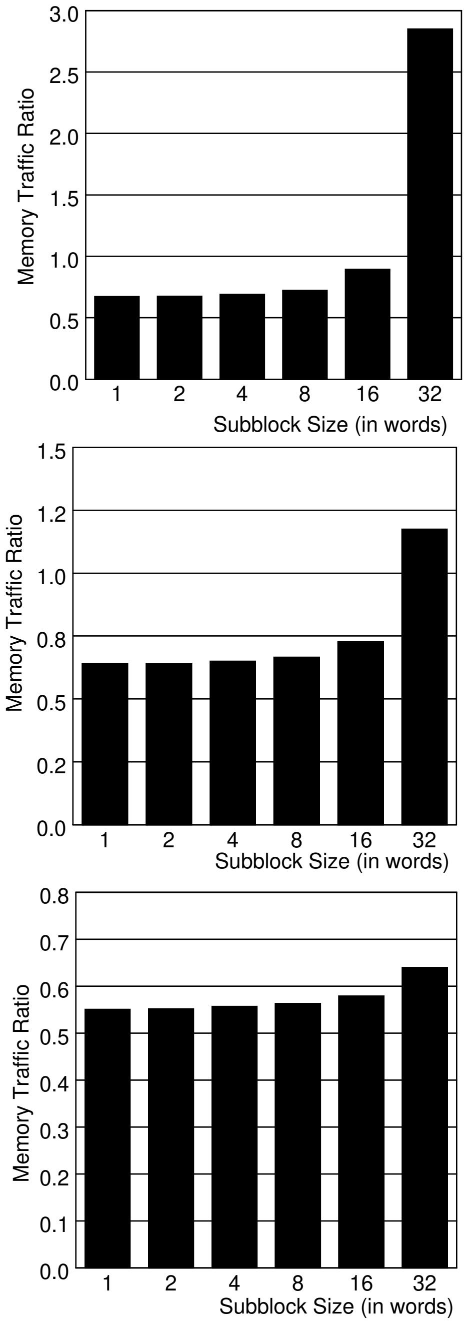

FIGURE 7. MTR plots for FEM3K. Cache sizes 64K, 128K and 512K.

TABLE 6. Fetch-size Superfluity Measure Wf_c for various subblock sizes (FEM3K)

Cache Size	Words per Subblock / Bytes per Block					
	1W/ .25K	**2W/ .5K**	**4W/ 1K**	**8W/ 2K**	**16W/ 4K**	**32W/ 8K**
64K	.0000	.0009	.0216	.0640	.2250	.6927
128K	.0000	.0006	.0132	.0331	.0998	.3443
256K	.0000	.0004	.0079	.0196	.0475	.1478
512K	.0000	.0003	.0037	.0102	.0314	.1147
1024K	.0000	.0004	.0015	.0034	.0062	.0117

Wf_c numbers for FEA are not encouraging as shown in Table 7. Higher coefficients indicate more superfluous traffic. Again we concentrate on the three moderate cache sizes (128K, 256K and 512K) and vary subblocks from 4 words to 16 words. As expected, the MTR is greater than 1 for all the cases we simulated. This indicates that the application has very low reuse of data (temporal locality at the word level). To show how much superfluous traffic occurs in each case, Figure 9 plots the MTR and TSR data side-by-side. Averaged over all cases, close to half of all data transferred to the cache are not utilized. The average superfluous traffic per memory reference as shown in Figure 9 clearly demonstrates the excessive burden placed on the cache-to-memory bus.

Observations

Our analysis of FEM3K and FEA reveals two interesting behaviors. FEM3K represents an application with good spatial locality. Most misses are to subblocks within already allocated blocks, which implies less penalty than unallocated block misses. High miss rate does not translate to more memory traffic, which implies high data reuse and low superfluity. FEA, on the other hand, represents a poor data locality application. A significant proportion of the misses cause allocation of new blocks, a strong indication that this application does not have small-enough stride or local gather/scatter reference behavior.

From our analysis, the KSR1 data subcache is a good cache configuration for FEM3K. We contend however that

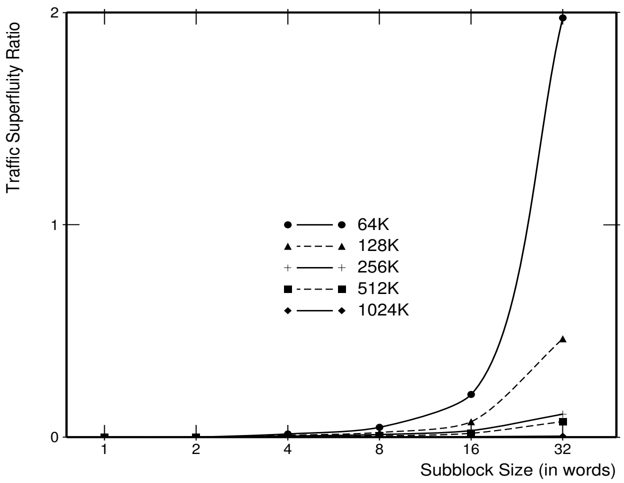

FIGURE 8. Traffic Superfluity Measure TSR for FEM3K. Cache sizes 64K, 128K, 256K, 512K and 1024K

TABLE 7. Fetch-size Superfluity Measure Wf_c for various subblock sizes (FEA)

Cache Size	Words per Subblock / Bytes per Block		
	4W/1K	**8W/2K**	**16W/4K**
128K	.2281	.4805	.7244
256K	.1361	.3021	.5175
512K	.1018	.2192	.3853

even such a good-behaving application could benefit further from a subblock prefetch policy. We have considered two such policies and found our preliminary results encouraging, but premature for presentation here at this time. The KSR1 cache configuration is, however, not appropriate for FEA and it shows very poor superfluity results over all configurations studied. In our view, such application requires code optimization techniques or a substantially different cache organization for performance improvement. A good cache configuration must deliver low superfluity, low cache-memory traffic, and misses with low total penalty.

We looked at other parameters like cache associativity and replacement policy. As expected, 2-way associativity performs better than a direct-mapped cache. In fact, with larger subblock sizes, direct-mapped configurations showed relatively higher superfluity coefficients. The performance differences between LRU and random replacement were not substantial enough to report, especially for large cache sizes.

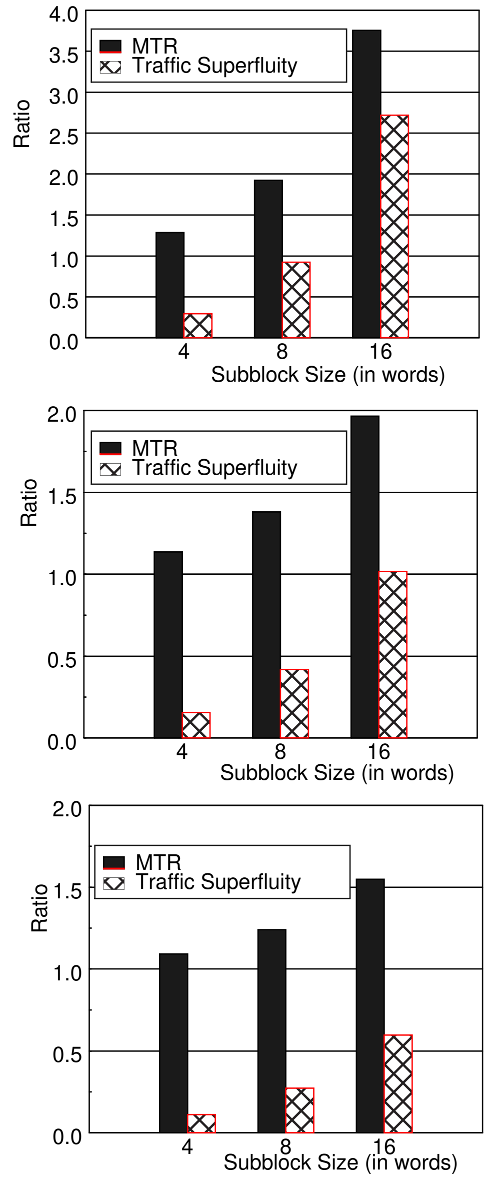

FIGURE 9. MTR and Traffic Superfluity comparisons for FEA (128K, 256K and 512K)

CODE OPTIMIZATION

Our discussion and results above demonstrate that cache performance depends not only on the design of the cache, but also on the reference patterns of the running programs. This, in our opinion, suggests that judging cache performance with benchmarks and generalizing the results must be done with considerable care. Benchmarks can be easily designed for a 100% hit ratio, but those results are meaningless when attempting to apply them to real applications.

Applications like FEA need restructuring to be able to exploit the gains that a cache can potentially provide for performance. There are a number of static (compile time) code optimization techniques for improving the data locality behavior of programs for both uniprocessor and multiprocessor environments [21] . Techniques of interest to us include loop unrolling, data blocking, data copying, and array grouping and copying. The last technique is attributed to Shih[21] .

To validate our evaluation technique, we studied an improved version of FEA that has undergone code restructuring with array grouping and copying [21] . We call this new version FEA.new. There is substantial improvement in superfluity as compared to FEA, an indication that the grouping increases the number of useful items in a block and reduces misses due to conflicts. Blocks will have a longer life occupancy in the cache, promoting better utilization. Figure 10 shows the memory traffic ratio comparison between FEA and FEA.new; and as Figure 11 shows, there is close to half reduction in superfluous traffic for all the cases studied. On the average, FEA.new displayed Sf_c coefficients that are a 10% improvement over FEA, and Wf_c coefficients that are a 20% improvement. The average 20% improvement in Wf_c can very well account for the decline in the superfluous traffic. Array grouping improves the code by stacking useful items together in a subblock/block.

CONCLUSION AND FUTURE DIRECTIONS

This study gives a broad overview of the technique of sectoring in caches for scientific applications while introducing a new set of metrics for cache performance evaluation; which stress on cache block and bus traffic usage. The norm has been that memory is cheap and space is free. It is our hope that the approach here will question this myth.

We have used the superfluity metrics in an attempt to understand the behavior patterns of real scientific applications; and also to help determine what cache parameters (like fetch size, block size, cache size) are adequate and appropriate for an application. We believe that these metrics can be helpful in determining and analyzing the spatial and temporal locality behaviors of a scientific application.

We plan to do more application and cache studies with our tool. The goal is to be able to extract common behavior patterns that will be helpful in choosing cache parameters like cache size, block size and fetch size that fit our "good cache" description. We expect that such studies will help us deduce better techniques for restructuring application codes to improve data locality and application performance.

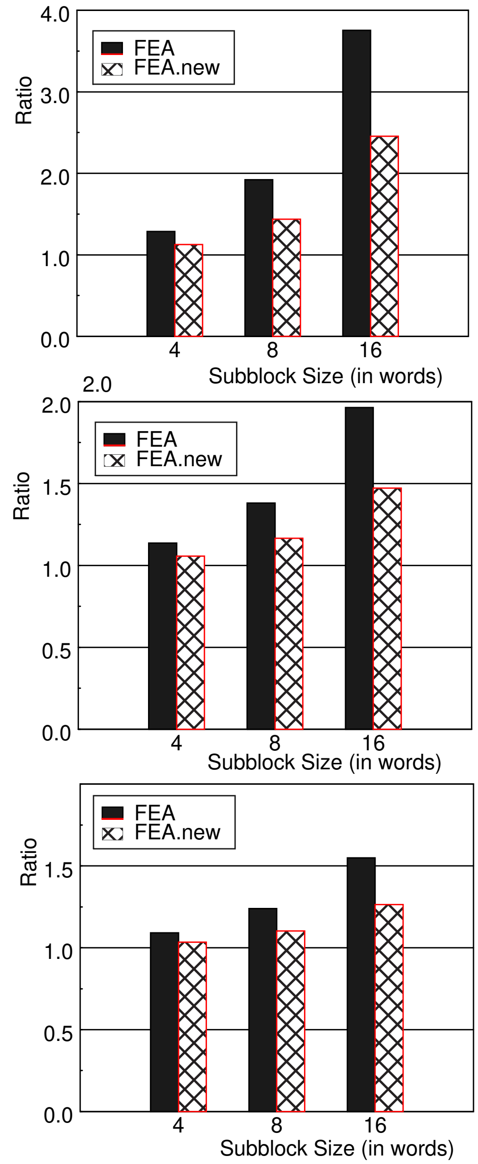

FIGURE 10. MTR comparison between FEA and FEA.new (128K, 256K and 512K)

REFERENCES

[1] Hill, M.D., and Smith, A.J., "Evaluating Associativity in CPU Caches," *IEEE Transactions on Computers*, December 1989

[2] Hill, M.D., and Smith, A.J., "Experimental Evaluation of On-Chip Microprocessor Cache Memories," *Proceedings of the 11th International Symposium on Computer Architecture*, June 1984

[3] Goodman, J.R., "Using Cache Memory to Reduce Processor-Memory Traffic," *Proceedings of the 10th International Symposium on Computer Architecture*, June 1983

[4] Eggers, S.J., and Katz, R.H., "The Effect of Sharing on the Cache and Bus Performance," *Proceedings of the 3rd Conference on Architectural Support for Programming Language and Operating Systems*, October 1989

[5] Liptay, J.S., "Structural Aspects of the System/360 Model 85, Part II: The Cache," *IBM System Journal*, Vol. 7 No. 1 1968

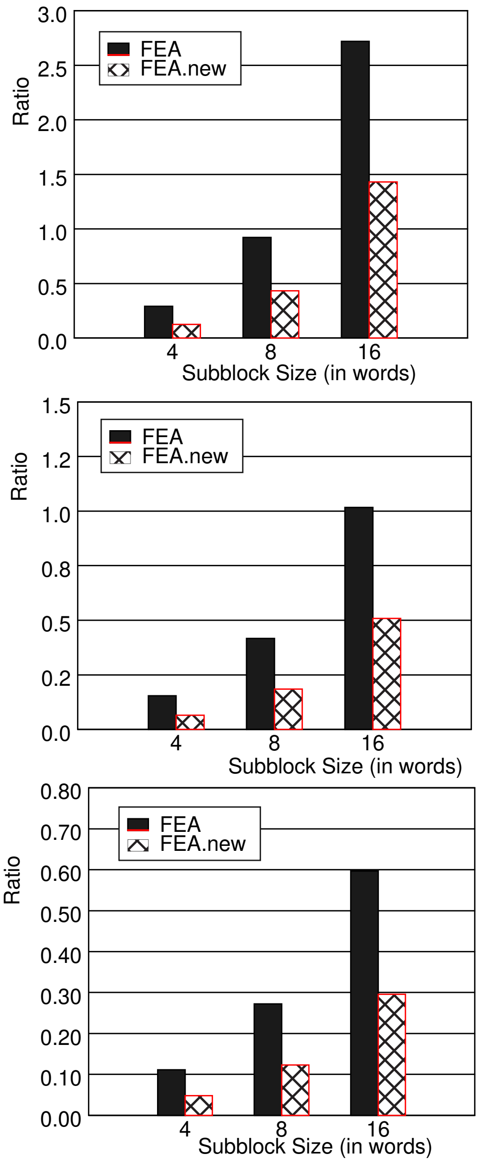

FIGURE 11. Traffic Superfluity comparisons between FEA and FEA.new (128K, 256K and 512K)

[6] Smith, A.J., "Bibliography and Readings on CPU Cache Memories and Related Topics," *Computer Architecture News*, January 1986

[7] Smith, A.J., "Second Bibliography on Cache Memories," *Computer Architecture News*, June 1991

[8] Smith, A.J., "Line (Block) Size Choice for CPU Cache Memories," *IEEE Transactions on Computers*, September 1987

[9] Seznec, A., "Decoupled Sectored Caches: conciliating low tag implementation cost and low miss ratio," *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994

[10] Torrellas, J., Gupta, A., and Hennessy, J., "Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System," *Proceedings of the 5th Conference on Architectural Support for Programming Language and Operating Systems*, October 1992

[11] Kane, G., and Heinrich, J., *MIPS RISC Architecture*, Prentice-Hall, 1992

[12] "TMS390Z50, Data Sheet", Texas Instrument, 1992

[13] "TMS390Z55, Data Sheet", Texas Instrument, 1992

[14] Pentium Processor User's Manual, Intel Corporation, 1993

[15] KSR Research Group, KSR1 Principles of Operations, October 1992

[16] Callahan, D., and Porter, A., "Data Cache Performance of Supercomputer Applications," *Rice University Technical Report*, May 1990

[17] Dubnicki, C., and LeBlanc, T.J., "Adjustable Block Size Coherent Caches," *Proceedings of the 19th International Symposium on Computer Architecture*, July 1992

[18] Chatterjee, J.J., and Volakis, J., "Application of Edge-based Finite Elements and ABCs to 3-D Scattering," *IEEE Transactions on Antennas and Propagation*, 1993

[19] Windheiser, D., Boyd, E.L., Hao, E., Abraham, S.G., and Davidson, E.S., "KSR1 Multiprocessor: Analysis of Latency Hiding Techniques in a Sparse Solver," *Proceedings of the 7th International Parallel Processing Symposium*, April 1993

[20] Laha, S., Patel, J.H., and Iyer, R.K., "Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems, "*IEEE Transactions on Computers*, Nov. 1988

[21] Shih, T., and Davidson, E.S., "Grouping Array Layouts to Reduce Communication and Improve Locality of Parallel Programs," to appear in *International Conference on Parallel and Distributed Systems*, December 1994

[22] Hennessy, J.L., and Peterson, D.A., *Computer Architecture, A Quantitative Approach*, Morgan Kaufman Publishers, 1990