Cache Resident Data Locality Analysis

Q. G. Samdani
Intel Corporation
galib.samdani@intel.com

M. A. Thornton
Mississippi State University
mitch@ece.msstate.edu

ABSTRACT

The data cache organization of a computer can significantly affect overall data access latencies when a program is executed. The cache performance depends on the locality characteristics of the data being processed in a program as well as the underlying architecture. A typical executing program has a data access profile that exhibits both temporal and spatial locality characteristics. Since most processors contain single data caches at a given level and since the single data cache cannot be optimized for purely spatial nor purely temporal locality data accesses, cache space pollution and inefficient usage of cache resources can occur. In the worst case, these phenomena can actually introduce additional data access latency through repeated line fills. Here an analysis and modeling scheme is presented that describes the runtime data access behavior of several benchmark programs in a typical, unified data cache. The motivation for the development of this model is to produce information that may aid in the design of a split data cache with one side optimized for temporal locality accesses and the other for spatial locality accesses.

1. Introduction

Cache memory usage is mandatory in state-of-the-art computers in order to reduce the overall data access latency resulting from slower main memory in comparison with the increasing speed of the processing element [1,2,3]. Cache memory hides data access latency by exploiting the locality characteristics of the running programs. A good knowledge of memory access behavior can lead to efficient cache memory subsystem designs. Memory access behavior is characterized by the principle of locality. Locality analysis of different types of programs during runtime can aid in defining an optimized cache subsystem organization. Locality modeling and analysis is necessary in research projects that investigate cache subsystem design. The locality behavior of a program can be of two types, spatial or temporal. Most current research projects [4,5,6] are investigating the spatial reuse of data of executing programs and strive to find a means to exploit this spatial reuse of data.

The purpose of the results discussed here is to analyze and model the actual runtime data access behavior in a typical cache. The analysis of the results motivates us to investigate alternative cache organizations that are able to intelligently use available resources. Instead of using a unified data cache, a cache organization that is split with each side optimized for a particular type of locality (i.e. spatial and temporal) has been proposed.

In the current literature, several investigations have proposed different schemes for instruction and data cache organization to reduce overall memory access latency. These include lockup-free caches [7,8], cache-conscious load scheduling [9], hardware and software pre-fetching [10,11] and multithreading [12]. The mechanism proposed in [9] identifies non-cacheable data by means of profiling. The scheme proposed in [10] is based on a run-time managed history table of the most recent load/store instructions. In [13] a pre-fetch engine is used which relies on a software or hardware optimized *Deterministic Prediction Approach* (DPA) in order to pre-fetch data that is estimated to be referenced in the future. Combined compile-time and run-time caching policies as proposed in [14] use memory access detection and automatic data caching based on compiler provided analysis of run-time memory access requirements.

A selective caching policy proposed in [14] leads to an organization that is similar to a conventional cache in which all memory instructions have an additional bit that is set (or reset) by the compiler. When a cache miss occurs, this bit controls whether a new block is retrieved from the L2 cache and placed in L1 or if the requested data is retrieved from the L2 cache directly and not written into the L1 cache.

A cache organization with both temporal and spatial subsystems has been proposed in [16,17,18] which uses a very simple heuristic based on the data type which can be changed by dynamic or pre-runtime profiling. Selective caching is a feature of current microprocessors such as that being used in the PowerPC. The HP PA-7200 [17] uses a software-managed data caching policy. Every memory instruction used by the HP PA-7200 includes a “hint bit” indicating that spatial locality is used to predict if the data referenced by that instruction shows only spatial locality characteristics and not temporal locality. The HP PA-7200 consists of two cache modules; the on-chip, fully associative, assist cache and a large direct-mapped off-chip cache. The assist cache holds data related to all memory references for which hint bits are explicitly set indicating spatial locality, whereas the off-chip main cache holds all data in which the hint bit is not set indicating the lack of spatial locality.

Current computer architectures typically use a fixed cache line size. Typically these cache lines can store a multiple number of memory words in a single cache line. This policy provides the advantage of prefetching a spatial memory zone on a miss. If the consecutive memory reference made by the processor is within the spatial space of the cache line, then the overall miss rate is reduced. The major drawback of using this policy is that data which exhibits temporal locality is stored with a set of data close to it in address space, but that may not exhibit any locality. Thus, the available cache space and bandwidth may be polluted by system when a large amount of non-usable data is resident in the cache. About 60 percent of available cache space can be polluted in some extreme cases due to this phenomenon [19].

To avoid cache pollution and latency, some intelligent spatial prefetching schemes have been proposed [6,19]. In [6] a *Spatial FootPrints* (SFP) table is maintained by using specialized hardware. Depending on the content of the SFP table, the predictor mechanism fetches a smaller or larger number of blocks when misses occur in the cache. Also, in [19] a somewhat similar strategy based on a *Spatial Locality Detection Table* (SLDT) is used to prefetch multiblocks or less in order to reduce memory access latency during runtime.

In this work, we present a model and the analysis of results from the model for determining the locality behavior exhibited by several benchmark programs executing in a load/store based uniprocessor with a typical unified data cache. The motivation for this analysis is to determine the data locality behavior of different programs, and to use the results to design an efficient cache organization that will not suffer from the inability to exploit varying data locality behaviors over a variety of executing programs.

The subsequent sections of this paper are organized as follows. Section 2 presents an overview of locality and the need for runtime locality analysis and modeling of executing programs. In section 3 we present the model that is used for the locality analysis. Next, we present and discuss experimental results of the cache access behavior by different SPEC integer and floating point programs. Finally, in section 5, we provide conclusions based on the experimental data.

2. Principle of Locality

To hide memory access latency due to fast processors with relatively slower main memory, a cache subsystem is used to attempt to store data which will be accessed in the future by the processor. This is accomplished by loading additional data other than that being requested by the processor during a cache line fill. A typical way of doing this is to retrieve additional data from the neighboring address space of the requested data. The purpose of writing neighboring data into the cache is to exploit the principle of spatial locality. Spatial locality exists due to the empirical observation that "data tends to be accessed that is close (in address space) to previously accessed data". Figure 1 shows an example of this type of locality where data block B is requested by the processor and the resulting cache miss causes a line fill to occur that loads blocks A through D. Thus, any consecutive memory blocks requested by the CPU within this spatial region will result in a cache hit and the access time of these data is equal to the (faster) cache access time.

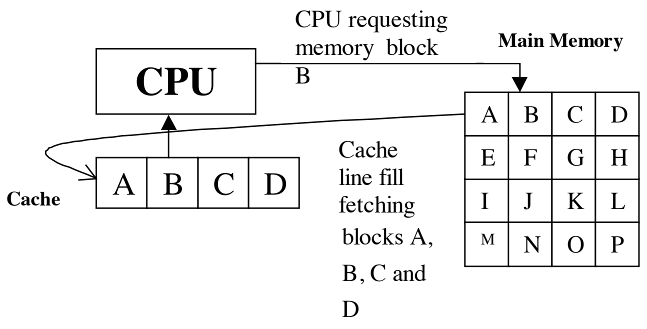
CPU requesting memory block B
Main Memory
CPU
A B C D
E F G H
I J K L
M N O P
Cache line fill fetching blocks A, B, C and D
Cache
A B C D

Figure 1: Cache Line Fill Illustrating the "Spatial Access"

Whenever the data access pattern is largely spatial in nature, the inclusion of large cache lines that contain more neighboring data can reduce the overall memory access latencies drastically. For strictly spatial data access patterns the reduction in memory access latency depends mainly on the cache line size.

Another type of locality is "temporal" locality or locality in time. This type of locality is characterized by certain locations in memory being accessed repeatedly in time. For example, this occurs when a CPU requests data blocks in the order B, G, M, B, G, M repeatedly during the execution of a program. The illustration shown in Figure 2 depicts this type of access pattern. In this case, the cache line fills are bringing additional memory blocks in each cache line that are not needed by the processor.

2.1 Motivation for Locality Analysis

In past work on data cache optimization, mainly numeric (scientific) programs have been considered for analysis of the data locality pattern. Since most numeric codes contain a large amount of nested loops, a significant amount of research has been attributed to the incorporation of more spatial reuse through different compiler optimization techniques such as unimodular transformations, loop fusion and distribution and tiling [20]. Some assertions of the spatial reuse of data have been made without doing any intra-loop reuse analysis [4]. Some computer architectures, such as the HP-7200 [17] do not use any detailed program locality information and depends only on spatial reuse of data.

To fully take advantage of the spatial locality present in a program's data access patterns and to also benefit from the temporal locality that is also present, a data cache may be organized with multiword line sizes. In Figure 1 it is seen that for the spatial access pattern B, C, D and A the access penalty is one cache miss since the next three consecutive accesses result in a cache hit. Thus, the effective miss rate is 25 percent and cache space utilization is 100 percent for this case. From Figure 2, if the access pattern is like B, G, M, B, G, M then the effective miss rate is increased to 50 percent and cache space utilization is reduced to 25 percent. This clearly indicates that the relatively large line size for taking advantage of spatial locality results in the pollution of the cache asset and also increases the memory access bandwidth. About 40% cache capacity wastage has been reported to be common [19].

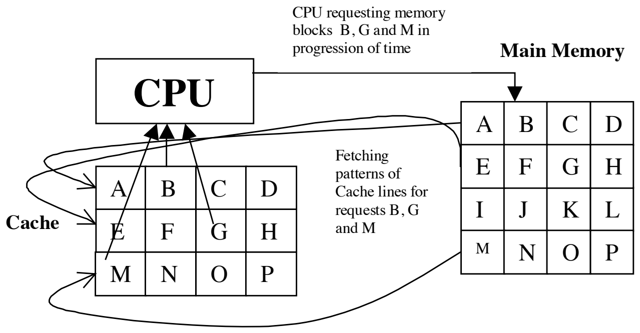
CPU requesting memory blocks B, G and M in progression of time
Main Memory
CPU
A B C D
E F G H
I J K L
M N O P
Fetching patterns of Cache lines for requests B, G and M
Cache
A B C D
E F G H
M N O P

Figure 2: Cache Line Fill Illustrating the "Temporal Access"

The depicted scenario indicates that the same cache organization will not perform equally well in all cases. To get an optimized performance, the cache organization needs to be tuned to benefit from both spatial and temporal data access behavior in executing programs. The tradeoff arises because increasing cache line size to exploit more spatial locality causes more cache pollution and wasted bandwidth when temporal accesses are requested. Alternatively, decreasing line size and adding more lines to a cache can result in inefficient usage when the accesses are largely spatial in nature since cache miss rates will increase. Furthermore, as is demonstrated later in this paper, the data access behavior varies largely from program to program. Data access behavior can be purely spatial, purely temporal or (more typically) a combination of both types of locality. It is possible to optimize a cache organization to provide optimum performance for a particular program. But, it is a very difficult task (if not impossible) to provide optimum performance for all types of program data access behavior. A reasonable choice in this case is to design a cache subsystem that will perform well on average. Analysis and modeling of the program data access behavior over a number of different programs can provide estimates of average-case behavior. This motivates us to carefully study and analyze the data access behavior of the programs that cover a wide range of applications. We use the SPEC benchmark suite as a representative sample of different types of application programs.

3.0 Locality Analysis Method

The data locality behavior of different application programs is analyzed during runtime in order to observe the characteristics of interest. In the results we present here, parameters of interest are generated through the accumulation of statistics based on data access patterns in a general cache as a program executes. In this approach, a specific cache architecture is considered and runtime data access profiles of different *SPEC* benchmark programs are stored. Initially, we modeled different cache sizes with varying line sizes. Among these, a four-way set associative 32 KB cache with 128 byte (32-bit words) line size was considered as the baseline organization to analyze and model cache data locality in terms of miss rates and a window width to capture both spatial and temporal locality. This target cache architecture was simulated using the ***C*** language and complied using the **`Unix cc`** compiler. Input to the program consists of memory traces gathered during the execution of the *SPEC* benchmarks.

The memory traces of the *SPEC* benchmarks used in this investigation are those available from the anonymous ftp site of the New Mexico State University Trace Database [21]. The traces contain the addresses of the memory references and also a field indicating whether it is instruction address or data read/write address. Since we are only interested with data caching, a filter program written in ***C*** to extract only the data load/store related addresses is used. The cache simulator then uses the data load/store related traces as input and generates the analysis results after simulating the cache.

For locality profiling purposes, the simulator keeps track of the number of accesses in each line of the cache as well as the average time difference of each word being accessed in a line over successive hits, or the "temporal stride". Although the term "stride" is generally used to refer to the absolute distance between different memory addresses, here we are using it in a temporal sense to refer the relative time difference in terms of the processor clock cycles. Also, the analysis tool records the number of hits for each word in a line. Analyzing the runtime behavior of the *SPEC* benchmark programs' memory traces allows the data access locality characteristics of these programs to be noted.

For the locality analysis, we use the line hit-rate and strides of the words in the lines as well as word hit frequency. Usually, for spatial locality, the strides of the words in a line should be similar or should have a fixed difference with an equal or close number of hits. For temporal locality behavior, the number of access to a line should become very high and we may also expect that the strides of the words and word-hit frequencies will vary a lot. Figures 3 and 4 show the typical nature of the strides for temporal and spatial locality behavior in a cache line for two benchmark programs used in this test bench.

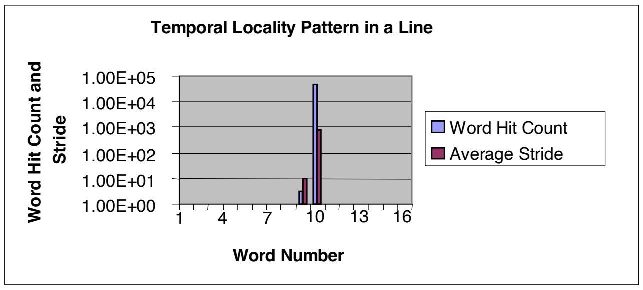

Figure 3. Temporal access pattern in a cache line

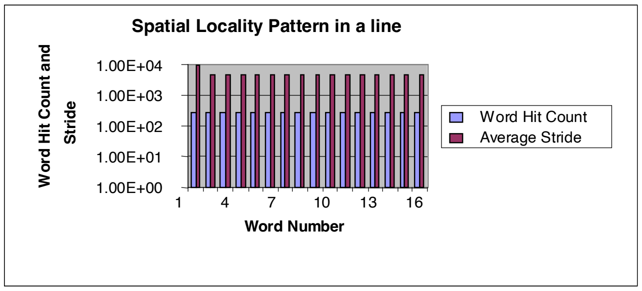

Figure 4. Spatial access pattern in a cache line

We used the following equation for the estimation of hit rate (in %) for spatial or temporal locality:

$$EHit_{Spatial/Temporal} = \left(\frac{100}{TWHC}\right) \sum_{i=1}^{N_{Spatial/Temporal}} WHC_i$$

Where:

EHit = Estimated percent of Hits due to Spatial or Temporal Locality

WHC_i = i^{th} Word Hit Count due to Spatial or Temporal Locality

$N_{Spatial/Temporal}$ = Number of Word Hits due to Spatial or Temporal Locality

TWHC = Total Number of Word Hit Count in the cache

To facilitate this estimation process, our model uses counters for each line of each set in the cache and for all corresponding words in the lines. Two-dimensional unsigned integer array variables are used to store the count values. The mapping process of a 4-way set associative cache is used to get the array indexes of these counter variables in a manner similar to hashing where the hash function is actually the cache mapping function. These counters are used to maintain the hit counts for each word in each line of the sets. For each respective word in the cache, the average time between successive hits is also maintained in another variable in terms of memory access cycles that we refer to as stride (in this case, temporal stride) in the plots. Figure 5 illustrates this basic strategy of counting the hits for a single 4-way set that contains 4 words per line.

As input, the analysis program uses memory traces obtained through the simulated execution of the *SPEC* benchmarks assuming a load/store CPU with the cache structure described above. After processing the hit rate and average stride of all words in the cache, the portion of the cache hits due to spatial and due to temporal accesses is determined. This determination is based on the 'hit count' and 'average stride' values for each word in the cache and is compared with the other words' hit count and stride values. For spatial accesses, the hit count and stride should be similar in value for each word in relation to the other words in a specific line of the cache. This observation forms the basis of how spatial locality is detected. The spatial accesses are isolated by simple relative comparisons of both the word and total line hit count values. For temporal accesses, the words with large differences in stride and hit count as compared with other words in the line are considered and their cumulative counts are also recorded for each line. Following the same process for all of the lines in the cache, a combined set of statistics based on spatial, temporal and unused word counts are obtained to calculate the percentage of cache hits due to spatial versus temporal locality. Figure 6 shows a flow diagram illustrating the major steps of the analysis method.

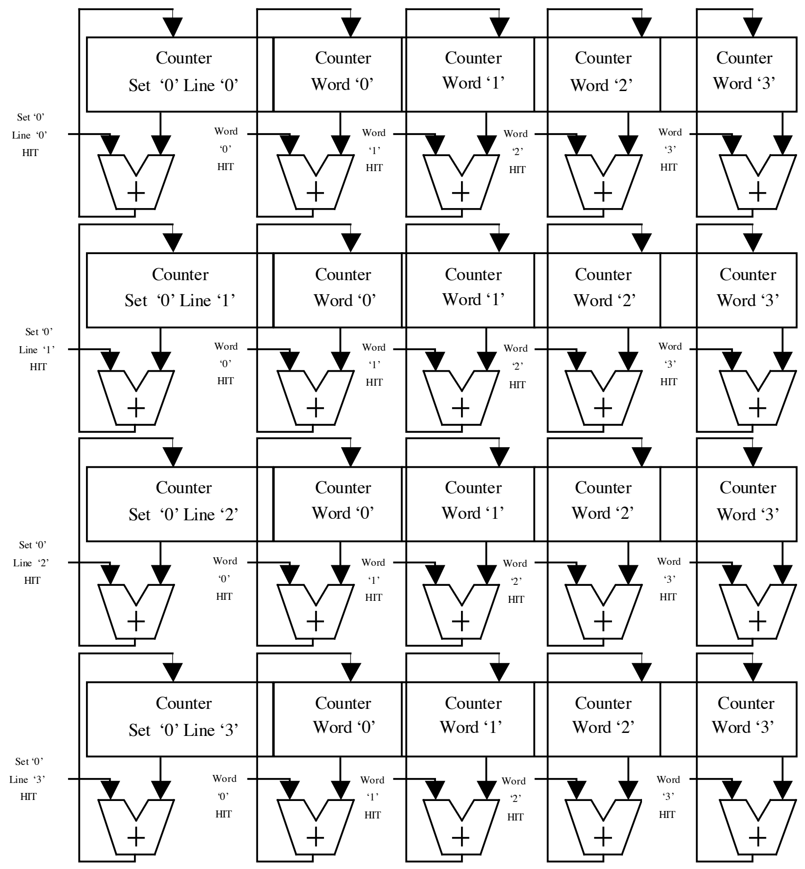

Figure 5. Line and word hit count strategy in a set

4.0 Data Locality Analysis Results

Data locality behavior of several *SPEC* integer and floating point programs is shown in Table 1. From the data locality behavior of the benchmark programs, it is apparent that the data access patterns do not show purely spatial or temporal locality in any case. The ratio of spatial versus temporal locality varies from program to program. These results indicate that the ***spice2g6, gcc*** and ***doduc*** benchmarks have a bias toward more temporal locality. Table 1 also indicates that most of the benchmark programs have some significant amount of temporal locality. The average spatial locality is 68 percent and average temporal locality is 32 percent for the SPEC benchmark programs that were used in this study. The spatial and temporal locality distributions of the SPEC benchmarks are shown in Figures 7 and 8.

Figure 9 shows the pollution of cache space due to spatial fetching of data in the cache lines. The results suggest that, on average, 23% of the available cache space is polluted by the spatial pre-fetching of data. In an extreme case the pollution was 62% (***wave5***).

Careful analysis of the results suggests that the address space of the memory references could be pre-dominantly spatial, pre-dominantly temporal or a combination of each. This is illustrated in Figure 11 where set A represents accesses that exhibit spatial locality and set B indicates those with temporal locality. The results indicate that programs typically contain a subset of accesses that have characteristics of both sets A and B. The intersection of these two classes of memory access types is represented by the set, C, in Figure 11. As an example, consider a program that consists of several consecutive loops, each of which accesses an array of data sequentially. Clearly, the accesses within a single loop are spatial in nature, however examining the access pattern of a single array element is temporal in nature due to the existence of multiple loops, and hence, multiple accesses of the same element.

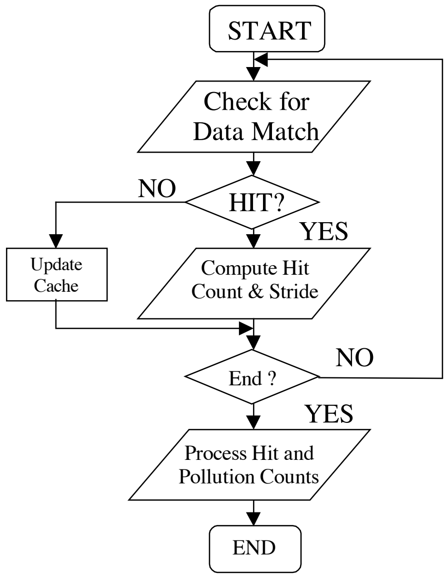

Figure 6. Diagram illustrating the cache data analysis

Table 1: Locality behavior of *SPEC* benchmark programs

Benchmark	Spatial Reuse (%)	Average Spatial Reuse (%)	Temporal Reuse (%)	Average Temporal Reuse (%)	Cache Space Pollution (%)	Average Space Pollution (%)
espresso	0.54	0.68	0.46	0.32	0.34	0.23
spice2g6	0.38		0.62		0.25	
doduc	0.45		0.55		0.01	
li	0.54		0.46		0.07	
eqntott	0.67		0.33		0.18	
compress	0.63		0.37		0.01	
mdljdp2	0.64		0.36		0.28	
wave5	0.63		0.37		0.62	
tomcatv	0.99		0.01		0.14	
ora	0.90		0.10		0.61	
alvinn	0.79		0.21		0.15	
ear	0.81		0.19		0.10	
sc	0.55		0.45		0.40	
mdljsp2	0.49		0.51		0.31	
swm256	0.96		0.04		0.10	
gcc	0.44		0.56		0.10	
su2cor	0.87		0.13		0.01	
nasa7	0.99		0.01		0.38	

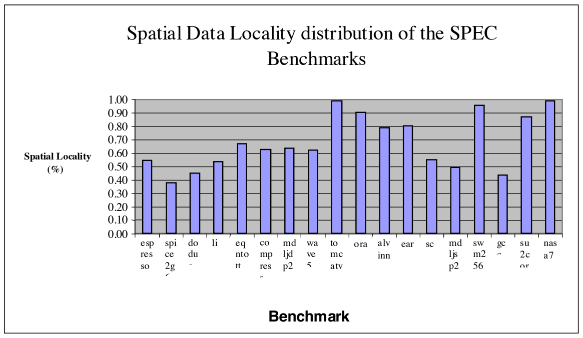

Figure 7. Spatial reuse patterns with *SPEC92* benchmarks

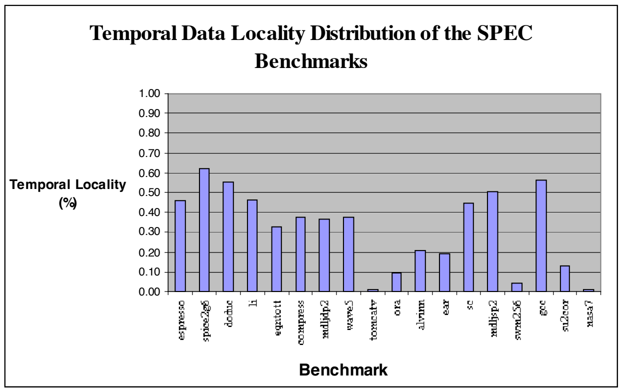

Figure 8. Temporal reuse patterns with *SPEC92* benchmarks

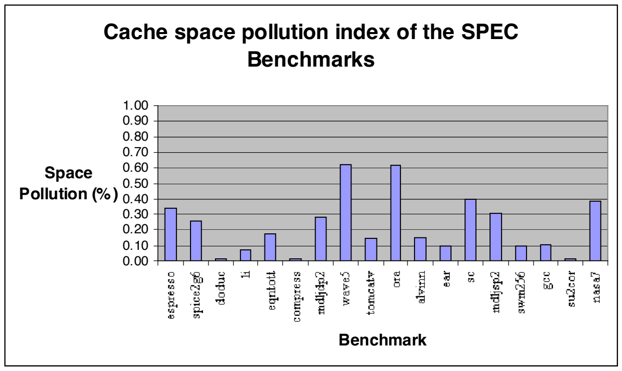

Figure 9. Cache space pollution using *SPEC92* benchmarks

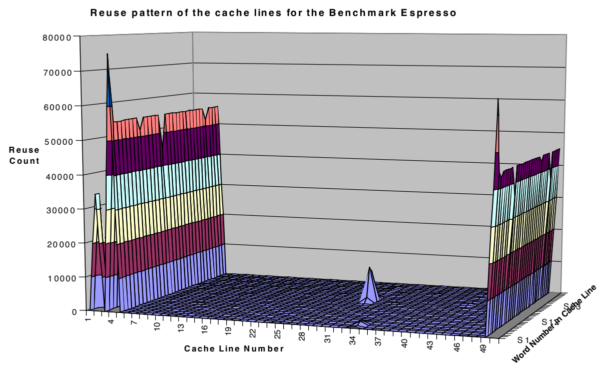

Figure 10. Reuse pattern of portion of cache space by the benchmark *espresso*

5.0 Conclusion

Based on the locality analysis made above, we conclude that data access behavior of different programs needs to be supported. Thus, both spatial and temporal locality data should be cached. Therefore, a split data cache is justified to facilitate both types of locality. A unified data cache can perform poorly in some cases by wasting valuable cache capacity. The data that should be cached in a spatial cache are those whose reuse frequency is good enough to allow for future cache hits. Since, spatial reuse is dominant in most of the cases, a relatively larger spatial cache with bigger line sizes should be used as compared to the temporal cache in the split data cache.

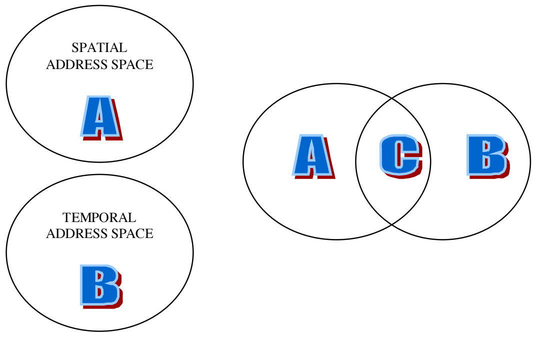

Figure 11. Diagram of Overlapping Spatial and Temporal Locality Characteristics

REFERENCES

[1] Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., and Yelick, K. "A Case for Intelligent RAM: IRAM", *IEEE Micro*, vol.17, (no. 2), March-April 1997, pp.34-44.

[2] Chen, T. F. "Reducing memory penalty by a programmable prefetch engine for on-chip caches", *Microprocessors and Microsystems,* 21, 1997, pp. 121-130.

[3] Handy, J. **The Cache Memory Book**, 2nd Edition, Academic Press, New York, 1998, pp. 188-198.

[4] McKinley, K. S., Temam, O. "A Quantitative Analysis of Loop Nest Locality", *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October, 1996.

[5] Sanchez, J. and Gonzalez, A. "Fast, Accurate and Flexible Data Locality Analysis", *Proceedings of Parallel Architectures and Compilation Techniques*, October 13-17, Paris, 1998.

[6] Johnson, T. L., Merten, M. C., and Hwu, W. "Run-time Spatial Locality Detection and Optimization", *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 57-64, Research Triangle Park.

[7] Hennessy, J. L., and Patterson, D. **Computer Architecture: A Quantitative Approach**, 2nd Edition, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996, pp. 390-426.

[8] Flynn, M. J. **Computer Architecture – Pipelined and Parallel Processor Design**, Narosa Publishing House, London, 1996, pp. 396-417.

[9] Abraham, S. G., Sugumar, R. A., Rau, B. R., and Gupta R. "Predictability of Load/Store Instruction Latencies", *Proceedings of the 26th International Symposium on Microarchitecture*, December, 1993, pp. 139-152.

[10] Callahan, D., Kennedy, K., and Porterfield, A. "Software prefetching", *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, April, 1991, pp. 40-52.

[11] Chen, T. F., and Baer, J. L. "A Performance Study of Software and Hardware Data Prefetching Schemes", *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April, 1994, pp. 69-73.

[12] Burger, D., Goodman, J. R., and Kagi, A. "Memory Bandwidth Limitations of Future Microprocessors", *Proceedings of the International Symposium on Computer Architecture*, 5/96, USA.

[13] Avila A. **Reference Prediction Based on Memory Access Patterns for Scientific Codes**, Ph.D. Dissertation, University of Arkansas, Fayetteville, December 1998, pp. 15-19.

[14] Dwarkadas, S., Lu, H., Cox, A. L., Rajamony, R., and Zwaenepoel, W. "Combining Compile-Time and Run-Time Support for Efficient Software Distributed Shared Memory", *Technical Report, Dept. of Computer Science, Univ. of Rochester and Dept. of Electrical & Computer Engineering, Rice University*.

[15] Sanchez, F. J., Gonzales, A., and Valero, M. "Static Locality Analysis for Cache Management", *Proceedings of Parallel Architectures and Compilation Techniques*, 1997.

[16] Milutinovic, V., Milutinovic, D., Ciric, V., Starcevic, D., Radenkovic, B., and Ivkovic, M. "Some Solutions for Critical Problems in the Theory and Practice of Distributed Shared Memory: Ideas and Implications", *IEEE Proceedings*, 1997.

[17] Chan, K. K., Hay, C. C., Keller, J. R., Kurpanek, G. P., Schumacher, F. X., and Sheng J. "Design of the HP PA 7200 CPU", *Hewlett-Packard Journal*, February 1996.

[18] Gonzalez, A., Valero, M., and Aliagas, C. "A data cache with Multiple Caching Strategies Tuned to Different Types of Locality", *Proceedings of ICS*, pp. 338-347, 1995.

[19] Kumar, S, Wilkerson, C. "Exploiting Spatial Locality in Data Caches using Spatial Footprints", *IEEE*, pp. 357-368, 1998.

[20] Muchnick, S. **Advanced Compiler Design and Implementation**, Morgan Kaufman Publishers, San Francisco, CA, pp. 687-697, 1997.

[21] "New Mexico State University Trace Database", Parallel Architecture Research Laboratory, (Online), Available: ftp://tracebase.nmsu.edu/pub/README., Accessed: January 15th, 2000.