Investigating the Use of Cache as a Local Memory

Lizy Kurian John, Raghuveer Reddy, Vijay Kammila and Peter Maurer
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620
kurian@csee.usf.edu

Abstract

When caches were first designed, they were very small by today's standards, and the components from which they were constructed were very expensive. For many of today's computer architectures, this is no longer true. Caches have become quite large, some as large as a megabyte, and, there is good reason to question whether the structure of a conventional cache was determined by the size and cost factors that prevailed when they were first developed. If this is the case, then different sorts of structures may be more appropriate for today's caches.

Our objective is to obtain a deeper understanding of useful program behavior that can be employed at optimizing program and data layout and to motivate architectural features aimed at selective exploitation of locality. In this paper, we present observations from several profiling experiments to advocate the possibility of fully or partially replacing the cache with a local memory, the local memory being used to house the heavily used references. We also present a memory structure called the SemiCache Architecture, which replaces the cache with a combination of cache and local memory, and present simulation results to prove the feasibility of the scheme.

1 Introduction

When caches were first designed, they were very small by today's standards, and the components from which they were constructed were very expensive. For many of today's computer architectures, this is no longer true. Caches have become quite large, some as large as a megabyte, and, there is good reason to question whether the structure of a conventional cache was determined by the size and cost factors that prevailed when they were first developed. If this is the case, then different sorts of structures may be more appropriate for today's caches. Investigation of such structures is especially important because the conventional cache has several disadvantages.

One major disadvantage associated with the conventional cache is that it exploits locality in a blind manner. Forcing all references through the cache, can reduce the effectiveness of the cache, because there is always the possibility that a once-only-used piece of data replaced a heavily used block of data. Another disadvantage associated with the traditional cache is the real estate consumed by address tags. The mapping schemes in caches, force us to incorporate memory to store tags, to facilitate identification of the data items. The directory portion of the cache can consume as much as 50% of the real estate within the cache [3].

Our objective is to obtain a deeper understanding of useful program behavior that can be employed at optimizing program and data layout and to motivate architectural features aimed at selective exploitation of locality. The objective of this paper is to present some observations on program behavior that supports use of a local memory for highly used data. We also present an architecture that replaces cache with a combination of cache and local memory, the local memory being used to house the highly used parts of the program. The complete details of implementation of a system that performs this efficiently is yet to be worked out. But the observations presented here will reveal the necessity for investigation of new approaches to design of the memory hierarchy.

The rest of this paper is organized as follows. In section 2, we present observations from profiling experiments that unveil the memory access characteristics of sample RISC programs. In section 3, the SemiCache architecture is described and preliminary simulation results are presented. In section 4, we offer summary and concluding remarks.

2 Locality Characteristics of Programs

Locality of reference is the fundamental motivation behind the design of the memory hierarchy. Locality of reference describes the fact that over an interval of time, the addresses generated by a typical program tend to be restricted to small sets of its name space. The working set model for program behavior was introduced by Denning [12] [13] [11] [10]. The working set may be defined as a set of recently referenced pages that estimated a program's memory demand in the immediate future. During the execution of a program, the working set quickly accumulates in the top level of the memory hierarchy to reduce the effective memory access time of a reference.

Locality characteristics of programs have been studied in the past by Denning [12] [13] [11] [10], Hatfield [15], Hagman et. al. [14] Darema-Rogers et. al. [34], Alanko et. al. [4], etc. These studies were performed years ago when processor register sets were very small and on-chip caches were almost non-existent. With the advent of RISC architectures, large register sizes, and excellent optimizing compilers, a detailed study of the memory reference patterns of large programs is in order. Program behavior studies in the RISC era include that of Cmelik et. al. [7], Stephens et. al. [35], Cvetanovic et. al. [8] etc. Cmelik et.al. analyzed characteristics of the MIPS and SPARC programs. Stephens et. al. characterized the IBM RS/6000 instruction set and the ALPHA instruction stream is analyzed in [8]. Except the RS/6000 studies by Stephens et. al.[35], none of them contain any analysis of locality.

2.1 Profiling Experiments

We performed profiling experiments on the DEC5000 workstation that uses MIPS R3000/3010 processors, using SPEC benchmarks as the workload. We used *pixie* and an in-house profiler program to gather the statistics presented in the paper. The *pixie* is a very powerful profiling program that generates a wide variety of information about the program being traced. We also developed a custom profiler program for the MIPS processors which profiles the executables of programs and traces the instruction and data reference addresses and frequencies, as the program is executed. It runs under a Unix environment and uses the Unix *ptrace()* utility to create the process to be traced as its child, single-steps it, and gathers the required data from the child's address space. In our program, we incorporate routines to calculate the frequency of each instruction and data address. The program generates histograms of instruction and data addresses. Parts of the information generated by our profiler are common to *pixie* output and hence could be cross-checked to ensure the correctness of our profiler.

Benchmark	Total Instr mem references	Distinct Instr mem references
espresso	521.4 X 10^6	17,493
eqntott	1,398.7 X 10^6	3,509
xlisp	6.012.9 X 10^6	5,352
tomcatv	1,372.6 x 10^6	5,169
dnasa7	8,889.3 X 10^6	15,392
fpppp	2,414.4 X 10^6	24,127
doduc	1,575.6 X 10^6	25,337
matrix300	2,347.6 X 10^6	4,880
compress	6,232.5 X 10^6	1,759
ear	93.7 X 10^6	6,144

Table 1. Unique instruction memory references in some of the SPEC programs

We use the SPEC benchmarks for our studies. Table 1 contains information about the static and dynamic code size of SPEC programs, as generated by *pixie*. An important observation can be made here. The SPEC programs make billions of instruction memory references during execution, but the number of unique instruction references made by the programs is considerably smaller. The number of unique instruction references made by SPEC programs range from approximately 1,750 to 25,000. Since each instruction requires 4 bytes on the MIPS, a cache of 7 KBytes to 100 KBytes will be sufficient to hold the entire program. The DECStation 3100 has a 64 KByte instruction cache. If instruction caches are large enough to hold entire programs, it is certainly worthwhile to investigate architectures that copy entire programs to the cache RAM before execution. The SPEC programs may be smaller than typical applications, but the 90/10 rule can be used to visualize that even when programs cannot be entirely contained in the cache, most significant parts can be.

2.2 Locality of Instruction Memory References

The 90/10 rule in computer architecture tells that most programs spend 90% of their execution time in less than 10% of the code. The 90/10 rule can be illustrated by presenting the cumulative percentage of instructions consuming a specific percentage of the execution time. Using *pixie* we can obtain details about the dominant basic blocks, dominant procedures, and dominant lines in the program. We use this information to quantify locality based on the percentage of instructions executed for a specific percentage of execution time. Our investigations show that in 6 out of the 8 traced programs, less than 20% of the code constitute 90% of the execution time. Stronger proofs on locality can be obtained by considering the dominant basic blocks or dominant lines rather than dominant procedures, because every line in the procedure may not be referenced with the same frequency. At basic block level, in 6 out of the 8 traced programs, 10% of the code constitutes 90% of the execution time (literally proving the 90/10 rule)!!!!

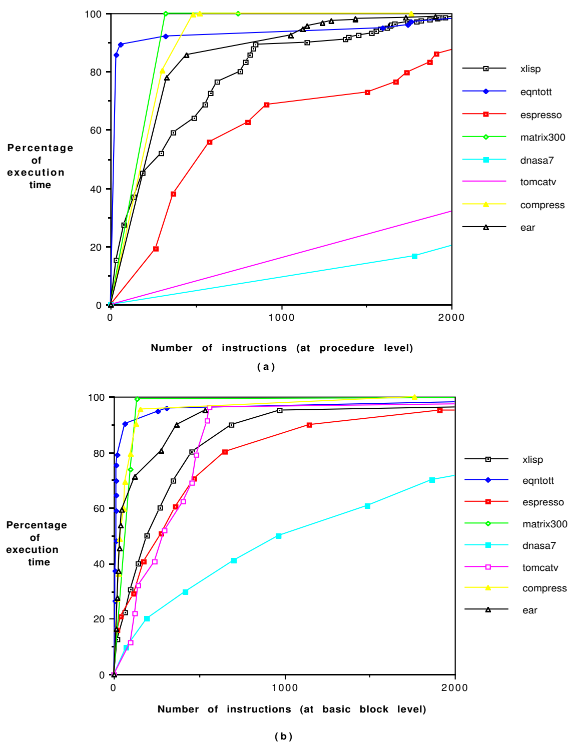

Figure 1: Instruction memory Reference Locality - Percentage of execution time versus number of instructions

Since we are interested in investigating the feasibility of using cache as a local memory, information about the absolute number of instructions constituting the bulk of the execution time is very important. Fig. 1 illustrates the percentage of execution time versus the number of instructions in the dominant parts of the program. Fig. 1.a is generated considering dominant procedures. and Fig. 1.b. is constructed considering the dominant basic blocks. We observe that in 5 out of the 8 SPEC programs, all procedures that constitute 90% of the execution time contain less than 1500 unique instructions. At basic block level, in 7 out of the 8 programs traced, 90% of all references are to less than 1000 unique instructions. In the IBM RS/6000 locality studies presented by Stephens et. al. [35], they also had observed that in 5 of 9 SPEC programs, less than 1000 unique instructions contributed to 90% of the execution time. Most modern processors contain on-chip instruction caches of size 8 or 16 KBytes which can hold 2000 to 4000 unique instructions, and hence it is reasonable to assume that dominant parts of most programs can be kept in small local memories for the entire duration of program execution.

Procedures and basic blocks with high frequency of reference are primary candidates for copying to the local memory if cache is converted to a local memory. Copying at a procedure level can be implemented for many programs. Our locality studies find that many large programs have procedures that are called millions of times. For instance, in *eqntott*, the procedure *cmppt* consuming 140 bytes is called 2,841,621 times and consumes 85.7% of the execution time. Any procedure referenced once or never are excellent candidates to be not copied to the local memory, in the event that the entire program cannot be held in the local memory. Program profiling has been successfully done by researchers in the cache profiling and cache conscious programming research area [26] [18] [36] [27] [17]. In some traditional FORTRAN programs, the entire program appears as a single procedure and copying at the procedure level may not be feasible. But dominant basic blocks can be easily identified and copied to local memory. With the current trend of modular programming, it is extremely easy to find small sections of heavily used code. Programmers themselves can give hints to the compiler as to which routines should be identified as local memory sections.

Phase diagrams were used by proponents of 'overlays' to plan a good overlay sequence. According to Denning [11], a phase diagram depicts program time as a sequence of phases and address space as a sequence of segments. We present the phase behavior of the instruction references in a few SPEC programs in Fig. 2.) Each phase in Fig. 2 (shown on the x-axis) represents execution of one million instructions. It may again be observed that a few addresses are favored by the program over the bulk of the execution time. There really is no need to cache the rarely used instructions. If a local memory that can house the heavily used instructions can be built, the system would function very efficiently.

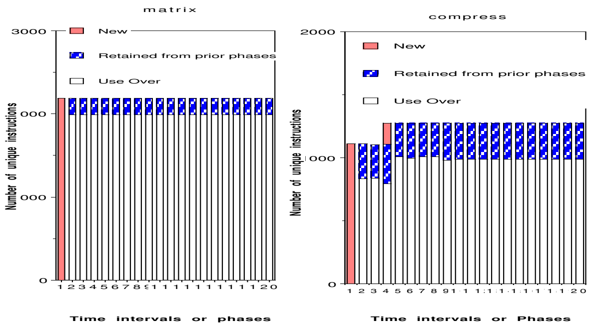

Figure 2: Phase behavior of instruction stream of example SPEC programs

2.3 Data Reference Locality

Next, we characterize the data locality of a few programs from the SPECsuite, *espresso*, *tomcatv* and *dnasa7*. For *espresso*, the entire run for input file *bca.in* was traced. For *tomcatv* and *dnasa7*, only 300 million references were traced and analyzed (due to time and storage limitations). In *espresso*, 520 million instructions make approximately 108 million data references to about 91K unique locations. In *tomcatv* and *dnasa7*, the first 300 million instructions make roughly 150 million memory references to 0.9 million and 0.1 million unique data locations respectively.

The frequency of referencing of each data memory address was computed by the tracer program. Then the frequencies of reference of the various data addresses are sorted and the dominant references are identified. The addresses in descending order of frequency of usage is presented in Fig. 3. The values on the horizontal axis indicate only a serial number. The vertical axis represents the logarithm (base 10) of the frequency of reference. The reference frequencies of some of the addresses are extremely high compared to the other addresses, and hence a log scale was chosen. (dnasa7 exhibited a behavior similar to that of tomcatv and hence was omitted from the figure due to space limit.) Fig. 4 illustrates phase behavior of a few of the SPEC programs. In *ear* and *espresso*, in each phase of the program, only a subset of the data is being used. In *dnasa* and *tomcatv*, the whole data set is being used in each phase. Based on this phase behavior, simple copying of all required data to local memory is unlikely to be feasible for data references. But even for caches, the performance of *dnasa* and *tomcatv* will not be good unless blocking algorithms as in [25] are applied. For local memory, effective algorithms to manage efficient and selective copying of data to the local memory, must be devised. Without efficient algorithms and optimization, caches and local memory will need to be extremely large in order to exploit any significant locality.

3 The SemiCache Architecture

In this section, we present the SemiCache architecture, obtained by converting parts of the on-chip and off-chip caches to a local memory (see Fig 5). The lo-

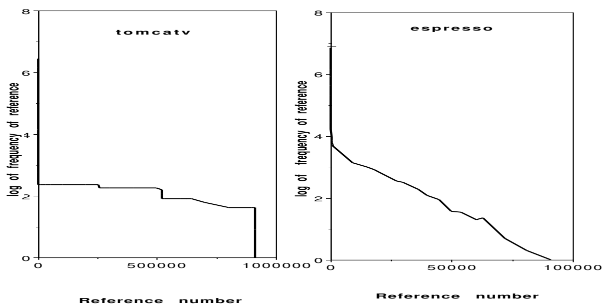

Figure 3: Data locality plots: Frequency of reference versus unique references in descending order of frequency

cal memory obtained by converting the on-chip cache is labeled LM1 and the off-chip local memory is labeled LM2. Program profiling is done to identify usage of instructions and LM1 is used to house heavy-use elements and LM2 is used to house medium use elements.

For every reference, the virtual address is first translated to physical address and searches checking for presence of data in LM1, LM2, on-chip cache and off-chip cache are initiated at the same time. If the desired data is in the on-chip local memory and can be accessed directly from the on-chip local memory. If the on-chip local memory does not have the requested address, but if the on-chip cache has the data, the cache can deliver it. If the reference misses in the on-chip cache as well, the miss is serviced by checking the main memory, bearing in mind that LM2 is considered as part of the main memory for addressing purposes. Since LM2 houses medium-use data which would not fit in LM1, there is a very high possibility that the data is in LM2.

If a process is being activated for the first time, program and data for the job will be transferred directly from the disk to the main memory (which consists of LM1 and LM2 as well) via a high speed bus. With context switching, the data in LM1 and LM2 can be put to elsewhere in the main memory, and data for the new job can be put to LM1 and LM2. Let us say there are n processes that are multiprogrammed. Each process when active has its high frequency data in LM1 and LM2. If the process was active before but was switched out of LM1 and LM2 (no page fault), the high usage parts of code and data are moved to LM1 and LM2 while activating the process again. *This tends to create the picture of using the local memory as a CPU-local swap space.* If multiprogramming results in very frequent context switches with thrashing, moving data from slower parts of main memory to LM1 and LM2 occurs frequently and may constitute severe overhead. Different schemes to handle this problem are being investigated.

Preliminary simulations illustrate that the SemiCache outperforms a conventional cache. In Fig. 6.a, we present the total execution time for the SPEC

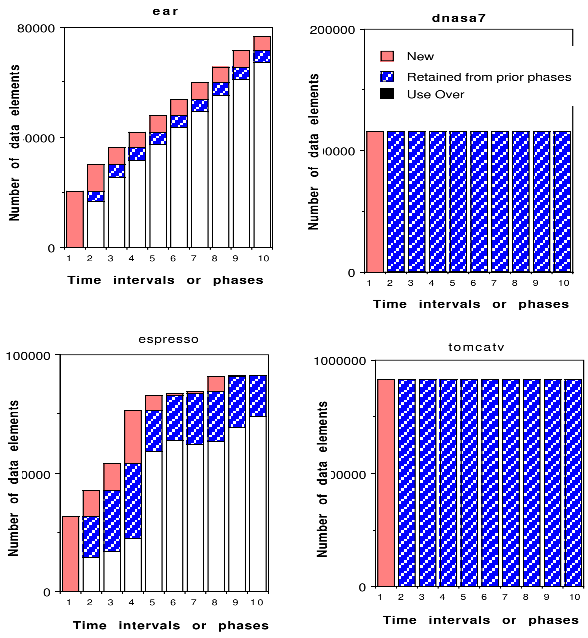

Figure 4: Phase behavior of data reference stream of example SPEC programs

benchmarks for pure cache, SemiCache, and all local memory (or NoCache) structures. For SemiCache, half the size is cache and the other half is local memory. Fig. 6.a shows that the SemiCache outperforms a standard cache. The performance of pure cache and SemiCache organizations suffer when a rarely used block knocks a frequently used item from the cache. In the All local memory (or NoCache) architecture, rarely used elements are able to bypass the cache, thereby preventing the pollution of the cache. Not surprisingly, the performance of the different techniques converges as the size approaches the size of the program. We also conducted several experiments to determine the performance of a system with a small amount of local memory in addition to the cache, the results of which are presented in Fig. 6.b. These results show that even adding a tiny amount of local memory to a conventional cache can significantly improve its performance. It may be noted that we considered only one layer of the cache and a single program workload for our simulations. Appropriate architectural changes have to be worked out to deal with context switches and multiple layers of caches.

4 Concluding Remarks

In this paper, we presented program behavior studies and preliminary simulations to show that partially or fully converting the cache to a local memory can improve the performance of memory access. We analyzed the SPEC programs and presented information about the usage frequencies and patterns of instruction and data addresses. There is sufficient basis to believe that managing the cache as a local memory will become a feasible alternative very soon.

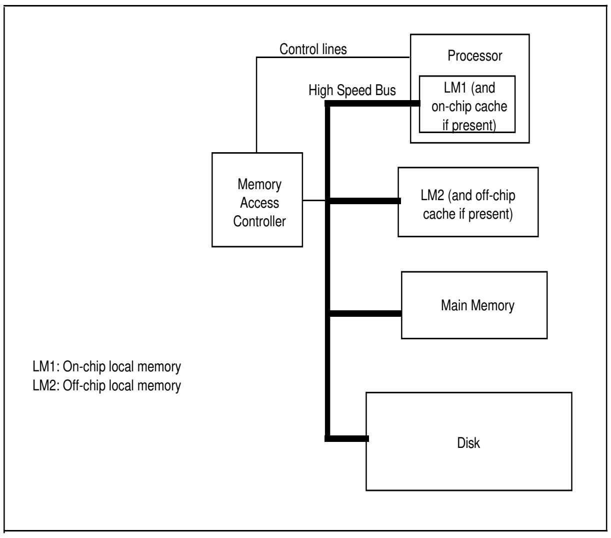

Figure 5: The SemiCache Architecture

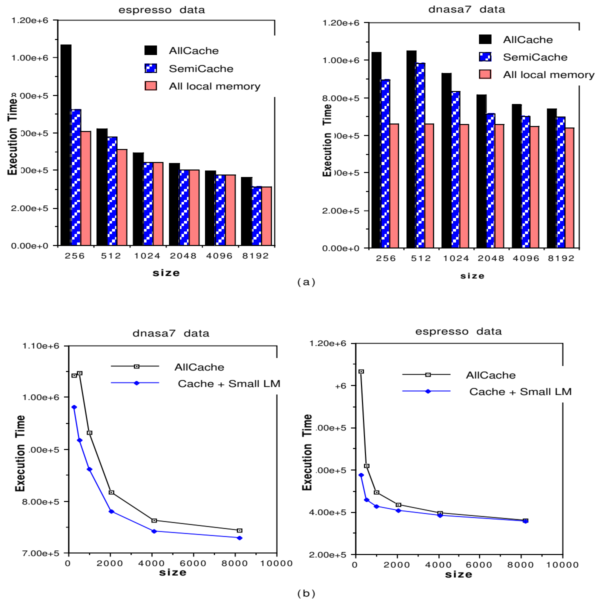

Figure 6: Performance of conventional cache, SemiCache and All local memory systems

Ultimately we believe that systems should migrate to a cacheless (all local memory) configuration, but this will require the development of on-chip caches in the megabyte range. In the initial phase of the transition, a small amount of local memory can be added to the cache; in the next phase, a major part of the cache can be converted to local memory, and in the final phase, the entire cache can be used as a local memory. The main disadvantage of the scheme is that it requires explicit loading of program and data segments into the local memory. To make a good choice of which segments to load, it may be necessary to profile the program. However, this is similar to the profiling [26] [18] [36] already taking place for cache-conscious programming, and, we do not feel that this is a particularly restrictive constraint.

References

[1] S. G. Abraham, R. S. Sugumar, B. R. Rau and Rajiv Gupta, "Predictability of load/store instruction latencies", In Proceedings of MICRO-26, 1993, pp. 139-152.

[2] A. Agarwal and S. Pudar, "Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches", In International Symposium on Computer Architecture, 1993, pp. 179-190.

[3] D. B. Alpert and M. J. Flynn, "Performance Trade-offs for Microprocessor Cache Memories", IEEE Micro, August 1988, pp. 44-53.

[4] T. Alanko, I. Haikala and P. Kutvonen, "Methodology and Empirical Results of Program Behavior Measurements", Proceedings of the 7th IFIP W.G.7.3 International Symposium on Computer Performance, Modeling, Measurement and Evaluation, Toronto, May 1990, pp. 55-66.

[5] B. N. Bershad and J. B. chen, "Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches", Proceedings of ASPLOS-VI, pp. 158-170.

[6] C-H Chi ad H. Dietz, "Unified Management of Registers and Cache Using Liveness and Cache Bypass", Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, Vol. 24(7), pp. 344-355, June 1989.

[7] R. Cmelik, S. Kong, D. Ditzel, and E. Kelly, "An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks", Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems", 1991, pp. 290-302.

[8] Z. Cvetanovic and D. Bhandarkar, "Characterization of Alpha AXP Performance Using TP and SPEC Workloads", Proceedings of the 21st International Symposium on Computer Architecture, April 1994, pp. 60-70.

[9] D. Callahan, K. Kennedy and A. Porterfield, "Analyzing and Visualizing Performance of Memory Hierarchies", Instrumentation for Visualization, ACM Press, New York, 1990.

[10] P. J. Denning and S. C. Schwartz, "Properties of the Working-Set Model", Communications of the ACM, Volume 15, No. 3, 191- 198.

[11] P. J. Denning, "Working Sets Past and Present", IEEE Transactions on Software Engineering, Vol. SE-6, No. 1, January 1980, pp. 64-84.

[12] P. J. Denning, "The Working Set Model for Program Behavior", Communications of the ACM, Volume II, No. 5, May 1968, pp. 323 - 333.

[13] P. J. Denning, "Virtual Memory", Computing Surveys, Vol. 2, No. 3, September 1970, pp. 153-189.

[14] R. B. Hagman and R.S. Fabry, "Program Page Reference Patterns", Proceedings of the 1982 ACM Sigmetrics Conference on Measurements and Modeling of Computer Systems, Seattle, WA, 1982, pp. 20-29.

[15] D. J. Hatfield, "Experiments on Page Size, Program Access Patterns and Virtual Memory Performance", IBM Journal of R & D, Vol. 16, No. 1, Jan 1972, pp. 58-66.

[16] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for cache and local memory management by global program transformation", Journal of Parallel and Distributed Computing, No. 5, pp. 587-616, 1988.

[17] A. J. Goldberg and J. Hennessy, "Performance Debugging Shared-Memory Multiprocessor Programs with MTool", Proceedings of Supercomputing '91, IEEE CS Press, Los Alamitos, California, Order No. 2158, 1991, pp. 481 - 490.

[18] A. Gupta, M. Martonosi, and T. Anderson, "MemSpy: Analyzing Memory-System Bottlenecks in Programs", Performance Evaluation review, Vol. 20, No. 1, June 1992, pp. 1-2.

[19] K. Hwang and F. Briggs, "Computer Architecture and Parallel Processing", Prentice Hall, 1984.

[20] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Buffers", Proceedings of the 17th International Symposium on Computer Architecture, 1990, pp. 364-373.

[21] B. W. Kernighan, "Optimal Sequential Partitions of Graphs", JACM, Vol 18., No.1, 1971, pp. 34-40.

[22] R. E. Kessler, M. D. Hill, and D. A. Wood, "A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches', IEEE Transactions on Computers, Vol. 43, No. 6, June 1994, pp. 664 - 675.

[23] L. Kurian, P. Hulina, L. Coraor and D. Mannai, "Classification and Performance Evaluation of Instruction Buffering techniques", Proceedings of the 18th International Symposium on Computer Architecture, May 1991, Toronto, Canada, pp. 150-159.

[24] L. Kurian, P. Hulina, and L. Coraor, "Memory Latency Effects in Decoupled Architectures", IEEE Transactions on Computers, Vol. 43, No. 10, pp. 1129-1139, October 1994.

[25] M. S. Lam, E. E. Rothberg and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms", Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems", 1991, pp. 63 – 74.

[26] A. R. Lebeck and D. A. Wood, "Cache Profiling and the SPEC Benchmarks: A Case Study", IEEE Computer, October 1994, pp. 15-26.

[27] "MemSpy: Analyzing memory system bottlenecks in Programs", Proceedings of the ACM SIGMETRICS Conference, pp. 1-12, 1992.

[28] S. Mirapuri, M. Woodacre and N. Vasseghi, "The MIPS R4000 Processor", IEEE Micro, April 1992, pp. 10-22.

[29] J. C. Mogul and Anita Borg, "The Effect of Context Switches on Cache Performance", Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems", 1991, pp. 75 – 84.

[30] S. McFarling, "Program Optimization for Instruction Caches", Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems", 1989, pp. 183 – 191

[31] S. Palacharla and R.E. Kessler, "Evaluating Stream Buffers as a Secondary Cache replacement", Proceedings of the 21st International Symposium on Computer Architecture, April 1994, pp. 24-33.

[32] A. Porterfield, "Software Methods for Improvement of Cache Performance on Supercomputer Application", Ph.D dissertation, Rice University, 1989.

[33] B. R. Rau, "Pseudo-Randomly Interleaved Memory", Proceedings of the 18th International Symposium on Computer Architecture, May 1991, pp. 74-83.

[34] F. Darema-Rogers, G. F. Pfister, and K. So, "Memory Access Patterns of Parallel Scientific Programs", Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Banff, Canada, May 1987, pp. 46-58.

[35] C. Stephens, B. Cogswell, J. Heinlein, G. Palmer, and J. P. Shen, "Instruction Level Profiling and Evaluation of the IBM RS/6000", Proceedings of the 18th International Symposium on Computer Architecture, May 1991, pp. 180-189.

[36] J. Torrellas, C. Xia, and R. Daigle, "Optimizing Instruction Cache Performance for Operating System Intensive Workloads", Proceedings of the High Performance Computer Architecture Symposium, Jan 1995, pp. 360-369.