To appear in the PAID'98 Workshop Digest (in conjuction with the ISCA'98)

Data Locality Analysis of the SPECfp95

F. Jesús Sánchez and Antonio González

Dept. of Computer Architecture
Universitat Politècnica de Catalunya
Barcelona - SPAIN

E-mail: {fran,antonio}@ac.upc.es

Abstract

This paper presents a detailed analysis of the locality exhibited by the SPECfp95 benchmark suite. This study is performed by means of a tool that is based on a static analysis enhanced by a simple profiling. This new approach results in a fast, accurate and flexible data locality analysis tool. It is fast because its run-time overhead is almost negligible, having a slowdown of 0.05. Besides, a single run of a benchmark can provide information for a variety of cache configurations. The tool is accurate because the information that is unknown at compile time is obtained by a profiling step. Finally, the tool is very flexible in the sense that it can provide many different statistics about the locality exhibited by programs and it can evaluate a variety of cache architectures.

1. Introduction

Memory penalties are one of the main reasons why computers performance is quite below peak performance for most applications. Understanding the source of the problems is the first step towards devising new hardware organizations and/or new code transformations to overcome them.

The user may be interested in quantifying the memory penalties but this information is not enough in many cases. A more detailed explanation of the different causes for these penalties is sometimes required in order to investigate the appropriate optimization. Examples of the type of information that the user may be interested in are listed below:

- Classifying the different types of cache misses into the three commonly used categories (compulsory, capacity, conflict) can be important to choose different types of optimizations. Capacity misses could be best reduced by blocking [5][3]; conflict misses by padding [13]; and compulsory misses by prefetching [2][11], among other possibilities.
- Identifying the parts of the program that are responsible for most penalties may help to reduce the optimization effort by focusing on such cases.
- Conflict misses are the dominant type of misses for many numerical applications. Identifying which data structures are responsible for these conflicts may be required in order to eliminate them by means of padding [13] or copying [17], among other possibilities.
- Quantifying the intrinsic reuse of a program can be used as an upper bound of the locality that can be exploited. This is a useful measure in order to know how far from optimal the current performance is.
- Evaluating the memory performance for a variety of cache architectures for a set of applications can be interesting for the design of an embedded processor with a cache memory optimized for such programs.
- Including some bits in the memory instructions so that the compiler can provide some hints to the hardware regarding the locality exhibited by each memory instructions is becoming a common practice. For instance, the PA7200 has a bit in order to identify memory instructions with only spatial locality [4]. The PowerPC provides the possibility of identifying instructions that do not exhibit much locality and thus, to bypass the cache for such instructions [16]. Having different cache memories specialized in exploiting different types of locality may be a promising alternative to increase the cache performance [14]. In all these cases, the compiler is responsible for providing the information that is codified in the memory instruction and that will determine during execution the proper action that the hardware must take.

In this paper we present detailed evaluation of the locality exhibited by the SPECfp95 benchmark suite, including the different issues listed above. To perform such

evaluation we have developed the SPLAT tool [15], which is an innovative tool that overcomes the limitations of previous ones to provide this type of information, as discussed below.

Current tools to analyze the locality of programs have important drawbacks that restrict the information that such tools can provide. Current data locality analysis tools can be classified into three categories, depending on the approach they use to perform the analysis:

- Memory simulators (e.g. [6][10]). An adequate memory simulator can provide all the information listed above. However, these tools are very slow, having an slowdown of several orders of magnitude. For instance, in the survey of Uhlig and Mudge [19], the slowdowns reported for trace-driven simulators are in the range of 45 to 6245. Besides, in most cases, these figures refer to simulators that just report miss ratios. These slowdowns are not affordable for some real applications that take several hours to run without any overhead. This is especially true if the analysis is just a step in an iterative process that consists of repetitive analysis and optimization steps. We believe that this iterative process is currently the most appropriate approach to optimize the locality exploitation of a given application since fully-automatic approaches are still not completely successful.

  Some recently proposed memory simulators (e.g. [9][12]) are claimed to have a much smaller slowdown. Their approach is based on eliminating/reducing the overhead for memory references that hit in cache. Therefore, the final overhead is dependent on the miss ratio. However, for typical miss ratios the resulting overhead can still be too high. Values in the range of 2-40 are reported in [19]. Besides, these overheads correspond to experiments where the user is interested just in the total miss ratio. If more complete information, like that listed above, is required the slowdown would be much higher. Besides, some of the information listed above (e.g. quantifying the intrinsic reuse) may require to instrument also the cache hits, which would result in a slowdown similar to that of trace-driven simulators. The implementation of special memory instructions to observe the behavior of the memory system is discussed in [8].
- Tools based on the hardware counters provided by many current microprocessors (e.g. [1]). These tools are fast but they can provide a very limited information, restricted by the hardware counters provided by the manufacturer. For instance, identifying conflicting data structures cannot be done relying just on hardware counters for any of the current microprocessors. Moreover, these tools can only analyze the locality exploited by the memory hierarchy of the actual microprocessor.
- Tools based on a static locality analysis (e.g. [18][7]). This approach is very fast since it has a negligible slowdown. However, it may be inaccurate for some programs due to the lack of information at compile time. For instance, loop bounds, initial addresses of data structures, size of array dimensions, etc. may be unknown at compile-time, which may significantly affect the accuracy of the analysis.

2. Overview of the SPLAT tool

In this section we present an overview of the SPLAT tool, which is the tool used to extract the statistics presented in the next section. For more details we refer the interested reader to [15].

The tool consists of three main steps. First it performs a reuse analysis based on the approach proposed by Wolf and Lam [20]. Then, some data unknown at compile-time (e.g. basic block counts) is obtained from a profiling of the program. Finally, the locality analyzer combines the reuse and profile information to provide the locality statistics. The tool can analyze the locality exploited by different cache architectures by repeating just the last step.

The tool is fast and accurate. The slowdown of the tool is about 0.05. The accuracy of the tool has been proved by comparing its results with that of a simulator [15]. The main limitation of the tool is that it is oriented just to numerical applications. For non-numerical applications the reuse analysis is likely to be rather inaccurate due to the abundant use of pointer references.

3. Locality of the SPECfp95

This section presents a quantitative analysis of the locality exhibited by the SPECfp95 programs. Due to current limitations of the compiler platform, we provide statistics for seven out of the ten benchmarks. Each program has been compiled with full optimizations and the reported statistics refer to the whole run of them.

3.1. Intrinsic reuse

The intrinsic reuse exhibited by a program can be used as a lower bound of the memory bandwidth required by a given program. That is, every reference that does not exhibit any type of reuse will surely require a memory reference to the next level of the memory hierarchy.

Memory reuse can be classified into four categories: self-temporal (ST), self-spatial (SS), group-temporal (GT) and group-spatial (GS). The temporal reuse is independent of the particular cache architecture of the underlying hard-

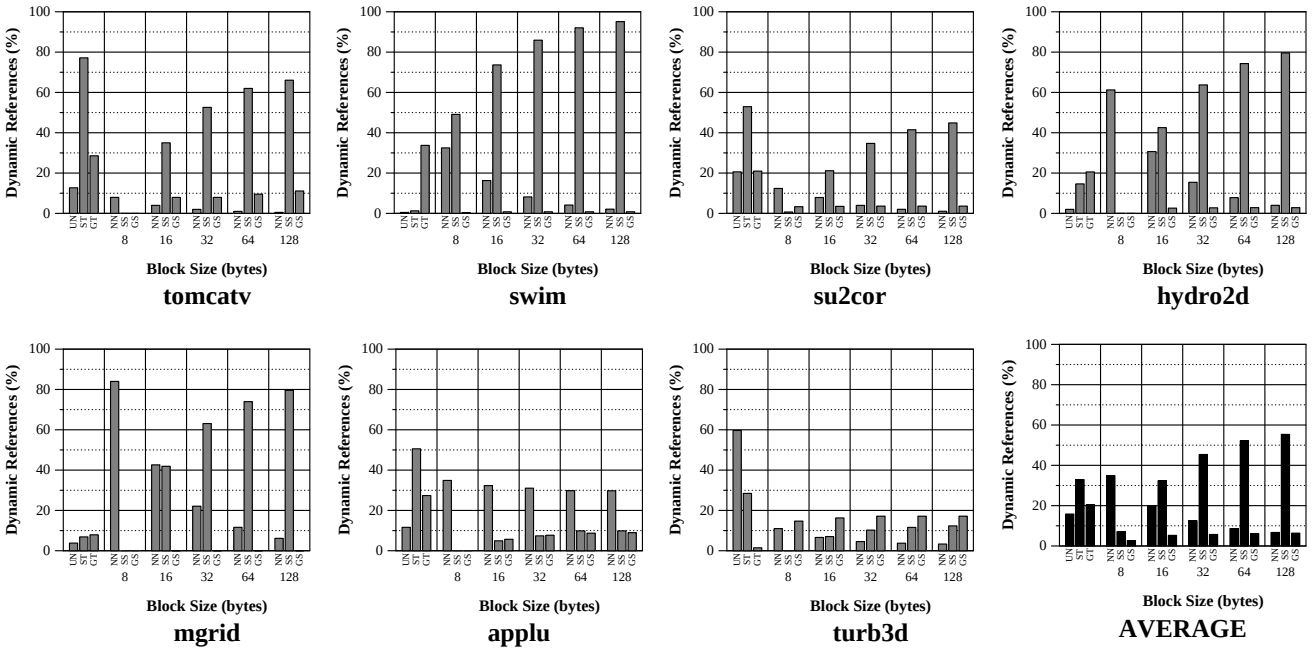

Figure 1. Intrinsic reuse

ware. On the other hand, spatial reuse just depends on the cache line size, in addition to the program characteristics. Regarding group reuse, currently the tool can only analyze the reuse among memory references that are in the same loop, which can result in an underestimation of group reuse. We are currently extending the tool to identify reuse among references in different loops.

Figure 1 quantifies the amount of reuse of the SPECfp95 for some of the programs and the average. The reuse is quantified for a cache line size ranging from 8 to 128 bytes. In addition to the previously mentioned four categories of reuse, the graphs include a fifth category that corresponds to those references without any type of reuse (NN) and a sixth one that corresponds to those references for which the tool has not been able to detect its type of reuse (UN). Notice that a given reference may exhibit several types of reuse and thus, the different bars may add up to more than 100%.

On the average for all programs, it can be seen that self-spatial reuse is the most frequent type of reuse (it is exhibited by 56% of all references). Self-temporal reuse is also significant (33% of references). Group-temporal is the next in importance (20% of references) and finally, group-spatial is the least common one (7% of references). Notice also that group-spatial reuse stabilizes for a 32-byte line size whereas self-spatial reuse provides diminishing returns for a line size greater that 128 bytes.

The results for individual programs are very different, and the dominant type/s of reuse depends on the concrete benchmark:

- ***Tomcatv***: the dominant types of reuse are both self-temporal and self-spatial. Group-temporal reuse is also important. In general, the intrinsic reuse for this program is very high (note that the NN bar is very small, even for blocks of 8 bytes).
- ***Swim***: for this program the dominant types of reuse are group-temporal and self-spatial. The other reuses are very low for all block sizes.
- ***Su2cor***: this benchmark presents a high degree of temporal reuse, mainly self-temporal. It is also a program with a high intrinsic reuse.
- ***Hydro2d***: the dominant type of reuse for this program is self-spatial, followed by temporal reuse (both self and group). Finally, group-spatial reuse is almost negligible.
- ***Mgrid***: for this program the temporal reuse is low, both self and group. Likewise, group-spatial reuse is almost null. The dominant type of reuse is self-spatial. Note that the no-reuse bar is very sensitive to the block size.
- ***Applu***: the most relevant aspect of this program regarding its reuse is the low impact of the block size. Note that the increment of spatial reuse or the decre-

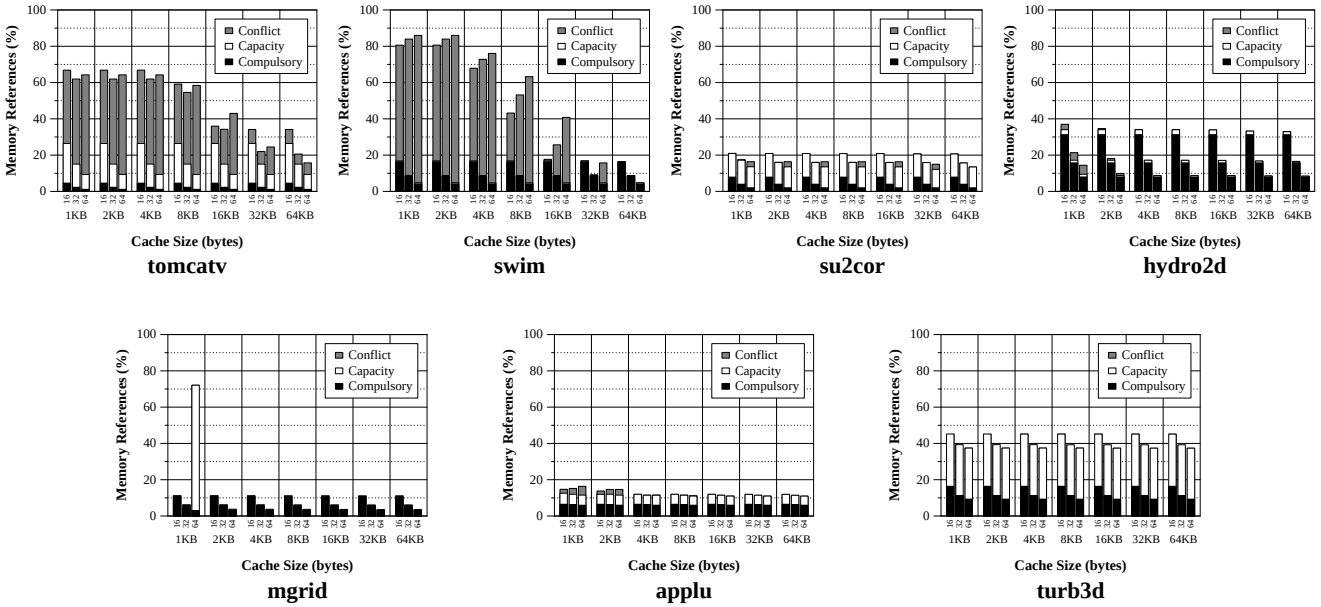

Figure 2. Different kinds of cache misses

ment of no-reuse is almost negligible for blocks bigger than 8 bytes. Besides, this is the program with the lowest amount of reuse, as denoted by the relatively high NN bar.

- ***Turb3d***: in general for this program the reuse is poor, mainly group-temporal and self-spatial reuse. However, note that the percentage on unknown references is very high (about 60%), so the results may not be very accurate for the overall program.

3.2. Quantifying different types of cache misses

Quantifying the different types of misses may be useful to decide the particular optimization that may best improve the performance of a give program. Misses are traditionally classified into three categories: compulsory, capacity and conflict. Each type of misses can be best reduced with different techniques as underlined in the introduction. Currently the tool can estimate conflict misses just for direct-mapped caches although the extension to set-associative caches is straightforward.

Figure 2 shows the miss ratio of the programs studied in this paper for a cache size ranging from 1 KB to 64 KB and a line size of 16, 32 and 64 bytes. For each configuration, the total miss ratio is divided into the three different categories. The y-axis represents the percentage of the total executed memory instructions whose reuse is known (the first column for each graphic in figure 1 - UN column - represents the dynamic percentage of references with unknown reuse).

For each program, the source of misses may be quite different:

- ***Tomcatv***: this program has a very large number of conflict misses, especially for caches smaller than 16 KB. Increasing the cache size reduces cache conflicts although other techniques like padding could be more cost-effective, as we will show in the next section. Capacity misses are also significant and hardly vary for the considered range of cache capacity. This is because the working set of this program is higher than 64 KB. Finally, notice that the most effective line size depends on the cache capacity. For very small caches, the line size has a small effect. For intermediate caches, smaller lines behave better since they significantly reduce the number of conflict misses, due to the larger number of lines. For large caches, the best performance is obtained by the largest line size. This is due to the reduction in capacity misses. Notice that increasing the line size may reduce capacity misses although it may seem counterintuitive. This may happen if there are references that exhibit both spatial and temporal reuse but temporal reuse cannot be exploited due to capacity constraints. In this situation, increasing the line size will result in a better exploitation of

spatial locality and thus, capacity misses will be reduced. For instance, assume the following code:

```
do i=1,8
   do j=1,1024
     ... A[j] ...
   enddo
enddo
```

In this example the reference *A[j]* has spatial reuse in loop *j* and temporal reuse in loop *i*. If the cache capacity is 512 elements, the temporal reuse cannot be exploited. Therefore, if the line size is 4 elements, this code will produce 256 (1024/4) compulsory misses (for the first iteration of loop *i*) and 1792 ((1024/4)∗7) capacity misses (the rest of iterations of loop *i*). However, if the line size is 8 elements, there will be 128 (1024/8) compulsory misses and 896 ((1024/8)∗7) capacity misses.

- ***Swim***: the main source of misses for this program is conflict misses. For cache size smaller than16Kbytes, misses due to interferences represent the majority of misses. For a cache of 16Kbytes it is also the dominant source of misses for block sizes of 32 and 64 bytes. Finally, for caches with a size of 32 or 64Kbytes, compulsory misses are the most important cause of miss. For instance, for caches of 64Kbytes, compulsory misses are the only source of misses.

  Note that for this program the effect of capacity misses is almost negligible.
- ***Su2cor***: the low impact of both cache and block sizes in the total number of misses for this program is remarkable. The behavior for all cache sizes is practically constant, and the impact of block size is less than 5%. It can be observed that the major part of misses are due to capacity misses, but the working set of the program is bigger than 64Kbytes, since capacity misses up to that size do not decrease.
- ***Hydro2d***: for this program all the working set can be stored in a very small cache and thus, increasing cache capacity hardly improves performance. Increasing the line size favors the exploitation of spatial locality and reduces the number of compulsory misses.
- ***Mgrid***: as *hydro2d*, for this program the main cause of misses are compulsory misses. If the cache has just 16 blocks (that is, a 1Kbyte cache with a block size of 64 bytes), there is no space to keep most of the data in cache, so the number of capacity misses is very high.
- ***Applu***: as *su2cor*, the number of misses for this benchmark is almost constant for all configurations.
- ***Turb3d***: this program shows a high miss ratio. However, note that the results are presented for references whose reuse could be studied, and for this program the percentage of unknown references is almost 60% (see figure 1). For this reason, the results of this graph m ay not be representative of the overall program.

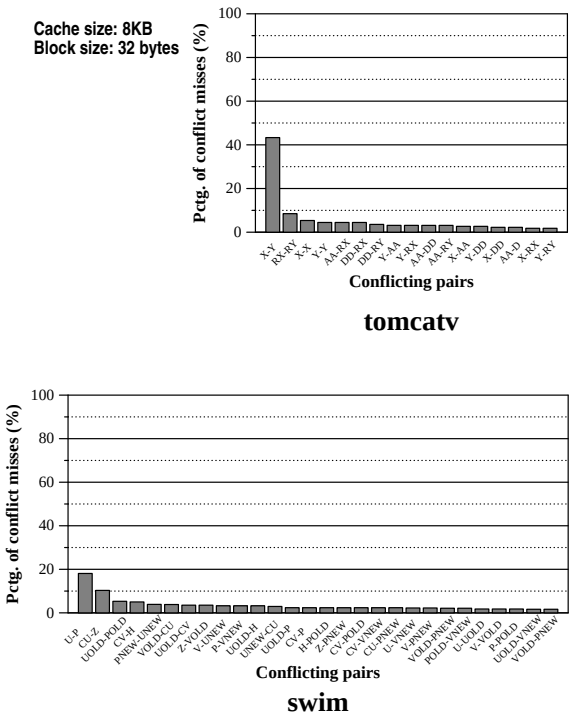

Figure 3. Percentage of conflict misses between data structures

The reuse information together with the quantification of the different types of reuse can be used by the compiler to set appropriately the hints provided by memory instructions in some microprocessors. For instance, if the cache has a bypass capability, those references without any reuse could be marked as non-cacheable. Besides, if two different memory instructions frequently collide, one of them could also be marked as non-cacheable. In this way, the locality exhibited by the other instruction could be exploited, which is better than not exploiting any of both. For instance, it has been reported that this type of analysis when applied to drive a selective caching policy may provide about 25% reduction in average memory access time and 65% reduction in next level memory bandwidth [14].

3.3. Conflicting data structures

For those programs with a high percentage of conflict misses, it may be interesting to identify which data struc-

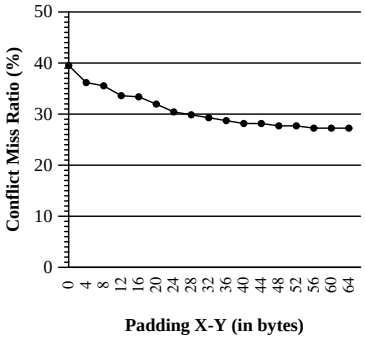

Figure 4. Reduction in conflict miss ratio after padding

tures are responsible for such conflicts. Techniques like padding or copying can be then applied to such data structures to try to reduce these conflicts.

For instance, figure 3 shows the percentage of conflicts between any pair of data structures, ordered from highest to lowest, for the *tomcatv* and the *swim* benchmarks, for a 8 KB direct-mapped cache with 32 bytes per line. For the *tomcatv* program, it can be seen that data structures X and Y are responsible for the majority of conflict misses. In addition, in figure 2 we can observe that most misses are due to conflicts, which suggest that padding may be an effective technique to reduce memory penalties. For instance, figure 4 shows the resulting conflict miss ratio of *tomcatv* after inserting a number of empty bytes between the two data structures. It can be seen that just with this naive padding scheme, conflict misses are significantly decreased, from 39.5% to 27.2%.

For the *swim* program, conflict misses are more distributed among a larger set of data structures.

3.4. Critical code sections

Most of the memory penalties are in many cases caused by a very small percentage of the code. Identifying these most penalizing sections may help the programmer/compiler to focus the effort on such parts of the code.

For instance, figure 5 shows the percentage of cache misses (over the total number of misses) that are caused by every innermost loop, for two applications. Besides, for each loop, its corresponding percentage of misses is split into the three different types: compulsory, capacity and conflict. An 8 KB, direct-mapped cache with 32 bytes per line is assumed.

Notice that in both cases the vast majority of misses are due to a very few sections of code: three innermost loops for *swim* and six innermost loops for *turb3d*. For

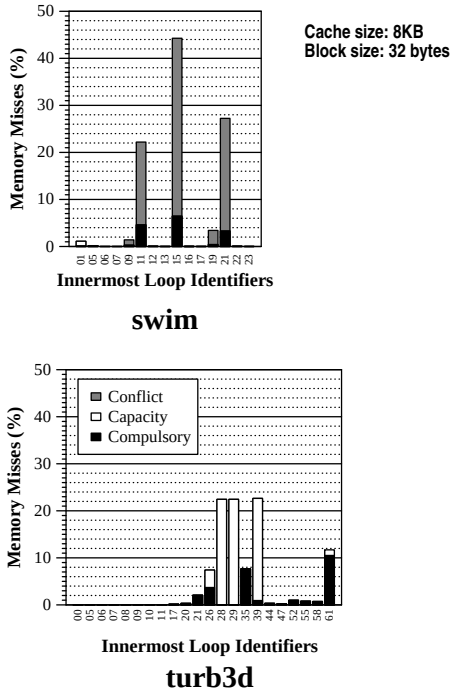

Figure 5. Cache misses per innermost loop

swim, most of the misses are due to conflicts whereas in *turb3d*, both capacity and compulsory misses have a significant contribution.

4. Conclusions

In this paper we have presented a detailed analysis of the locality exhibited by the SPECfp95 benchmark suite. This detailed evaluation has been performed by means of a new data locality analysis tool that is very fast, which allows to obtain statistics for the whole execution of real programs and many different cache configurations with a negligible slowdown.

We have shown that different programs exhibit very different locality characteristics. Detailed evaluation of the locality exhibited by a program may then be essential to choose the best approach to be taken to improve it.

Fully-automatic optimization tools have proved so far insufficient due to the variety of different scenarios that they should cope with. We then believe that the best approach today towards memory optimization is by means of an iterative (and interactive) process in which repetitive analysis and optimization steps are interleaved until the final result is acceptable. Therefore, the speed of the analysis tool as well as the range of information that it can provide are critical. We have shown that the type of analysis

presented in this paper can be very useful for such an approach.

Acknowledgments

This work has been supported by the Spanish Ministry of Education under contract CICYT-TIC 429/95, the ESPRIT Project MHAOTEU (EP24942), and by the Catalan CIRIT under grant 1996FI-3083-APDT.

References

[1] G. Ammons, T. Ball and J.R. Larus, "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling", in *Procs. on of the 1997 Conf. on Programming Languages Design and Implementation (PLDI-97), 1997*

[2] D. Callahan, K. Kennedy and A. Porterfield, "Software Prefetching", in *Procs. of IV Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91)*, pp. 40-52, April 1991

[3] S. Carr, K.S. McKinley and C-W. Tseng, "Compiler Optimizations for Improving Data Locality", in *Procs. of V Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*, pp. 252-262, Oct. 1994

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

[5] D. Gannon, W. Jalby and K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformations", Journal of Parallel and Distributed Computing, 5, pp. 587-616, 1988

[6] J. Gee, M. Hill, D. Pnevmatikatos and A.J. Smith, "Cache Performance of the SPEC92 Benchmark Suite", *IEEE Micro*, pp. 17-27, Aug. 1993

[7] S. Ghosh, M. Martonosi and S. Malik, "Cache Miss Equations: an Analytical Representation of Cache Misses", in *Procs. of Int. Conf. on Supercomputing (ICS'97)*, pp. 317-324, July 1997

[8] M. Horowitz, M. Martonosi, T.C. Mowry and M.D. Smith, "Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors", in *Procs. of Int. Symp. on Computer Architecture (ISCA'96)*, pp. 260-270, 1996

[9] M. Martonosi, A. Gupta and T. Anderson, "Memspy: Analyzing Memory Performance System Bottlenecks in Programs", *Performance Evaluation Rev.*, 20(2), June 1992

[10] K. S. McKinley and O. Temam, "A Quantitative Analysis of Loop Nest Locality", in *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*, 1996

[11] T.C. Mowry, M.S. Lam and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching", in *Procs. of V Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'92)*, pp. 62-73, Oct. 1992

[12] S. Reinhardt, R. Pfile and D. Wood, "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers", in *ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, pp. 48-60, 1993

[13] G. Rivera and C-W. Tseng, "Data Transformations for Eliminating Conflict Misses" in *Procs. of Conf. on Programming Language Design and Implementation (PLDI'98)*, 1998

[14] F.J. Sánchez, A. González and M.Valero, "Static Locality Analysis for Cache Management", in *Procs. Int. Conf. on Parallel Architectures and Compiler Techniques (PACT'97)*, 1997

[15] F.J. Sánchez and A.González, "Fast, Accurate and Flexible Data Locality Analysis", *Technical Report DAC-UPC-1998-8, UPC-Barcelona*, May 1998

[16] J.M. Stone and R.P. Fitzgerald, "Storage in the PowerPC", *IEEE Micro*, vol. 15, no. 2, pp. 50-58, April 1995

[17] O. Temam, E.D. Granston, W. Jalby, "To Copy or not to Copy: A Compile-time Technique for Assessing when Data Copying Should be Used to Eliminate Cache Conflicts", in *Procs. of Supercomputing'93 Conf. (SC'93)*, pp. 410-419, 1993

[18] O. Temam, C. Fricker and W. Jalby, "Cache Interference Phenomena", in *ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, May 1994

[19] R.A. Uhlig and T.N. Mudge, "Trace-driven Memory Simulation: a Survey", *ACM Computing Surveys*, 1997

[20] M.E. Wolf and M.S. Lam, "A Data Locality Optimizing Algorithm", in *Procs. of Conf. on Programming Language Design and Implementation (PLDI'91)*, pp. 30-44, 1991