Analyzing Data Locality in Numeric Applications

SPLAT provides programmers a fast and accurate study of memory behavior without the necessity of a costly memory simulator. The tool is suitable for use as a step in an iterative optimization process in time-consuming numeric applications.

Jesús Sánchez
Antonio González
Polytechnic University of Catalonia, Barcelona

Memory performance is becoming a major bottleneck in current microprocessors. A great deal of research has aimed at developing techniques for improving memory performance. Some of these techniques rely on hardware alone, but many require programmer or compiler support. Examples of the latter are software prefetching, blocking, and copying. To use these techniques effectively, the programmer must have some knowledge of the program's behavior. For instance, prefetching is useful only if it is limited to instructions that frequently produce cache misses. Adding a prefetch instruction to every memory instruction could result in significant performance degradation.

These techniques might also require quantification of the different types of cache misses (see sidebar, page 60). For instance, microprocessors can avoid compulsory misses through both hardware and software prefetching.[1] Blocking, or tiling, is a method of avoiding capacity misses; copying and padding are techniques for reducing the effect of conflict misses.[1]

Many processors provide hints in their memory instructions that the compiler can use for optimizing memory performance. Examples of such hints are the PowerPC's cache bypass facility and the hints incorporated by the IA-64 instruction set. Effective use of these hints requires information about the program's locality behavior.

The process of obtaining information about a program's locality characteristics is data locality analysis. Traditionally, this analysis takes place either at compile time or at runtime.[2,3] The former approach incurs low overhead but is relatively inaccurate because the compiler lacks some information. The runtime approach usually takes the form of a memory hierarchy simulation, which is quite accurate but very slow.

In this article, we introduce SPLAT (Static and Profiled Data Locality Analysis Tool). The tool's purpose is to provide a fast study of memory behavior without the necessity of a costly memory simulator. SPLAT consists of a static locality analysis enhanced by simple profiling data. Its overhead is low because it performs most of the analysis at compile time, and because the required profiling support is just a basic-block-execution count. Many commercial compilers support this profiling option. Compared with simulation techniques, SPLAT's estimation technique is highly accurate for numeric codes.

The tool is useful not only for compilers but also for programmers. To tune a program, programmers should know its performance, the

0272-1732/00/$10.00 © 2000 IEEE

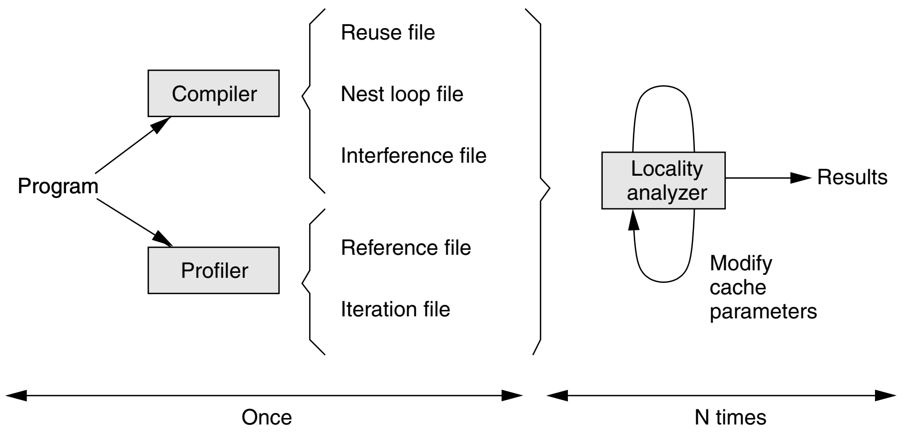

Figure 1. SPLAT's global analysis scheme.

critical parts that produce most memory penalties, the sources of these penalties, and the data structures responsible for most cache misses. SPLAT provides this type of information.[4]

Overview

SPLAT performs locality analysis using static information computed by the compiler and dynamic data obtained through simple profiling. Figure 1 diagrams SPLAT's global analysis scheme.

The locality analyzer uses static information to identify the types of misses that will occur during execution of the program. To identify compulsory misses, the compiler must compute the intrinsic reuse of data. For capacity misses, it must also compute the volume of data referenced by each loop iteration. Finally, to identify conflict misses, the compiler computes interferences among data references. The compiler reports all this information to three files:

- *Reuse file.* For each memory instruction and each loop in which it is enclosed, this file stores the type of reuse (unknown, none, self-temporal, self-spatial, group-temporal, or group-spatial). If the reuse is spatial, the file also stores the stride (the difference between the effective address of two dynamic memory references that result in spatial reuse). If the reuse is group-temporal or group-spatial, the file also contains the distance (the number of iterations before the reuse takes place). The compiler derives this reuse information from the reuse vectors proposed by Wolf and Lam.[5]
- *Nest loop file.* This file represents the program's loop structure. The file stores each loop's parent loop.
- *Interference file.* This file contains the initial addresses of each pair of static memory instructions (with the same nesting level and with no other loop between them) that have the same reference pattern, if the addresses are known at compile time. (In the seven SPECfp95 benchmarks studied in this article, the initial addresses and dimension sizes are known at compile time for about 75% of all dynamic memory instructions.) Two instructions have the same reference pattern if 1) their corresponding variables have the same number of dimensions, 2) each dimension's size is the same in both references, and 3) the expressions representing the indexing functions for each dimension differ only in a constant value.

In the programs studied in this article (a subset of the SPECfp95 benchmark suite), most of the loops have unknown bounds since they depend on the data input set. Moreover, each memory instruction's number of executions depends not only on the number of iterations of the loops enclosing it, but also on conditional statements, which are difficult to analyze at compile time. In SPLAT, therefore, we use a profiler to quantify loop bounds and instruction count.

The profiling consists of the number of executions of each basic block, a facility provid-

Key concepts

The following memory-related terms are important in this article:

Reuse and locality

A reuse occurs whenever a memory instruction references the same data as another instruction (either the same static instruction or another one). However, when a processor executes these instructions, some factors may inhibit the exploitation of this reuse in a given memory hierarchy level (for instance, the cache memory's limited storage). Reuse (also called *intrinsic reuse*) is a measure inherent in a program, and it depends on neither the instruction execution order nor the cache configuration. We call the amount of reuse actually exploited by a memory hierarchy level the *locality* of the program with respect to that memory level.

Wolf and Lam define the types of reuse and locality we talk about in this article.[1] *Temporal reuse* occurs when one or more instructions access the same memory location several times. It is *self-temporal* or *group-temporal* depending on whether the same static instruction or different instructions access the memory location. On the other hand, *spatial reuse* occurs when different nearby memory locations are accessed. It is *self-spatial* if the same static instruction accesses the locations and *group-spatial* if different instructions access the locations. A static instruction's reuse and locality with respect to a loop of the loop nest that encloses it are the reuse and locality exhibited by the instruction's dynamic instances corresponding to the loop's various iterations. An instruction in a loop nest can have a different type of reuse and locality for each loop.

Cache misses

Cache misses traditionally fall into three categories: *compulsory, capacity,* and *conflict*.[2] Compulsory misses (also called *cold-start misses*) occur the first time a cache block is accessed. In contrast, both capacity and conflict misses are replacement misses—in other words, the data in the cache was replaced before the current access. Capacity misses occur because the cache cannot contain all the blocks needed during a program's execution. Conflict misses occur when the mapping function maps too many blocks to the same set.

References

1. M.E. Wolf and M.S. Lam, "A Data Locality Optimizing Algorithm," *Proc. Conf. Programming Language Design and Implementation (PLDI)*, ACM Press, New York, 1991, pp. 30-44.
2. M.D. Hill, *Aspects of Cache Memory and Instruction Buffer Performance*, PhD thesis, UCB/CSD 87/381, Univ. of California at Berkeley, Nov. 1987.

ed by many current compilers (for example, the Sun f77). From this information, the profiler derives the number of each memory instruction's executions and the average number of each loop's iterations, which are stored in the reference file and the iteration file. This dynamic information and the static information in the reuse, nest loop, and interference files are the input to the locality analyzer.

Locality analysis

The locality analysis consists of three phases: reuse, volume, and interference. The first phase identifies all the reuse exhibited by the program. This information is the basis for the rest of the analysis. However, identifying compulsory misses requires no additional analysis; compulsory misses consist of all references without any reuse. For instance, for an instruction with only self-temporal reuse, the first reference is a compulsory miss, whereas the rest are hits.

The volume phase identifies capacity misses. Finally, the interference phase computes conflict misses.

The core of the locality analysis is the function qreuse (Figure 2), which is used by all three phases. This function quantifies the locality of each memory instruction for the various types of reuse.

First, SPLAT applies qreuse to a program, using as an input the static reuse information. All references that cannot exploit any type of reuse will cause a cache miss (a compulsory miss), and thus a new block will be brought into the cache. In this way, the analyzer computes the number of blocks brought into the cache by each loop, which is referred to as loop volume.

Then, whenever the program references a volume of data greater than the cache size between two accesses to the same block, the analysis registers a nonexploitable reuse. The number of nonexploitable reuses due to capacity constraints reflects the number of capacity misses.

Finally, the analyzer identifies instructions that cause self-interferences due to their strides, and instructions that cause interferences with other instructions in the same loop. The lost reuses resulting from these causes represent the number of conflict misses.

The algorithm for the qreuse function proceeds as follows: In each phase, this function performs the locality analysis of each memory instruction (from 0 to NMINSTR) except those with unknown reuse. (The latter correspond to references outside loops, array references inside loops with nonlinear expressions in any of their dimensions, or references with expressions that contain variables that are not loop indices. References with unknown reuse are assumed always to miss in cache. They represent 15% of the total memory references in the analyzed programs.) The analysis starts from the innermost loop (denoted $N-1$) and ends with the outermost loop that includes each instruction i (denoted 0).

Using the reuse vectors, qreuse computes the following values for each memory instruction i in loop j:

- GIt_j (group reuse iterations in loop j)—the number of iterations with group reuse in loop j.
- $NGIt_j$ (no group reuse iterations in loop j)—the number of iterations without group reuse in loop j.
- TIt_j (total iterations in loop j)—the average number of iterations of loop j in which instruction i is executed.
- $ATIt_j$ (accumulated total iterations in loop j)—the number of executions of the memory instruction per each iteration of loop j. It is computed as

$$\prod_{i=j+1}^{N-1} TIt_i$$

The quantification of each reuse type for each loop enclosing the reference is stored in vectors NN (no reuse), ST (self-temporal), SS (self-spatial), GT (group-temporal), and GS (group-spatial). For instance, $ST_i[j]$ represents the number of executions of instruction i that exhibit self-temporal reuse considering all iterations of loop j. The qreuse function quantifies each type of intrinsic reuse identified by the compiler as follows (see Figure 2):

```
function qreuse () {
  for i=0 to NMINSTR do {
    NNi[N] = 1;
    STi[N] = SSi[N] = GTi[N] = GSi[N] = 0;
    for j=N-1 to 0 do {
      Compute (GItj, NGItj, TItj, ATItj);
      switch (SELFReuse[j]) {
        case NONE:
          NNi[j] = NGItj * NNi[j+1];
 A        STi[j] = TItj * STi[j+1];
          SSi[j] = TItj * SSi[j+1];
        break;
        case TEMPORAL:
          NNi[j] = NNi[j+1];
 B        STi[j] = (TItj - 1)*ATItj + STi[j+1];
          SSi[j] = TItj * SSi[j+1];
        break;
        case SPATIAL:
          factor = stride / blocksize;
          NNi[j] = (factor *NGItj) * NNi[j+1];
 C        STi[j] = TItj * STi[j+1];
          SSi[j] = (factor * TItj) * SSi[j+1] +
                  ((1-factor) * TItj) * ATItj;
        break;
      }
      GTi[j] = NGItj * GTi[j+1];
      GSi[j] = NGItj * GSi[j+1];
      switch (GROUPReuse[j]) {
        case NONE:
        break;
        case TEMPORAL:
 D        GTi[j] += GItj * ATItj;
        break;
        case SPATIAL:
          GSi[j] += GItj * ATItj;
        break;
      }
    }
  }
}
```

Figure 2. Algorithm for quantifying intrinsic reuse: qreuse.

- *Section A.* The instruction does not have either kind of self-reuse in loop j. In this case, for each iteration of j without group reuse, the number of executions without any reuse is the number of executions without reuse in the loop $j+1$ (that is, $NN_i[j] = NGIt_j * NN_i[j+1]$). For each iteration of loop j, the number of executions with self-temporal or self-spatial reuse is the number of executions with such reuse in loop $j+1$ (for example, $ST_i[j] = TIt_j * ST_i[j+1]$).

```
DO i = 1, N
  A(i)
  DO j = 1, M
    B(j, i)
    C(i)
    B(j, i+2)
  ENDDO
  D
ENDDO
```

	Loop j								Loop i							
	ST	SS	GT	GS	GIt	NGt	TIt	ATIt	ST	SS	GT	GS	GIt	NGt	TIt	ATIt
A(i)									✗	✓(1)	✗	✗	0	N	N	1
B(j, i)	✗	✓(1)	✗	✗	0	M	M	1	✗	✗	✓(2)	✗	N−2	2	N	M
C(i)	✓	✗	✗	✗	0	M	M	1	✗	✓(1)	✗	✗	0	N	N	M
B(j, i+2)	✗	✓(1)	✗	✗	0	M	M	1	✗	✗	✗	✗	0	N	N	M
D									✓	✗	✗	✗	0	N	N	1

✗	Does not exhibit this reuse
✓	Exhibits this reuse
(1)	Stride
(2)	Distance
M and N	Loop bounds

(a)

	Loop j					Loop i				
	NN	ST	SS	GT	GS	NN	ST	SS	GT	GS
A(i)						N/4	0	3N/4	0	0
B(j, i)	M/4	0	3M/4	0	0	M/2	0	3NM/4	(N–2)M	0
C(i)	1	M–1	0	0	0	N/4	N(M–1)	3NM/4	0	0
B(j, i+2)	M4	0	3M/4	0	0	NM/4	0	3NM/4	0	0
D						1	N–1	0	0	0

(b)

Figure 3. Quantifying reuse: two nested loops with reuse types and parameter values (a); qreuse analysis of the two loops (b).

- *Section B.* The instruction has self-temporal reuse in loop *j*. In this case, the first iteration of loop *j* has the same number of no reuses as the whole execution of loop $j + 1$. The executions corresponding to the remaining iterations reuse the data of the first iteration. Therefore, $NN_i[j] = NN_i[j + 1]$. All executions except the first iteration exploit self-temporal reuse. For this iteration, the number of self-temporal reuses corresponds to that exhibited by the next inner loop. Self-spatial reuse is computed as in section A.
- *Section C.* The instruction has self-spatial reuse in loop *j*. In this case, qreuse computes a value called a factor, which represents the percentage of references that access a new cache block. Then, for each iteration of *j* without group reuse that references a new cache block, the number of executions without reuse is the number of executions without reuse in loop $j + 1$. Self-temporal reuse is computed as in section A. Finally, the algorithm computes self-spatial reuse as follows: For iterations of *j* such that *i* references a new block, the number of self-spatial reuses is the same as the number in the next inner loop. For the remaining iterations, all the executions exhibit self-spatial reuse.
- *Section D.* The algorithm computes group reuse (spatial and temporal) as follows: First, for iterations of *j* such that *i* does not exhibit group reuse, the number of executions with group reuse is the same as that of the next inner loop. For the remaining iterations, all executions exhibit group reuse.

Reuse phase. The input to this phase is the reuse file computed at compile time, and this phase applies the qreuse function using this reuse information. After computation of qreuse, $NN_i[0]$ contains the number of compulsory misses of instruction *i*.

Figure 3a shows an example of code composed of two nested loops. The left-hand parts of the tables show the type of reuse exhibited by each memory instruction in each loop. If the instruction exhibits self-spatial reuse, the stride is also shown. If it exhibits group reuse, the distance is also listed. The right-hand part of each table shows the values of the parameters defined earlier.

Figure 3b is an example of how the function qreuse works. The tables quantify the types of reuse for each reference in each loop in Figure 3a, as computed by qreuse. Looking at the table corresponding to loop *i*, we can see the reuse exhibited by each reference for the whole loop nest, since loop *i* includes loop *j*. For instance, among the NM executions of reference $B(j,i)$, we see that $3NM/4$ exhibit self-spatial reuse, $(N-2)M$ exhibit group-temporal reuse, and $M/2$ exhibit no reuse. Note that the number of reuses totals more than NM because a particular dynamic instruction can exhibit more than one type.

Volume phase. A factor that can inhibit the exploitation of reuse is the cache memory's limited storage. That is, if the number of data blocks referenced between two consecutive reuses of the same block exceeds the cache capacity in block units, an LRU (least recently used) fully associative cache cannot exploit this reuse. The result is a capacity miss.

This phase computes the volume (in cache blocks) that each memory instruction contributes to the total volume of the loops that enclose it. The analyzer obtains this value directly from the data computed in the reuse phase. For a given loop *j*, each execution of instruction *i* that does not exhibit any type of reuse will bring a new block into the cache. On the other hand, if a particular execution of an instruction has any type of reuse, it does not bring any additional data into the cache. Therefore, the value of $NN_i[j]$ is the volume contributed by instruction *i* to loop *j*.

After computing the volume of every loop, the analyzer marks some reuses nonexploitable:

- If an instruction has self-reuse in loop *j* (temporal or spatial), but the volume of loop *j* is greater than the total number of cache blocks, this reuse will likely not be exploited by a conventional cache.
- If an instruction has group reuse (temporal or spatial), and the volume corresponding to the loop's distance iterations is greater than the total number of cache blocks, this reuse will likely not be exploited.

Next, SPLAT computes the function qreuse again, but without considering the nonexploitable reuses. The newly computed $NN_i[0]$, as in the previous phase, represents the cache misses of instruction *i*, and the difference from its previous value is the number of capacity misses of instruction *i*.

Interference phase. Conflict misses can have a high impact on cache memories with a low degree of associativity, especially direct-mapped caches. These misses are hard to identify because they depend on various dynamic factors such as each data structure's initial memory address and the instruction order. The interference phase identifies conflict misses by finding interferences among data references.

Interferences are of two types: self-interferences and cross-interferences. Self-interferences occur when different data blocks referenced by the same static instruction are mapped onto the same cache location. Cross-interferences occur among different static instructions. SPLAT detects a subset of these interferences and focuses on direct-mapped caches. (Alternatively, we have also developed a more accurate, but more complex, interference analysis for set-associative caches.[6] This approach uses the cache miss equations[2] and some efficient techniques to reduce the complexity of the analysis. Nevertheless, as we will show, the simplified analysis presented in this article is quite accurate for the evaluated programs.)

For every array reference and every loop for which the reference does not exhibit temporal locality, the analysis assumes that self-interferences occur if the following condition is met:

$$\mathit{cache\_size\_in\_blocks} < N * 2^{\mathit{stride\_family\_in\_blocks}}$$

N represents the number of iterations of the loop. The *stride_family_in_blocks* is derived from the stride of the reference in the analyzed loop, measured in cache block units. If the stride is not an integral number of blocks, the stride is rounded up to the next integer. The stride_family identified as x is the set of strides $s * 2^x$ such that s is any odd number. All strides belonging to the same family (for example, $12 = 3 * 2^2$ and $20 = 5 * 2^2$ belong to family 2) have the same self-interference behavior.

For each reference and each loop, SPLAT computes a self-conflict ratio, which denotes the percentage of the N iterations of the loop that produce self-interferences. The amount of reuses in outer loops decreases by this factor due to self-interferences.

Regarding cross-interferences, we focus on those usually called ping-pong interferences. Two static instructions cause ping-pong interferences if they reference different data blocks that map onto the same cache block for every execution. These interferences will completely inhibit the exploitation of any reuse exhibited by the interfering instructions. This type of conflict is analyzed for each pair of memory instructions that meet the following conditions:

- Variables whose base address and size of every dimension is statically known—that is, variables allocated at compile time.
- Both references follow the same pattern.
- The difference, or "hole," between the addresses of the first element referenced by both instructions (addresses R_A and R_B) modulo the cache size is less than the cache block size. That is, there is no chance of interference if the two references do not map onto the same cache block:

$$hole_{AB} = | R_A \bmod cache\_size - R_B \bmod cache\_size |$$
$$hole_{AB} < block\_size$$

For each instruction that meets these conditions, a real value between 0 and 1 that represents the percentage of interference (PI) is defined. If PI is 0, the instruction is free of interferences. If PI is 1, this instruction conflicts with another instruction for every loop iteration. Values between 0 and 1 represent different percentages of interference—that is, the percentage of total iterations in which an instruction causes a cache miss due to interferences. For two instructions A and B that interfere, SPLAT computes this factor as follows:

$$PI_{AB} = (block\_size - hole_{AB}) \mid block\_size$$

We derive this expression from the fact that the probability of interference grows as the difference between the addresses ($hole_{AB}$) decreases. When the two instructions always reference the same location (that is, $hole_{AB} = 0$), they interfere for every execution. If an instruction conflicts with various other instructions, the analysis considers the maximum PI.

The reuse of an instruction i in a loop that is not marked nonexploitable in the volume phase will be exploited only by the percentage of references that are free of interferences. That is, the number of reuses computed in the previous phase ($ST_i[0]$, $SS_i[0]$, $GT_i[0]$, and $GS_i[0]$) are multiplied by $(1 - PI_i)$. The rest of the references will produce a cache miss.

To summarize, the interference phase applies the self-conflict ratio and the PI factor to the locality vectors ($ST_i[0]$, $SS_i[0]$, $GT_i[0]$, and $GS_i[0]$). The accumulated difference between the previous values of the elements of these vectors and the current ones is the number of conflict misses.

Performance

SPLAT would be useless if its results were inaccurate or if obtaining them was too costly. We validated SPLAT's accuracy by comparing its estimated miss ratios with those obtained through a cache simulator. We also found that the tool's overhead is almost negligible.

Framework

We implemented SPLAT's static analysis using the Ictineo compiling platform with full optimizations. We used the following programs from the SPECfp95 benchmark suite: tomcatv, swim, su2cor, hydro2d, mgrid, applu, and turb3d. Our study considered a direct-mapped cache. The results represent the profiling and execution of each program, using the "train" input set for profiling and the "test" input set for simulations. This method took the effect of different input data sets into account. See our technical report for more details and additional performance results.[4]

Accuracy

We simulated a direct-mapped cache memory of different capacities (1, 8, and 64 Kbytes) and block sizes (16, 32, and 64 bytes). Figure 4 shows the results for three benchmark programs. Two of them (tomcatv and swim) show high variability in the miss ratio; the other (hydro2d) has a miss ratio much less affected by the cache parameters. Also, tomcatv and swim have a high conflict miss ratio, whereas hydro2d has a low conflict miss ratio.

The graphs show the simulated and estimated cache miss ratios for the various cache configurations. SPLAT's results are very close to the simulation results, showing that the tool is accurate for a typical range of cache parameters. We obtained similar results for the remaining benchmarks.

Another way to measure the estimate's accuracy is to compute the average absolute error per instruction. This error indicates how far from reality the estimate is for each instruction. We compute the dynamic average error per instruction as

$$avg\_derror = \frac{\sum_{i}^{NINSTR} \left| missratio_{est_i} - missratio_{sim_i} \right| * nrefs_i}{\sum_{i}^{NINSTR} nrefs_i}$$

where $missratio_{est}$ represents the estimated miss ratio of a particular memory instruction, and $missratio_{sim}$ represents the miss ratio obtained by simulation. The dynamic average error per instruction is around or less than 10% for all programs and all cache configurations. The impact of references with unknown reuse is very low, since normally these instructions are outside of loops and are rarely executed.

We also studied the estimated error for a particular configuration (in this case, an 8-Kbyte cache with 32-byte blocks). The results show that a large percentage of dynamic instructions have a low error rate. The tool's accuracy is extremely high for the hydro2d program; about 90% of the instructions had no error at all.

Slowdown

SPLAT's overhead consists of three parts (corresponding to the components in Figure 1):

- *Compiling.* The static reuse analysis is a new pass of our compiler platform. The time required for this phase is similar to the time required by any other pass of our compiler. The tool must perform this step only once per program.
- *Profiling.* The slowdown of a simple basic-block-count profiling ranges from 0.0 to 0.1 on a SuperSparc/60 workstation (a 0.1 slowdown means that the program takes 10% more time due to the profiling). This step must be performed once per program and data input set.
- *Locality analysis.* The time needed to execute this phase (for a particular set of cache parameters) is no more than a few

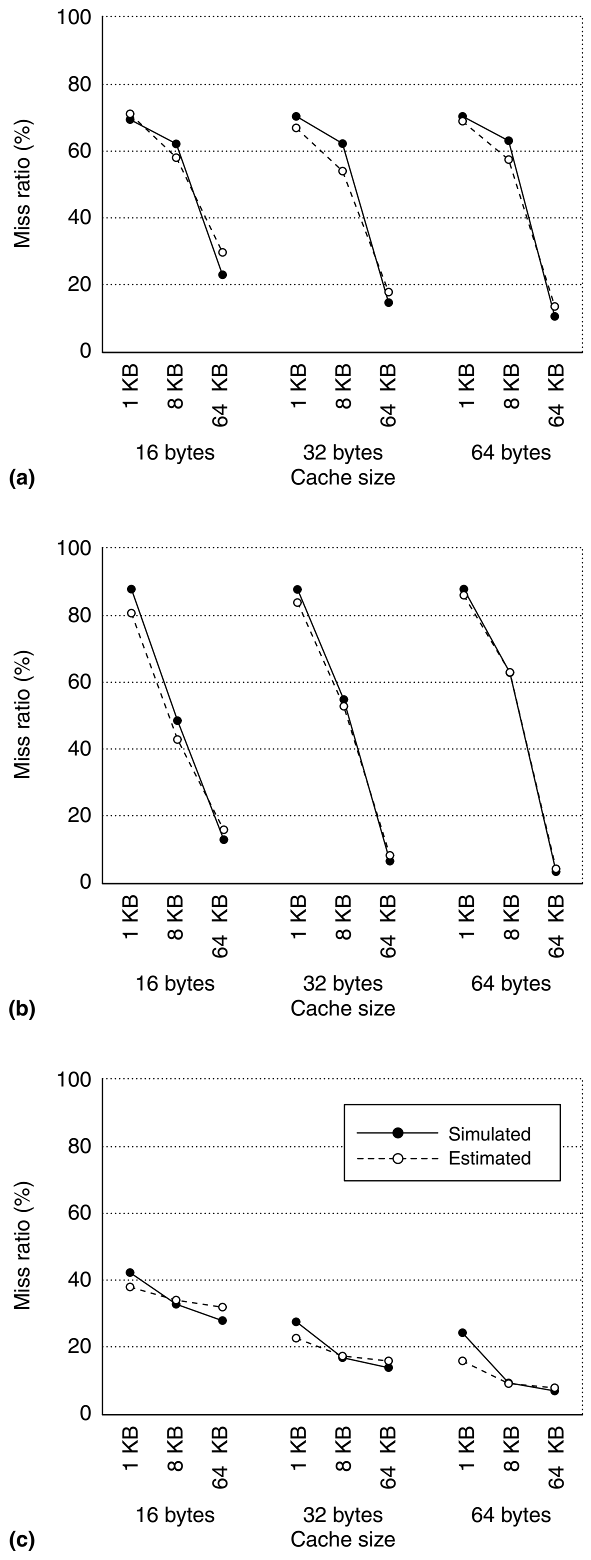

Figure 4. Comparison of SPLAT's results with simulation results for three SPECfp95 programs: tomcatv (a), swim (b), and hydro2d (c).

seconds, and the tool spends most of that time reading the data files.

Overall, SPLAT's overhead is almost negligible. In addition, the tool can analyze multiple cache configurations with about the same overhead as one, since only the locality analysis must be repeated.

So far, fully automatic optimization tools have proved insufficient to handle the variety of scenarios they must cope with. The best approach to memory optimization appears to be an iterative and interactive process, interleaving repetitive analysis and optimization steps until the final result is acceptable. The type of analysis presented here can be very useful in such an approach. The speed of the analysis tool and the range of information it provides are critical. Moreover, tools like SPLAT can take advantage of the hints included in instruction set architectures to make efficient use of the memory hierarchy.

We have successfully used SPLAT and an extended version with a more powerful interference analysis to solve problems such as managing a multimodule cache[7] and performing variable padding.[8] We are currently using the extended tool to improve the instruction scheduler for a distributed cache memory architecture. MICRO

Acknowledgments

This work was supported by the Spanish Ministry of Education under contract CICYT-TIC 98-511, the ESPRIT Project MHAOTEU (EP24942), and the Catalan CIRIT under grant 1996FI-3083-APDT.

References

1. D. Bacon, S. Graham, and O. Sharp, *Compiler Transformations for High-Performance Computing*, Tech. Report UCB.CSD-93-781, Univ. of California, Berkeley, 1993.
2. S. Ghosh, M. Martonosi, and S. Malik, "Cache Miss Equations: An Analytical Representation of Cache Misses," *Proc. Int'l Conf. Supercomputing (ICS 97)*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1997, pp. 317-324.
3. R.A. Uhlig and T.N. Mudge, "Trace-Driven Memory Simulation: A Survey," *ACM Computing Surveys*, Vol. 29, No. 2, June 1997, pp. 128-170.
4. J. Sánchez and A. González, *SPLAT: A Static and Profiled Data Locality Analysis Tool for Numeric Applications*, Tech. Report UPC-DAC-1999-68, Dept. of Computer Architecture, Universitat Politècnica de Catalunya, Barcelona, 1999; http://www.ac.upc.es.
5. M.E. Wolf and M.S. Lam, "A Data Locality Optimizing Algorithm," *Proc. Conf. Programming Language Design and Implementation (PLDI 91)*, ACM Press, New York, 1991, pp. 30-44.
6. X. Vera et al., *A Fast Implementation of Cache Miss Equations*, Tech. Report UPC-DAC-1999-50, Dept. of Computer Architecture, Universitat Politècnica de Catalunya, Barcelona, Nov. 1999; http://www.ac.upc.es.
7. J. Sánchez and A. González, "A Locality Sensitive Multi-Module Cache with Explicit Management," *Proc. Int'l Conf. Supercomputing (ICS 99)*, IEEE CS Press, 1999, pp. 51-59.
8. X. Vera, A. González, and J. Llosa, *Near-Optimal Padding for Removing Inter-Variable Conflict Misses*, Tech. Report UPC-DAC-2000-30, Dept. of Computer Architecture, Universitat Politècnica de Catalunya, Barcelona, 2000; http://www.ac.upc.es.

Jesús Sánchez is an assistant professor and a PhD candidate in the Department of Computer Architecture of the Polytechnic University of Catalonia, Barcelona. His research interests focus on computer architecture and compilers. Sánchez received an MS in computer science from the Polytechnic University of Catalonia.

Antonio González is an associate professor in the Computer Architecture Department of the Polytechnic University of Catalonia. His research interests center on computer architecture, compilers, and parallel processing. González received an undergraduate degree in computer science and a PhD in computer science, both from the Polytechnic University of Catalonia. He is a member of the IEEE Computer Society and the ACM.

Send comments to Jesús Sánchez and Antonio González, Dept. of Computer Architecture, Universitat Politècnica de Catalunya, Barcelona, Spain; [fran, antonio]@ac.upc.es.