Fast, Accurate and Flexible Data Locality Analysis

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 tool based on a new approach for analyzing the locality exhibited by data memory references. The tool is very fast because it is based on a static locality analysis enhanced with very simple profiling information, which results in a negligible slowdown. This feature allows the tool to be used for highly time-consuming applications and to include it as a step in a typical iterative analysis-optimization process. The tool can provide a detailed evaluation of the reuse exhibited by a program, quantifying and qualifying the different types of misses either globally or detailed by program sections, data structures, memory instructions, etc. The accuracy of the tool is validated by comparing its results with those provided by a simulator.

1. Introduction

Memory performance is becoming an important bottleneck in current microprocessors. A huge research effort has been devoted to propose novel techniques to improve its performance. Some of these techniques are just hardware-oriented but there are many of them that require some support of the programmer/compiler. This type of techniques usually require some knowledge of the behavior of the program.

For instance, prefetching is useful if it is only performed for instructions that produce cache misses. Adding a prefetch instruction to every memory instruction may result in significant performance degradation. These techniques may also require the quantification of the different types of cache misses, traditionally noted as compulsory, capacity and conflict misses. For instance, compulsory misses can be avoided through prefetching, both hardware [2] and software [6]. Blocking or tiling is used to avoid capacity misses [3], whereas some examples of techniques to reduce the effect of conflict misses are copying [11] and padding [7].

There are many processors that provide some type of hints in their memory instructions that the compiler can use to optimize the memory performance. Examples of such hints are the *spatial locality only* hint in the PA7200 [4], the cache bypass facility provided by the PowerPC [10] and the Dual Data Cache [8]. An effective use of such hints requires again information about the program locality behavior.

The process of obtaining information of the locality characteristics of a given program is known as *data locality analysis*. This analysis has been performed traditionally either at compile-time or at run-time. The former approach has a low overhead but it is relatively inaccurate since there is much information that the compiler does not know. The latter usually takes the form of a memory hierarchy simulator, which is quite accurate but very slow.

In this paper we propose a novel data locality analysis mechanism that consists of a static locality analysis enhanced with very simple profiling data. The overhead of this mechanism is very low since most of the analysis is performed at compile-time and the required profiling support is just a basic block execution count. This profiling option is currently supported by many commercial compilers. Besides, we show in this paper that the proposed mechanism is highly accurate for numeric codes by comparing it with techniques based on simulation.

This tool is useful not only for the compiler but also for programmers. In order to tune a program, a programmer may be interested in first knowing its performance, locating those critical parts where most of the memory penalties are produced, identifying which data structures are responsible for most of the cache misses, etc. The tool presented here provides these functionalities, but due to paper length constrains we do not show any of them in this work. We refer the interested reader to [9] for detailed examples of its functionality. This information can be used to optimize the performance by means of different techniques such as padding, bypassing, blocking, prefetching, etc.

The rest of this paper is organized as follows. Section 2 presents the locality analysis tool. The accuracy of the tool is validated in section 3. Finally, the main conclusions are summarized in section 4.

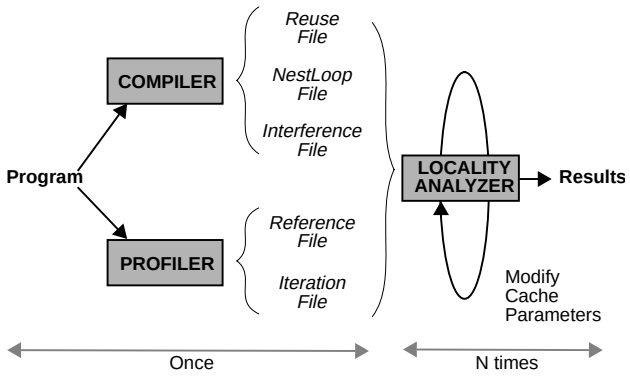

Figure 1. Global scheme

2. The SPLAT locality analysis tool

This section describes the proposed tool for data locality analysis, which is called SPLAT (Static-Profiled data Locality Analysis Tool).

The locality analysis is performed through some static information computed by the compiler and some dynamic information obtained by a simple profiling (see Figure 1).

The static information is aimed at computing the different types of misses that will happen during the execution. Compulsory misses require to compute the intrinsic reuse of data. Capacity misses require in addition to compute the volume of data referenced by each loop iteration. Finally, conflict misses are identified by computing interferences among data references. All this information is summarized in three files:

- ***Reuse file***: for each memory instruction and each loop in which it is enclosed, it stores its type of reuse (unknown, none, self-temporal, self-spatial, group-temporal or group-spatial). If the reuse is spatial it also stores the stride (i.e., the difference between the effective address of two consecutive executions). If the reuse is group (both temporal and spatial) it also contains the *distance*, which is defined as the number of iterations before the reuse takes place.
- ***Nest loop file***: this file is intended to represent the loop structure of the program. For each loop it stores its parent, which is defined as the loop that encloses it.
- ***Interference file***: for each pair of memory instructions (with the same nesting level and without any other loop in between) that have the same reference pattern, it contains their initial addresses if they are known at compile-time[1]. Two instructions have the same reference pattern if their corresponding variables have the same dimensions, and the expressions that represent the indexing functions for each dimension differ only in a constant value.

The profiling consists of just the number of executions of each basic block, which is a facility provided by many current compilers (e.g. the Sun f77 compiler). We have measured that the slowdown of this step in the execution of a program is negligible (less than 2%). From the information obtained by the profiler, the number of executions of each memory instruction and the average number of iterations of each loop can be derived (we have measured that for the programs analyzed in this work - see section 3.1. - at least one bound of every loop is unknown at compile time). These data are stored in the ***reference file*** and the ***iteration file*** respectively. Notice that for a particular input data the profile step has to be executed just once. However, a new execution of the profiler is needed if the input data is changed.

This static and dynamic information is used as an input to the locality analyzer. The locality analysis is divided into three phases: (i) reuse phase, (ii) volume phase, and (iii) interference phase. The first phase identifies all the reuse exhibited by the program. This information is the basis for computing misses. In particular, compulsory misses do not require any additional analysis: they consist of all references without any reuse. The volume phase is targeted to identify capacity misses. Finally, the interference phase computes the conflict misses.

2.1. Reuse phase

Reuse is a measure that is inherent in a given program and depends on neither the order in which instructions are later executed nor the cache capacity. In this phase, the different types of reuse exhibited by each reference are quantified.

The input to this phase is the *reuse file* that is computed at compile-time following the methodology described by Wolf and Lam in [12].

The quantification of the reuse is performed basically through the function `qreuse(i)` showed in Figure 2, which is applied to each memory instruction except for those with unknown reuse[2] (they correspond to references outside loops, or inside loops but with non-linear expressions, or expressions with variables that are not loop indices). The `i` parameter represents the instruction identifier. The analysis starts from the innermost loop and finishes with the outermost loop that includes the instruction `i`, which are denoted by `N-1` and 0 respectively.

1. In the SPECfp95 benchmark suite, about 75% of all memory references have their initial address and dimension sizes known at compile-time

2. References with unknown reuse are assumed to always miss in cache. They represent a 15% of the total number of memory references in the SPECfp95.

```
function qreuse (int i) {
 NN_i[N] = 1;
 ST_i[N] = SS_i[N] = GT_i[N] = GS_i[N] = 0;
 for j=N-1 to 0 do {
  switch (SELFReuse[j]) {
    case NONE:                                          A
      NN_i[j] = NGIt_j * NN_i[j+1];
      ST_i[j] = TIt_j * ST_i[j+1];
      SS_i[j] = TIt_j * SS_i[j+1];
    break;
    case TEMPORAL:                                      B
      NN_i[j] = NN_i[j+1];
      ST_i[j] = (TIt_j - 1)* ATIt_j + ST_i[j+1];
      SS_i[j] = TIt_j * SS_i[j+1];
    break;
    case SPATIAL:                                       C
      factor = stride / blocksize;
      NN_i[j] = (factor * NGIt_j) * NN_i[j+1];
      ST_i[j] = TIt_j * ST_i[j+1];
      SS_i[j] =(factor * TIt_j) * SS_i[j+1] +
               ((1-factor) * TIt_j) * ATIt_j;
    break;
  }
  GT_i[j] = NGIt_j * GT_i[j+1];                         D
  GS_i[j] = NGIt_j * GS_i[j+1];
  switch (GROUPReuse[j]) {
     case NONE:
     break;
     case TEMPORAL:
       GT_i[j] += GIt_j * ATIt_j;
     break;
     case SPATIAL:
       GS_i[j] += GIt_j * ATIt_j;
     break;
  }
 }
}
```

Figure 2. Algorithm to quantify intrinsic reuse

The function computes for each particular memory instruction in a particular loop *j* the following values:

- GIt_j: number of iterations with group reuse in loop j.
- $NGIt_j$: number of iterations without group reuse in loop *j*.
- TIt_j: total number of iterations of loop j.
- $ATIt_j$: number of executions per iteration of loop *j*. It is computed as $\prod_{i=j+1}^{N-1} TIt_i$.

The quantification of each type of reuse for each loop in which the reference is enclosed is stored in the 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 the iterations of loop *j*. Each type of intrinsic reuse identified by the compiler is quantified as follows (see Figure 2),

- *Section A*: the instruction does not have any 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* (i.e., $NN_i[j]=NGIt_j*NN_i[j+1]$). 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 (i.e., $ST_i[j]=TIt_j*ST_i[j+1]$).
- *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* and the executions corresponding to the remaining iterations reuse the data of the first iteration. Therefore $NN_i[j]=NN_i[j+1]$. Self-temporal reuse is exploited by all executions except for the first iteration. For this iteration, the number of self-temporal reuses corresponds to that exhibited by the next inner loop. Finally, self-spatial reuse is computed as in section A.
- *Section C*: the instruction has self-spatial reuse in loop *j*. In this case, a value called *factor* that represents the percentage of references that access a new cache block is computed. Then, for each iteration of *j* without group reuse that references a new cache line, the number of executions without any reuse is the number of executions without reuse in the loop *j+1*. Self-temporal reuse is computed as in section A. Finally, self-spatial reuse is computed as follows. For those iterations of *j* such that *i* references a new block, the number of self-spatial reuses are the same as those in the next inner loop; and for the remaining iterations, all the executions exhibit self-spatial reuse.
- *Section D*: group reuse is computed as follows (spatial and temporal are treated in the same way). First, for those 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.

After computing the function `qreuse(i)`, $NN_i[0]$ contains the number of *compulsory misses* of instruction `i`.

2.2. Volume phase

A factor that may inhibit the exploitation of reuse is the limited storage of cache memory. In other words, if the amount of different data blocks that are referenced between two consecutive reuses of the same block is higher than the cache capacity (in block units), this reuse cannot be exploited by an LRU fully-associative cache. The resulting cache miss is called a *capacity miss*.

In this phase, the volume (in cache blocks) that each memory instruction contributes to the total volume of the loops that enclose it is computed. This can be obtained directly from the data computed in the previous 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 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 cache. Therefore, the value of $NN_i[j]$ expresses the volume contributed by the instruction *i* to the loop *j*.
Once the volume of every loop has been computed, some reuses are marked as non-exploitable:

- If an instruction has self reuse in loop *j* (either 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 (either temporal or spatial) and the volume corresponding to *distance* (see beginning of section 2) iterations of the loop is greater than the total number of cache blocks, this reuse will likely not be exploited either.

Then, the function `qreuse` is computed again but without considering the reuses marked as non-exploitable. The new computed $NN_i[0]$, as in the previous phase, represents the cache misses of instruction *i* and the difference with its previous value is the number of *capacity misses* of instruction *i*.

2.3. Interference phase

Another factor that influences the locality is the effect of interferences. Typically, interferences or *conflict misses* are defined as those misses that occur in a direct-mapped or *n*-way set-associative cache but not in a fully-associative cache. This kind of misses may have a high impact for cache memories with a low degree of associativity, specially for direct-mapped caches.

The behavior of conflict misses is hard to predict because it depends on various dynamic factors such as memory addresses, instruction order, etc. Interferences may be of two different types: *self-interferences* and *cross-interferences*. Self-interferences occur when different data blocks referenced by the same instruction are mapped onto the same cache location, whereas cross-interferences occur among different memory instructions. The analysis proposed in this paper detects a subset of these interferences. The interference analysis is currently implemented for direct-mapped caches. Its extension to set-associative caches is an ongoing task.

For every array reference and every loop for which it does not exhibit temporal locality, self-interferences are assumed to occur if the following condition is met:

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

where *N* represents the number of iterations of the loop. The *stride_family_in_blocks* is related to the stride of the reference in the analyzed loop, expressed 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* defined by *x* is the set of strides $\sigma \cdot 2^x$ with σ odd [5]. All the strides belonging to the same family (e.g., $12=3 \cdot 2^2$ and $20=5 \cdot 2^2$ belong to family 2) have the same behavior from the point of view of self-interference.

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

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

a) Variables whose base address and size of every dimension is statically known (75% of all references as previously reported).
b) The difference or "hole" between the addresses (modulo the cache size) of the first element referenced by both instructions is less than the cache block size.

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

c) Both references follow the same pattern (see the above description of the *interference file* for a definition of reference pattern).

For each instruction, a real value between 0 and 1 that represents the *percentage of interference* (*PI*) is defined. If *PI* is 0, this instruction is free of interferences whereas if *PI* is 1, it means that this instruction conflicts with some other instruction for every iteration of the loop. Values in between represent different percentages of interference, that is, the percentage of total iterations in which an instruction misses due to interferences. For two instructions A and B that interfere, this factor is computed as:

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

If an instruction conflicts with various other instructions, the maximum *PI* is considered.

The reuse of an instruction *i* in a loop that is not marked as non-exploitable in the volume phase will be exploited only by the percentage of references that are free of interferences, that is, for $(1-PI_i) * nrefs_i$, where $nrefs_i$ is the number of executions of instruction *i*. The rest of references will produce a cache miss.

Then, the function `qreuse` is computed again but considering just the reuses that are free of interferences. The new computed $NN_i[0]$, as in previous phases, represents the number of cache misses of instruction *i* and the difference with its previous value is the number of *conflict misses*.

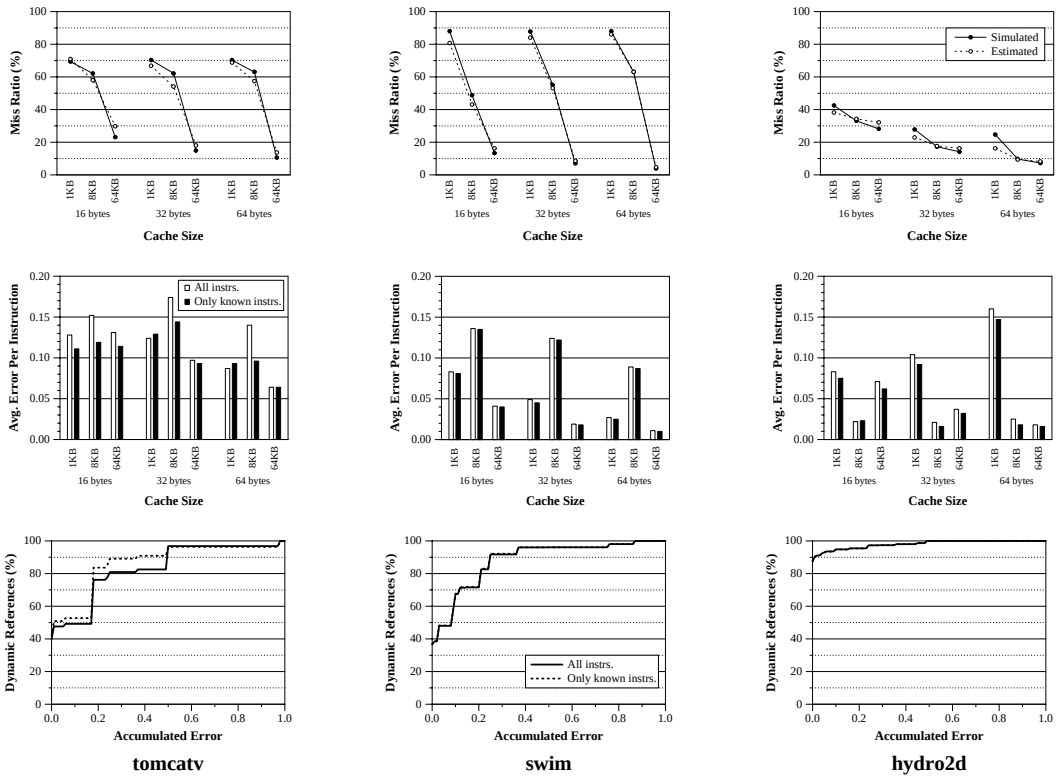

Figure 3. Comparison of the tool results against simulation results

3. Validation

The SPLAT tool estimates the data locality exhibited by a program through some information computed at compile-time and some simple dynamic information obtained by a profiler. The aim of this tool is a fast study of the memory behavior without the necessity of a costly memory simulator. However, this tool would be useless if the obtained results were far from the reality. In this section, we validate the accuracy of the proposed tool by comparing the estimated miss ratios with those obtained through a cache simulator.

3.1. Framework

The static analysis used by the SPLAT tool has been implemented using the ICTINEO compiling platform [1]. Currently, ICTINEO assumes an infinite number of registers and thus, the references produced by spill code are not considered in this work. Optimizations usually applied by current compilers (such as common subexpression elimination, deadcode removal, invariants, etc.) are implemented and are applied to the resulting code. In this way, the resulting code is very similar to the code generated by a production compiler.

For the tool validation we have used some programs from the SPECfp95 benchmarks suite. These programs are: *tomcatv, swim,* and *hydro2d*.

A direct-mapped cache has always been considered. The results presented in this paper correspond to the profiling/execution of the whole execution of each benchmark using the test input data.

3.2. Error in the estimation

In order to validate the tool, the results obtained by simulation and the results produced by the SPLAT tool have been compared. With this goal, we have simulated a direct-mapped cache memory of different capacities (1KB, 8KB and 64KBytes) and various block sizes (16, 32 and 64 bytes). Figure 3 shows the results for the analyzed programs, two of them showing a high variability in the miss ratio (*tomcatv* and *swim*), whereas the other one has a miss ratio that is much less affected by the cache parameters (*hydro2d*). Besides, *tomcatv* and *swim* are programs with a high conflict miss ratio whereas *hydro2d* has a very low conflict miss ratio.

The first row of graphs shows both the simulated and estimated cache miss ratios for the various configurations of cache. We can see in these graphs that the results obtained by the SPLAT tool are very close to the simulation results. That shows that the tool is accurate for a typical range of cache parameters.

Another way to measure the accuracy of the estimation is to compute the average absolute error per instruction. This error indicates how far from the reality the estimation is for each single instruction. These results are depicted in the second and third rows of graphs.

The second row shows the dynamic average error per instruction, which is computed as:

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

Black bars represent the error for instructions with reuse known at compile time for different cache and line sizes, whereas white bars correspond to the error considering all the instructions. We can see in these graphs that the impact of references with unknown reuse is very low, since normally these instructions are out of loops and rarely executed. The average error per dynamic instruction is around or less than 10% for all programs and all the cache configurations.

Finally, on the third row of graphs the estimated error for a particular configuration (in this case, a 8Kb cache with 32 bytes blocks) is more detailed. These graphs display the distribution function of the dynamic error per instruction. It can be seen that a very large percentage of dynamic instructions have a very low error. The accuracy of the tool is extremely high for the *hydro2d* program: about 90% of the instructions have no error at all.

4. Conclusions

We have presented a new data locality analysis methodology. So far this type of analysis has been performed by means of: a) simulators, which have a high slowdown and thus, they are unfeasible for some large commercial applications; b) hardware-counters, whose scope is quite limited to the actual processor architecture and the provided counters; and c) just a static analysis, which may be inaccurate because of the limited knowledge of the compiler. The proposed tool uses a static locality analysis combined with very simple profiling information, which results in a very fast tool and very accurate at the same time, as shown in the paper.

The tool is flexible in the sense that it provides many useful utilities to the programmer/compiler: cache miss ratio, quantification of the reuse, identifying critical parts of the program, determining the data structures that produce most conflict misses, etc. (see [9]).

Tools like the one proposed in this paper will be very useful to take advantage of the hints that some current microprocessors are offering to the programmer/compiler for an efficient use of the memory hierarchy.

Aknowledgments

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] E. Ayguadé, C. Barrado, A. González, J. Labarta, D. López, S. Moreno, D. Padua, F. Reig, Q. Riera and M. Valero, "Ictineo: a Tool for Research on ILP", in *SC'96, Research Exhibit "Polaris at Work"*, 1996

[2] J-L. Baer and T-F. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty", in *Procs. of ICS'97*, pp. 176-186, Nov. 1991

[3] S. Carr, K.S. McKinley and C-W. Tseng, "Compiler Optimizations for Improving Data Locality", in *Procs. of the V ASPLOS*, 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.T. Harper III and D.A. Linebarger, "A Dynamic Storage Scheme for Conflict Free Vector Access", in *Procs. of the 14th ISCA*, pp. 72-77, 1987

[6] T.C. Mowry, M.S. Lam and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching", in *Procs. of the V ASPLOS'92*, pp. 62-73, Oct. 1992

[7] G. Rivera and C-W. Tseng, "Data Transformations for Eliminating Conflict Misses" in *Procs. of PLDI'98*, 1998

[8] Jesús Sánchez, Antonio González and Mateo Valero, "Static Locality Analysis for Cache Managements", in *Procs. of PACT'97*, pp. 261-271, Nov. 1997

[9] Jesús Sánchez and Antonio González, "Data Locality Analysis of the SPECfp95", in *Digest of the 3rd. Workshop on PAID'98 (in conjunction with ISCA'98)*, June 1998

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

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

[12] M.E. Wolf and M.S. Lam, "A Data Locality Optimizing Algorithm", in *Procs. of PLDI'91*, pp. 30-44, 1991