Exploiting Non-Uniform Reuse for Cache Optimization: A Case Study

Claudia Leopold
Friedrich-Schiller-Universität Jena
Institut für Informatik
07740 Jena, Germany
claudia@informatik.uni-jena.de

Keywords: Data locality, caching, program restructuring

ABSTRACT

Due to the growing gap between processor speeds and memory speeds, cache optimizations have an increasing impact on sequential and parallel program performance. Existing techniques such as loop tiling focus on reuse between uniformly generated references, that is, between array accesses whose index expressions differ in a constant term only. In this paper, we show that the exploitation of *non*-uniform reuse can be worthwhile as well. We introduce two novel program restructuring techniques called *folding* and *snaking*, and study their performance impact on an exemplary loop nest. Folding achieves a speedup of up to 2.5, for this example, and the combined speedup of folding and tiling is up to 5.8.

1. INTRODUCTION

As has been widely observed, the performance of programs strongly depends on their ability to use caches efficiently. The importance of cache consciousness grows as the gap between processor speeds and memory speeds is getting larger. Furthermore, in shared- memory parallel machines, the existence of multiple processors increases the memory load.

To avoid program slowdown by memory access latency, data should be reused multiple times after having been loaded into cache. This can be achieved by program restructuring. Besides other factors, it is in particular the locality of a program whose improvement leads to speedups. *Locality*, or more specifically data locality, denotes the degree of concentration of the accesses to the same cache line during program execution.

Several locality optimization techniques have been proposed [1, 12, 17, 18], part of which are used in commercial compilers. Examples of techniques include:

- Loop permutation – Changing the order of loops in a nest
- Loop tiling – Blocking the loop iterations as explained later, and
- Array transpose – Changing the storage order of matrices, for instance from row-major to column-major.

Most of the existing transformations refer to loop nests. As a common characteristic, these transformations exploit either self-reuse (a single program statement accesses the same cache line multiple times), or reuse between *uniformly generated references* [5, 17]. A uniformly generated reference is a pair of array accesses whose index functions differ in a constant term only. For example, in the loop nest

```
for (i=2; i<N; i++)
   for (j=2; j<N; j++)
      A[i] = A[i+1] + A[i-2] +A[j]
```

the references $A[i]$, $A[i+1]$, and $A[i-2]$ are uniformly generated, whereas the references $A[i]$ and $A[j]$ are not.

Wolf and Lam [17] state that "little exploitable reuse exists between non-uniformly generated references", and thereby refer to a specific class of transformations (unimodular transformations and tiling). In this paper, we reconsider this statement, taking other transformations into account. We show that if a program exhibits non-uniform reuse, then its exploitation can improve the performance significantly. Therefore we study an example program and investigate in-depth the performance impact of various optimizations. We introduce a novel transformation called *folding* that exploits non-uniform reuse. In our example, folding achieves speedups of 1.14 to 2.5. It is thus comparable to tiling, the transformation that has previously been recommended for the example. Tiling and folding can be combined and then achieve speedups of up to 5.8.

Folding has been invented with the help of a locality optimization tool called IBLOpt. This tool suggests a second transformation, *snaking*, but the performance impact of snaking is minor. For both transformations, we present performance numbers measured on a Pentium PC and a DEC Alpha workstation. Furthermore we investigate the interplay between the novel transformations and standard compiler techniques. We refer to *gcc* and the Portland Group C compiler. In addition to running times, we measure cache misses and other metrics through the processor-internal hardware performance counters.

SAC 2001, Las Vegas, NV

Section 2 of this paper introduces the example program, outlines the IBLOpt tool, and explains the suggested transformations folding and snaking. Section 3 describes the experimental setting and then lists and discusses our experimental results. Section 4 surveys related work, and Section 5 finishes with conclusions.

2. FOLDING AND SNAKING

Our example program is given in Fig. 1. It has been adapted from Wolf and Lam [17], where it was used as an illustrative example for tiling. In [17], f was not specified, but assumed to be any simple function that accesses its arguments. We set $f(a, b) = a + b$ since a concrete function is needed for the experiments. Furthermore we have inserted initialization and printout statements to prevent the compiler from applying unwanted optimizations, in particular dead code elimination.

```
float f(float a, float b) return a+b;
main() {
/* Declarations and Initializations */
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        h+=f(A[i],A[j]);
/* Printout */
}
```

Figure 1: Input program

The example was input into the IBLOpt locality optimization tool [9, 10]. This tool produces suggestions for an improved program structure by optimizing one or several small instances of a given program. It unrolls the loops of the instance into a sequence of elementary assignment statements, and reorders the assignments such that the locality is improved and data dependencies are respected. To estimate the quality of the modified programs, the tool refers to a virtual cache whose capacity must be less than the number of inputs. In our case, the small instance was formed by setting $N = 6$, cache capacity$= 4$, and cache line size$= 2$.

Before we discuss the suggested transformations, let us first look at Fig. 2a), which visualizes the input program. Note that the program refers to a one-dimensional array, but both axes are indexed with this array, and thus a two-dimensional structure arises. Position (i, j) in the figure stands for $f(A[i], A[j])$. The arrows indicate the order of executing the function calls.

Figure 2b) illustrates *folding*. For all i, j, the function calls $f(A[i], A[j])$ and $f(A[j], A[i])$ access the same array elements, and thus locality is improved by executing the calls of each pair immediately after each other. Visually, this transformation corresponds to folding the lower left triangle of the figure over the upper right one. As indicated by the arrows, the execution order is thus

$$(0,0), (0,1), (1,0), (0,2), (2,0) \ldots$$

Figure 2c) shows *snaking*. This transformation improves locality since cache line ($A[4], A[5]$) is loaded into cache in the first row of Fig. 2c), and reused in the second.

Folding and snaking can be combined with *tiling*. This transformation is well-known, and is illustrated in Fig. 2d). Combining folding, snaking, and tiling, we obtain the program in Fig. 2e).

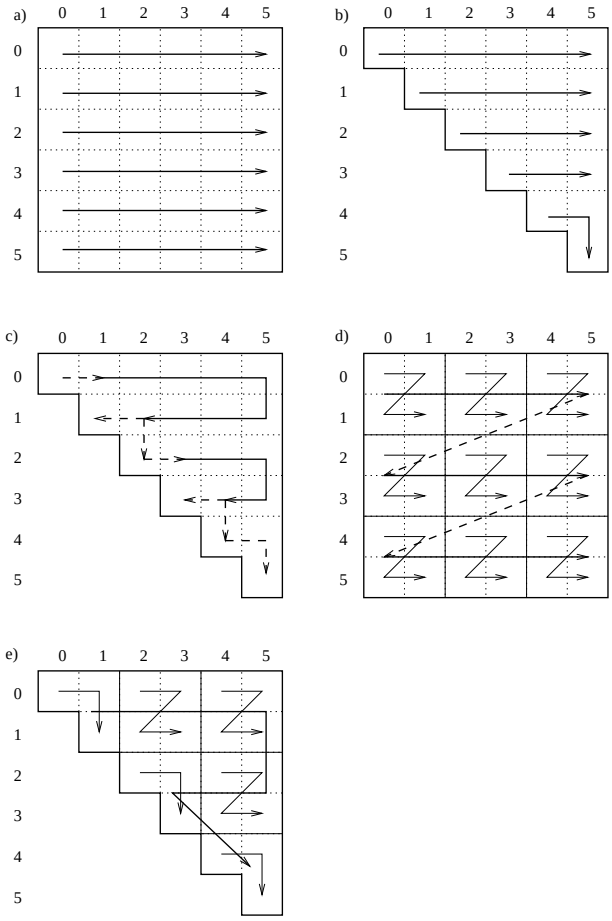

Figure 2: Program Variants: a) input, b) fold, c) foldSnake, d) tile, e) foldSnakeTile

Figure 2e) is the output of the IBLOpt tool. To state it in more detail, IBLOpt outputs a sequence of statement instances and a structured representation thereof that correspond to the execution order in Fig. 2e) (except for minor differences). It reports a number of simulated cache misses of 14 for the program in Fig. 2a), 8 for Fig. 2d), and 4 for Fig. 2e). From the IBLOpt output, the codes in Fig. 3 have been derived manually. We use the abbreviations *fold* if only folding is applied to the input program, *tileFold* if tiling and folding are applied, and so on.

Folding exploits reuse between non-uniformly generated references $A[i]$ and $A[j]$. According to the classification scheme of Wolf and Lam [17], this reuse is group-temporal, since different array accesses reuse the same data element. Snaking, in contrast, exploits self-temporal reuse, since the data element is reused by the same array access.

Although we refer to a single example program in this paper, folding and snaking are more general. Snaking can be combined with tiling in many loop programs, e.g. in matrix multiplication, but only occasionaly improves the performance. Folding can be generalized to loop nests in which two or more non-consecutive loop iterations access the same data, provided that the accesses occur at symmetric positions in the program, and that the transformation is not prevented by data dependencies. Symmetry is required since

```
a)  for  (jj=0;jj<N;jj+=s)
      for  (i=0;i<N;i++)
        for  (j=jj;j<jj+s     &&  j<N;j++)
          h+=f(A[i],A[j]);

b)  for  (i=0;i<N;i++)
      h+=f(A[i],A[i]);
      for  (j=i+1;j<N;j++)
        h+=f(A[i],A[j]);
        h+=f(A[j],A[i]);

c)  for  (jj=0;jj<N;jj+=s)
      for  (j=jj;j<jj+s     &&  j<N;j++)
        for  (i=jj;i<jj+s     &&  i<N;i++)
          h+=f(A[j],A[i]);
      for  (i=jj+s;i<N;i++)
        for  (j=jj;j<jj+s     &&  j<N;j++)
          h+=f(A[j],A[i]);
          h+=f(A[i],A[j]);

d)  for  (jj=0;jj<=N-2*s;jj+=2*s)
      for  (j=jj;j<jj+s;j++)
        for  (i=jj;i<jj+s;i++)
          h+=f(A[j],A[i]);
      for  (i=jj+s;i<N;i++)
        for  (j=jj;j<jj+s;j++)
          h+=f(A[j],A[i]);
          h+=f(A[i],A[j]);
      for  (i=N-1;i>=jj+2*s;i--)
        for  (j=jj+s;j<jj+2*s;j++)
          h+=f(A[j],A[i]);
          h+=f(A[i],A[j]);
      for  (j=jj+s;j<jj+2*s;j++)
        for  (i=jj+s;i<jj+2*s;i++)
          h+=f(A[j],A[i]);
  for  (j=jj;j<N;j++)
    for  (i=jj;i<N;i++)
      h+=f(A[j],A[i]);
```

Figure 3: Program variants: a) tile, b) fold, c) tileFold, d) tileFoldSnake (braces are omitted for brevity)

the accesses of each pair must be brought together without overly complicating the program structure.

3. EXPERIMENTAL EVALUATION

We experimentally compared the running time and other performance metrics of the program variants *input*, *tile*, *fold*, *tileFold*, and *tileFoldSnake*. Here, *input* refers to the input program from Fig. 1, and the other program variants are the same as in Fig. 3, except that declarations, initialization, and printout were added. N was set to 30000. With this value, the running time is manageable, and matrix A fits neither into L1 cache nor into L2 cache (for the D-variants). In preliminary experiments with $N = 300000$, comparable results were obtained.

The entries of array A are either type *float*, or a structure type that comprises seven floating point numbers. In the former case, the program variants are marked with a final F, and in the latter with a final D: inputF, tileF, inputD, and so on. The D-variants reduce the amount of spatial reuse.

We ran the experiments on two machines: a Pentium PC and a DEC Alpha workstation. The PC has a 266 MHz Pentium II processor with 16 KByte L1 data cache, 16 KByte L1 instruction cache, and 512 KByte L2 mixed data/instruction cache. For both L1 and L2, the cache line size is 32 Byte (= 8 floating point numbers), and the caches are 4-way set-associative.

The DEC Alpha workstation has a 500 MHz Alpha 21164 processor with 8 KByte L1 data cache, 8 KByte L1 instruction cache, 96 KByte L2 mixed data/instruction cache, and 8 MByte L3 cache. The line sizes of L1 and L2 are 32 Byte. L1 and L3 are direct-mapped, and L2 is 3-way set-associative.

We worked with two different compilers: *gcc* and the Portland Group C Compiler *pgcc*. Except where otherwise noted, options *gcc -O4 -funroll-loops* and *pgcc -fast -Mvect -Minline -Munroll* were used. We found these options to be most efficient for *inputF*.

Options *-O4* and *-fast* switch on various compiler optimizations that are rather unrelated to caches. *-funroll-loops* and *-Munroll* allow the compiler to unroll loops. *-Minline* allows the compiler to eliminate the function call to f, carrying out the addition directly in the main function. Inlining is automatically switched on with *gcc -O4*.

The option *-Mvect* instructs *pgcc* to attempt vectorization, which includes tiling and other loop transformations. In our example, these transformations are not applied. As *gcc* has no corresponding option anyway, the programs were tiled manually in both cases.

We chose the tile sizes by trying out the whole spectrum of reasonable opportunities and selecting the optimum, independently for each program variant. Figure 4 shows examples of outputs that we obtained during this stage. Here and in all tables below, running times are measured in seconds.

Figure 4 (and Tables 3 and 4 below) report the number of cache misses. These numbers have been obtained with the Performance Counter Library PCL [2], a set of architecture-independent library functions for accessing processor-internal hardware performance counters. PCL calls have been inserted into the source programs right before and after the loops, so that only the loop nests of Fig. 3 were measured, not the initialization or printout statements. Nevertheless, the measured values have limited accuracy. Particularly on the PC, they refer to all active processes, not just the one we are interested in. To reduce this problem, the experiments were run during periods of low load, and each experiment was repeated ten times. We report the minimum value obtained; as argued in Mukhopadhyay [13], the minimum is more meaningful than the average.

Tables 1 to 4 show representative results in a summarized form. For *pgcc*, a compiler directive has been inserted into the source program such that loop distribution is prevented for loops with a body of the form $h+ = \ldots; h+ = \ldots$. Otherwise, *pgcc* splits these loops into two, and thus destroys the effect of folding.

Table 1: Performance Overview of F-variants – Running Time

	inputF	tileF	foldF	tileFoldF	tiFoSnF
PC, gcc	14.33	15.95	11.08	10.26	10.22
PC, pgcc	14.20	16.90	12.48	10.39	12.61
Alpha	11.10	12.77	8.99	10.96	12.6

The results clearly demonstrate the usefulness of folding. With speedup being defined as the ratio between the running times of the input and current programs, the speedup is a factor of 1.14 to 2.5. The cache misses are reduced by a factor of about two. The

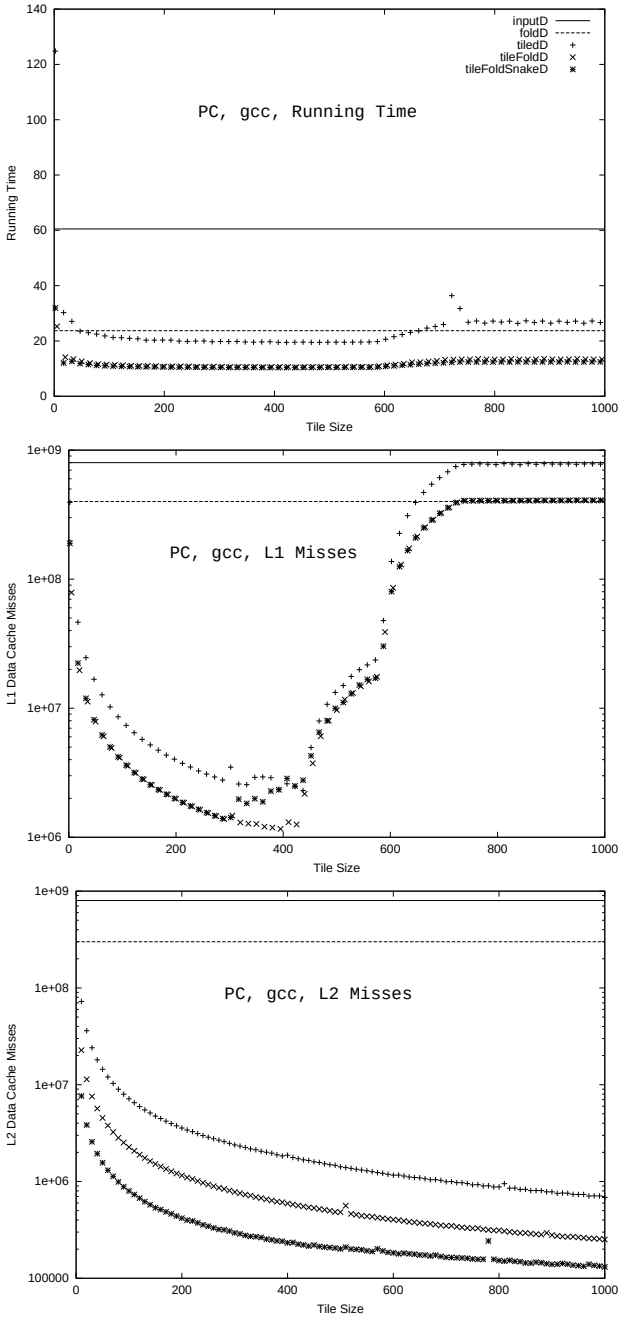

Figure 4: Influence of Tile Size on Performance Measures

Table 2: Performance Overview of D-variants – Running Time

	inputD	tileD	foldD	tileFoldD	tiFoSnD
PC, gcc	60.83	19.26	24.58	10.49	10.45
PC, pgcc	63.9	21.03	25.69	12.06	13.08
Alpha	50.97	14.80	27.01	11.99	13.20

Table 3: Performance Overview of D-variants – L1 Misses divided by 10^6

	inputD	tileD	foldD	tileFoldD	tiFoSnD
PC, gcc	788.39	5.14	394.00	1.08	2.58
PC, pgcc	789.24	7.95	394.08	3.79	1.44
Alpha	852.30	11.29	424.11	6.68	6.86

performance impact of snaking is minor: Whereas the number of L2 cache misses is quite consistently reduced, the running time gets worse in several cases.

As a general observation, speedups are much higher for the D- than for the F-variants, because the F-variants can make better use of spatial locality. Since a cache miss occurs only once per 8 array accesses, its cost is dominated by other delays. In consequence, tiling speedups are low or even negative. The D-variants use less spatial locality, and thus cache misses have a much higher performance impact.

In almost all cases, tiling, folding, and snaking reduce the number of cache misses. A few exceptions are probably due to conflict misses. The overall performance, in contrast, is sometimes degraded. This degradation is due to a more complicated program structure that increases the number of conditional tests, instruction cache misses etc. To better understand the reasons, we have measured various other performance counters that are supported by PCL. We found that in particular the number of mispredicted branches goes up.

Folding does not only reduce the number of cache misses, but also the number of floating point operations. By analyzing the *gcc* generated assembler code, we observed that when *input* carries out the four operations h+=A[i]; h+=A[j]; h+=A[i]; h+=A[j]; *fold* carries out the three operations reg=A[i]+A[j]; h+=reg; h+=reg. Only part of the overall speedup is due to this effect, however. We have checked that claim by replacing h+=f(. . .); h+=f(. . .) by h+=f1(. . .); h+=f2(. . .), with f1 denoting addition and f2 subtraction.

A favorable property of folding is its amenability to a combination with tiling, in which the overall performance effect is approximately additive. In the first row of Table 2, for instance, the speedup of tiling and folding is about a factor of three each, and the speedup of the combined transformation is about a factor of six.

Finally, we investigated the following variant of our original *input* program, in which f is an addition of three arguments:

```
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
      for (k=0; k<N; k++)
         h+=f(A[i],A[j],A[k]);
```

For this 3-dimensional example, we implemented tiling and fold-

Table 4: Performance Overview of D-variants – L2 Misses divided by 10^6

	inputD	tileD	foldD	tileFoldD	tiFoSnD
PC, gcc	688.95	1.57	207.18	0.49	0.22
PC, pgcc	699.87	1.68	227.38	0.57	0.35
Alpha	414.49	2.11	191.27	1.00	0.93

Table 5: Running Times for the 3-dimensional Program

	inputF	tileF	foldF	inputD	tileD	foldD
PC, gcc	15.28	19.43	12.78	26.40	19.75	12.79
PC, pgcc	16.04	26.42	13.25	26.51	30.38	13.73
Alpha	12.45	14.25	9.10	20.05	14.45	10.16

ing, and measured running times for the F- and D-variants. The results are qualitatively the same as in the previous example, and are summarized in Table 5.

4. RELATED WORK

Cache optimization techniques include program restructuring [11, 17] and data transformations [3, 7, 14, 16]. Wolf and Lam [17] provide a theoretical foundation for perfect loop nests, and suggest an algorithm that combines loop permutation, reversal, skewing, and tiling.

Work on tiling has, among other aspects, investigated the choice of tile size [4, 14], and the combination of tiling with padding [6, 14]. These aspects concern the avoidance of conflict misses, and are typically uncoupled from the decision to tile or not. Rivera and Tseng [15] discuss the application of tiling to multi-level memory hierarchies and conclude that it is usually sufficient to base program restructuring on a two-level model. An alternative to tiling is shackling [8]. This class of transformations is quite general, but includes neither folding nor snaking.

5. CONCLUSIONS

We have studied in-depth the performance impact of various program restructuring techniques on the example of a particular loop nest. Two novel transformations, folding and snaking, were suggested, of which in particular folding was found to be useful. Folding exploits the group-temporal reuse between non-uniformly generated references, a performance potential that has thus far been neglected in previous research. Folding achieves speedups of up to 2.5 for our example and can profitably be combined with tiling. In future work, we plan to investigate further applications, larger problem sizes, and other compilers.

6. REFERENCES

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, Dec. 1994.

[2] R. Berrendorf and H. Ziegler. *PCL: The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (Version 1.2)*, 1998/99. FZJ-ZAM-IB-9816, Available at http://www.fz-juelich.de/zam/PCL/.

[3] S. Chatterjee and S. Sen. Cache-efficient matrix transposition. In *Proceedings of the Sixth IEEE International Symposium on High-Performance Computer Architecture*, pages 195–205, 2000.

[4] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. *ACM SIGPLAN Notices*, 30(6):279–290, June 1995.

[5] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, Oct. 1988.

[6] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, Nov. 1999.

[7] M. Kandemir, J.Ramanujam, and A. Choudhary. Improving cache locality by a combination of loop and data transformations. *IEEE Transactions on Computers*, 48(2), 1999.

[8] I. Kodukula, K. Pingali, R. Cox, and D. Maydan. An experimental evaluation of tiling and shackling for memory hierarchy management. In *Proceedings of the ACM Int. Conference on Supercomputing*, pages 482–490, 1999.

[9] C. Leopold. Arranging statements and data of program instances for locality. *Future Generation Computer Systems*, 14:293–311, 1998.

[10] C. Leopold. Generating structured program instances with a high degree of locality. In *Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing*, pages 267–274. IEEE Computer Society Press, 2000.

[11] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[12] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 1997.

[13] N. Mukhopadhyay. *On the Effectiveness of Feedback-Guided Parallelization*. PhD thesis, University of Manchester, 1999.

[14] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the Int. Conference on Compiler Construction*, pages 168–182. Springer LNCS 1575, 1999.

[15] G. Rivera and C.-W. Tseng. Locality optimizations for multi-level caches. In *SC'99*, 1999. Available at http://w3.csc.ucm.es/Otros/sc99/techpap.htm.

[16] O. Temam, E. D. Granston, and 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 *Proceedings IEEE Supercomputing'93*. IEEE Computer Society Press, 1993.

[17] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991.

[18] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.