Compiler Directed Parallelization of Loops in Scale for Shared-Memory Multiprocessors

Gregory S. Johnson[1] and Simha Sethumadhavan[2]

[1] Department of Computer Sciences &
Texas Advanced Computing Center
The University of Texas at Austin, Austin TX 78712, USA
johnson@tacc.utexas.edu
[2] Department of Computer Sciences
The University of Texas at Austin, Austin TX 78712, USA
simha@cs.utexas.edu

Abstract. Effective utilization of symmetric shared-memory multiprocessors (SMPs) is predicated on the development of efficient parallel code. Unfortunately, efficient parallelism is not always easy for the programmer to identify. Worse, exploiting such parallelism may directly conflict with optimizations affecting per-processor utilization (i.e. loop reordering to improve data locality). Here, we present our experience with a loop-level parallel compiler optimization for SMPs proposed by McKinley [6]. The algorithm uses dependence analysis and a simple model of the target machine, to transform nested loops. The goal of the approach is to promote efficient execution of parallel loops by exposing sources of large-grain parallel work *while* maintaining per-processor locality. We implement the optimization within the Scale compiler framework, and analyze the performance of multiprocessor code produced for three microbenchmarks.

1 Introduction

Effective exploitation of multiprocessor systems is hampered by the complexity of developing efficient parallel code. It is not always intuitively clear to the programmer which regions of a code are parallel and how each might be tuned to achieve high per-processor performance.

Consider the simple loop nest in Fig. 1a. Spatial and temporal locality considerations favor ordering the loops as shown. However, achieving the maximal granularity of parallelism favors moving the i loop to the outer position (since the i loop is parallel while the j loop is not). Doing so distributes neighboring iterations of the i loop across processors resulting in shared cache lines. Though processors update distinct values of a, a given $a[i]$ is present in the cache on multiple processors. Thus a write to an $a[i]$ by one processor incurs the expense of invalidating the previous value of that $a[i]$ in the caches of the other processors. Advanced compiler techniques capable of addressing contradictions involving locality and the granularity of parallelism are required.

McKinley proposes a compiler optimization algorithm which promotes efficient execution of loops on SMP machines [6]. It does so by exposing sources of large-grain parallel work $while$ maintaining per-processor locality. The algorithm computes the cost

```
for j ← 1 to m do
  for i ← 1 to n do
    a[i] ← a[i] * b[i,j]
  endfor
endfor
```

(a)

```
for k ← 1 to n by strip do
  for j ← 1 to m do
    for i ← k to min(k + strip - 1, n) do
      a[i] ← a[i] * b[i,j]
    endfor
  endfor
endfor
```

(b)

Fig. 1. A simple loop nest is shown in (a). The same loop nest is shown in (b) following the application of the optimization algorithm. The loop nest has been transformed by strip-mining and interchange, such that the k loop features large-grained parallelism, while the i loop maintains good locality

of distinct orderings of the loops in a nest in terms of cache lines used. It then orders the loops such that those with the most reuse (over the fewest cache lines used) are placed innermost. Finally, the algorithm selects the outermost legally parallelizable loop, applies strip-mining, and moves any resulting iterator loop to the outermost position.

Figure 1b is the result of this algorithm applied to the loop nest in 1(a). The i loop has been strip-mined and the resulting iterator k moved to the outermost position. The iteration space of the i loop is broken into $strip$ sized contiguous regions. The k loop is parallel and results in one strip of i per processor (with $strip$ set to the trip count of i / the number of processors). Locality is maintained within each strip, and the granularity of the work per processor is maximized (as the parallel loop is outermost).

We develop an implementation of this algorithm within the Scale compiler framework. We then examine the execution time and cache performance of multiprocessor code produced for several microbenchmarks by the augmented Scale compiler. Our results indicate that significant performance gains are achievable using a straightforward implementation of this algorithm. However, we also find that this optimization is sensitive to loop structure and the availability of robust dependence testing within the compiler.

We present this work as follows. In the next section we briefly introduce the reader to the Scale compiler framework, and the specific features which support our optimization algorithm. The algorithm itself is described in Sect. 3. We illustrate its design by way of pseudocode and an example showing the transformation of a simple loop nest. In Sect. 4, we detail the key components of our implementation, showing how each affects the structure of a loop nest which performs matrix multiply. We examine the performance of this implementation in Sect. 5. Finally, we relate our work to previous efforts in parallelizing compilers for SMPs.

2 Scale

The Scalable Compiler for Analytical Experiments (Scale) was developed at the University of Massachusetts, for the purpose of enabling research in compiler optimizations. Scale includes a modular framework specifically designed to permit new optimizations to be rapidly prototyped and tested.

Our selection of Scale is driven by pragmatic considerations as well as the availability of low-level features which directly support our implementation. Scale includes frontends for the high-level languages in which many benchmarks are written (C and Fortran). Additionally, Scale includes key low-level facilities such as dependence analysis, loop abstractions, reference groups, and a simple model of the host machine. The relationship between the latter two facilities and our implementation is as follows.

2.1 Reference Groups

To accurately quantify the locality available in a loop, it is necessary to determine which array references access the same set of cache lines. Scale provides this functionality, organizing references into groups. Each group corresponds to a set of related references which exhibit one of the following: spatial locality (references refer to neighboring data elements), temporal locality (references are loop invariant), or no locality (a single reference is assigned to its own group). If the size of a cache line is also known, reference groups can be used to estimate the cost (in cache lines accessed) of a specific ordering of the loops in a nest.

2.2 Machine Model

Scale implements a simple model of the target machine which includes cache characteristics such as L1 line size. Given reference groups, cache line size cls, and the trip count of a loop C, the cost of the loop nest (with the target loop placed innermost) is estimated as follows. A reference R, representative of a group which appears in the target loop, is selected. If R is loop invariant, its cost (in cache lines required) in the context of that loop is 1. Such a reference is likely to be stored in a register. If R varies as a function of the loop index in the first array subscript dimension, its cost is taken to be C / cls cache lines (adjusted appropriately for non-unit strides). If R carries no reuse, its cost is estimated to be cls (a new cache line is required for this reference on each iteration). The total cost of the nest is the cost of the target loop multiplied by the trip counts of the outer loops. *MemoryOrder* and *NearbyPermutation* (introduced in the next section) reorder the loops in a nest by cost, such that loops with the most reuse (lowest cost) are innermost. In practice, this placement very often promotes the best overall reuse [7]. An example of this approach, applied to a triply-nested loop, is illustrated in Sect. 4.1.

Additionally, we extend the Scale machine model to include a processor count. During strip-mining, this value is used to divide the iteration count of the target loop into exactly P strips of roughly equal size. Note that McKinley's optimization assumes that the processors are homogeneous, and that each is equipped with a local L1 cache. In tandem with locality-driven loop ordering, strip size computation based on processor count insures that the potential for false sharing is minimized. Having set the stage, we now examine the optimization algorithm itself in greater detail.

3 Optimization Algorithm

Our work is based on an algorithm proposed by McKinley [6]. It utilizes dependence analysis and a simple model of the target machine, to strip-mine and interchange loops

```
INPUT: A loop nest $\mathcal{L} = \{l_1, ..., l_k\}$
OUTPUT: An optimized loop nest $\mathcal{P}$
ALGORITHM:
  procedure LoopParallelize($\mathcal{L}$)
    $\mathcal{MO}$ = MemoryOrder($\mathcal{L}$)
    $\mathcal{P}$ = NearbyPermutation($\mathcal{L}$, $\mathcal{MO}$)
    for $j = 1, m$  {outermost to innermost loop of $\mathcal{P}$}
      if (isParallel($p_j$) == true)
        $r_j$ = StripMine($p_j$)  where $r_j$ is the resulting outer loop
        markParallel($r_j$)
        if ($j$ != 1) permute $r_j$ into the outermost legal position in $\mathcal{P}$
        break
      endif
    endfor
```

Fig. 2. *LoopParallelize* reorders, strip-mines, and parallelizes loops in a given nest. The resulting nest features both large-grain parallel work and good per-processor locality.

in a nest. The goal of the algorithm is to promote efficient execution of parallel loops by exposing sources of large-grain parallel work while maintaining per-processor locality.

Our implementation utilizes the Scale dependence machinery to identify perfectly nested loops containing no function calls with unknown side-effects, within a target procedure. Each such nest is passed to our main optimization routine *LoopParallelize*, which may interchange, strip-mine, and / or mark parallel a member loop.

Pseudocode for *LoopParallelize* is shown in Fig. 2, and is very similar to the corresponding routine in [6]. *MemoryOrder* reorders the loops in a nest such that those with the most reuse are innermost and those with the least reuse outermost. This routine employs the Scale reference group data and cache line sizes for the target machine to compute the cost of each loop in terms of cache lines accessed. In the example in Fig. 1a, the *i* loop accesses fewer cache lines than the *j* loop, and is thus assigned to the innermost position by *MemoryOrder*. *NearbyPermutation* utilizes the dependence vectors produced by Scale to determine if the loop order proposed by *MemoryOrder* is legal (i.e. the vector for the reordered loop is lexicographically positive). If the proposed loop order is not legal, *NearbyPermutation* computes a close variation on this ordering which is legal. Refer to [6] for a full description of *NearbyPermutation*.

LoopParallelize now examines the newly reordered loop nest for parallelism. Working from the outermost loop to the innermost, and using the dependence information provided by Scale, *LoopParallelize* finds the first loop which is parallelizable. *LoopParallelize* strip-mines this loop. Strip-mining breaks the iteration space of the loop into contiguous "strips". The process converts a single loop into a doubly nested loop. The inner loop operates as usual, but only over "strip" iterations. The new outer "iterator" loop iterates over the strips. Strip-mining the *i* loop in Fig. 1a, results in the new *i* and *k* loops in Fig. 1b. *LoopParallelize* picks the strip size to be C / P where C is the trip count of i, and P is the number of processors in the target machine. Kennedy and McKinley have shown that this approach works well in the case where the iteration space is not smaller than the number of processors times the size of a cache line [3]. We assume this to always be the case. If the resulting iterator loop is not in the outermost position, *LoopParallelize* moves it to the outermost legal position (to maximize the granularity of parallelism), and marks it parallel. This step moves the *k* loop in Fig. 1b to the outermost position as shown.

The result of this effort is a set of optimized (where applicable) loop nests, each of which effectively balances large-grained parallel work with per-processor locality.

4 Implementation

Our augmented Scale compiler implements the McKinley optimization. We illustrate its behavior by way of application to the matrix multiply C code in Fig. 3a. The subsequent transformations are detailed step-by-step, through the following subsections. For clarity, the effect of each transformation is represented in classical C, rather than in the lower-level form produced by Scale.

4.1 MemoryOrder and NearbyPermutation

Recall that *MemoryOrder* computes an ordering of the loops in a nest such that those with the greatest locality over the fewest cache lines are placed innermost. If this loop order is illegal, *NearbyPermutation* finds a close variation that is legal, and performs the actual reordering. Consider the matrix multiply code in Fig. 3a, and assume row-major storage order. Given the original loop order as <i, j, k>, *MemoryOrder* computes that the best locality is achieved with the ordering <i, k, j>. Figure 3b illustrates how this order is computed, given a cache line size of four array elements, and the cost rules in Sect. 2.2. Observe that $c[i][j]$ and $b[k][j]$ exhibit spatial locality with loop j placed innermost, and $a[i][k]$ with k innermost. Also, $c[i][j]$ is loop invariant (temporal locality) if k is innermost, $a[i][k]$ if j is innermost, and $b[k][j]$ if i is innermost. Clearly the three reference groups benefit most (in terms of reuse), if loop j is placed innermost, followed by k and finally i. As this loop order is legal, *NearbyPermutation* interchanges loops j and k, resulting in the code seen in Fig. 4a.

```
for (i = 0; i < 100; i++) {
  for (j = 0; j < 100; j++) {
    for (k = 0; k < 100; k++) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

(a)

reference group	loop i innermost	loop j innermost	loop k innermos
c[i][j]	100 * 100 * 100	25 * 100 * 100	1 * 100 * 100
a[i][k]	100 * 100 * 100	1 * 100 * 100	25 * 100 * 100
b[k][j]	1 * 100 * 100	25 * 100 * 100	100 * 100 * 100
total cost	2,010,000	510,000	1,260,000

(b)

Fig. 3. A straightforward implementation of matrix multiply is shown in (a). The total cost of the loop nest in terms of cache lines required is shown in (b). A cost is computed for each loop in the nest to estimate the effect of placing it innermost. The cache line size in this example is 4 array elements. The table indicates that placing j innermost promotes the most reuse, followed by k and i

```
for (i = 0; i < 100; i++) {
  for (k = 0; k < 100; k++) {
    for (j = 0; j < 100; j++) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

(a)

```
for (ii = 0; ii < 100; ii += strip) {
  for (i = ii; i < min(100, ii + strip); i++) {
    for (k = 0; k < 100; k++) {
      for (j = 0; j < 100; j++) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
```

(b)

Fig. 4. The loop nest with loops j and k interchanged by *MemoryOrder* and *NearbyPermutation* to exploit greater locality, is shown in (a). The same nest is shown in (b) after applying *LoopStripMine* to loop i. The resulting loop ii breaks i into “strips" of size *trip count / processor count*

4.2 LoopStripMine

Following *NearbyPermutation*, *LoopParallelize* strip-mines the outermost parallel loop (no loop-carried dependencies at the level of that loop, or procedure calls with unknown side-effects at that level or below), via *LoopStripMine*. *LoopStripMine* divides the iteration space of the loop into contiguous "strips", such that each processor of the target SMP runs roughly equal iterations (one strip). It does so by adjusting the lower and upper loop bounds to match the "width" of a strip, and encloses the loop in an outer loop which iterates over the strips. As we allocate one strip per processor, this iterator loop is used only to demarcate the bounds of the parallel region which is later marked as such by *markParallel*. Figure 4b shows the permuted matrix multiply loop nest (Fig. 4a) following the application of *LoopStripMine*.

4.3 markParallel

LoopParallelize moves the iterator loop created by *LoopStripMine* to the outermost legal position (if it is not already there), and marks it as parallel using *markParallel*. *markParallel* marks the expression node representing the initialization of the loop index variable, and the loop exit node in the Scale control flow graph representation of the iterator loop. These nodes demarcate the bounds of the instructions which compose the target loop, and thus the bounds of the desired parallel region.

The Scale SPARC backend does not currently support multithreaded code. The complexity of modifying it to do so is significant. Instead, we modify the source-to-C emission routines to replace marked loops with OpenMP parallel regions, as shown in Fig. 5.

5 Results

We examine the performance of our optimizing compiler by analyzing the runtime behavior and cache performance of multiprocessor code produced for several microbenchmarks. Specifically, we compare the execution times and cache hit rates of the binaries over a range of thread counts on a 14-processor Sun Enterprise 5500 SMP machine. Issues with the C and Fortran frontends used by Scale (neither accepts all legal ANSI programs), and with the Scale dependence infrastructure, focus our results on three

```
#pragma omp parallel firstprivate(ii, i, k, j)
{
  ii = omp_get_thread_num() * strip;
  for (i = ii; i < min(100, ii + strip); i++) {
    for (k = 0; k < 100; k++) {
      for (j = 0; j < 100; j++) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
```

Fig. 5. The strip-mined matrix multiply code, after the application of *markParallel* and code emission. Loop ii has been replaced by an *OpenMP* parallel region

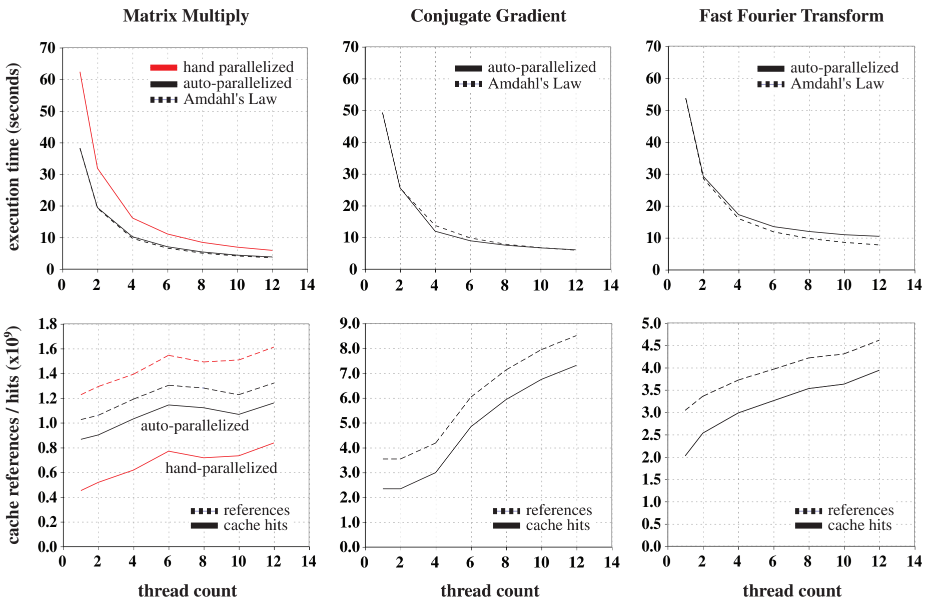

Fig. 6. Execution and cache performance of three benchmarks over multiple threads

microbenchmarks: Matrix Multiply (MM), Conjugate Gradient (CG), and Fast Fourier Transform (FFT). We perform source-to-source conversion via Scale with loop permutation (*MemoryOrder* and *NearbyPermutation*) and loop parallelization optimizations enabled. The resulting sources are compiled "-O3" with the Sun Forte 6.2 compiler. Our results are presented in Fig. 6. All measurements are averaged over five runs.

MM multiplies two 800 x 800 integer matrices. CG is a distillation of the kernel loop from the NAS Parallel Benchmarks (NPB) CG. Our CG loop nest performs an identical calculation, over the same iteration count as the Class-A version of the NPB CG. FFT computes the 2D FFT of a matrix with 4096 x 4096 elements. Our implementation correctly strip-mines and parallelizes the outermost (legally parallelizable) loop in the kernel nests of each benchmark. Additionally, Scale reorders the loops in MM as shown in Sect. 4, and (correctly) does not reorder the loops in either CG or FFT. A straightforward coding order for the outer levels of the CG and FFT loops matches the locality-driven ordering recommended by *MemoryOrder*, and imperfect nesting of the inner levels prevents more aggressive tuning.

5.1 Execution Time

The top half of Fig. 6 shows the execution performance of the benchmarks on 1, 2, 4, 6, 8, 10, and 12 threads (processor availability prevented collection of 14-thread timings). The graphs include results for the auto-parallelized code, and predictions based on Amdahl's Law. Additionally, the performance of a hand-parallelized version of MM (straightforward parallelization of loop i, with loop order <i, j, k> as in Fig. 3a) is shown. Recall that our optimizer also parallelizes i, but reorders the loops as <i, k, j>.

Amdahl's Law states that the performance of a parallel code is limited by the presence of even a small fraction of serial code. More precisely: $speedup = (s + p) / (s + p / P)$, where s is the fraction of the uniprocessor code that must execute serially, p is the fraction that may execute in parallel, and P is the number of processors. We compute Amdahl's Law for our codes on T threads by measuring the execution time on one thread (the thread creation event demarcates the time required for the serial setup code s_s from the parallel computation time p). We also measure the time s_t required to generate T-1 threads, and compute Amdahl's Law with $s = s_s + s_t$.

Amdahl's Law might be thought to predict the best possible performance of a given code on a given number of processors. However, Amdahl's Law does not consider cache effects. As a result, we see our auto-parallelized CG outperforming the prediction on 4 and 6 threads! This is likely due to a slight performance boost resulting from a larger aggregate cache. The cache performance graph for CG supports this. Note that the ratio of hits to references increases slightly between 4 and 6 threads. In all other cases, our codes perform nearly (and in the case of MM, *very* nearly) as well as the predictions.

5.2 Cache Performance

The lower half of Fig. 6 shows the cache behavior of the benchmarks. Total memory references (dashed lines) and cache hits (solid lines) are shown for the auto-parallelized codes, and in the case of MM, for the hand-parallelized code as well.

Consider the sizable differences in the cache performance of MM, as a result of merely reordering the constituent loops. Not only is the total reference count reduced, but the ratio of hits to references is dramatically improved. Notice too that the improvement is relatively stable across thread counts. The consistently high hit ratio for the auto-parallelized code suggests that cache performance was not a significant inhibitor of parallel efficiency. The overlapping lines in the graph of execution time, for the auto-parallelized code and that predicted by Amdahl's Law, supports this.

The sharp increase in references seen on CG and less so on FFT (neither benefits from loop permutation due to loop structure) as the number of threads increases, underscores the need for locality optimizations for SMPs. In the next section, we examine other efforts in this area, including a class of approaches which seek to improve locality around loop structure which inhibits optimizations such as this one, though at the expense of portability and complexity.

6 Related Work

Here, we detail the relationship between our implementation and prior art in locality optimizations and parallelizing compilers.

6.1 Locality Optimizations

Data Layout Restructuring (DLR) approaches to locality optimization improve the *spatial* locality of references to datum (typically array elements), increasing cache utilization and subsequently performance. DLR algorithms are advantageous where complex loop structure prevents reordering to improve locality. However, the analysis required for DLR is extremely complex for arrays referenced by more than one loop. Li et al. [5] propose a generalized framework for configurable DLR. They argue for affine application-specific arrays instead of a conventional row / column format. Leung [4] proposes an analysis for static array restructuring, and Chandramouli et al. [2] propose an analysis for performing dynamic array restructuring with hardware support for memory management. The latter work is complimentary in nature to the algorithm implemented here.

Program Control Restructuring (PCR) methods alter the control flow of the program to enhance *spatial* and *temporal* locality, thereby improving cache performance. A key feature of PCR algorithms is that they are less architecture-dependent than their DLR analogs. The *MemoryOrder* and *NearbyPermutation* components of the algorithm we implement, are PCR transformations. In a related effort, Wolf and Lam [8] propose a mathematical basis for quantifying reuse, and propose a unified framework for locality-improving loop transformations including interchange, reversal, skewing and tiling.

6.2 Parallelizing Compilers

Information on commercial parallelizing compilers is only sparsely available. We therefore refrain from qualitative comparisons and instead summarize the known [1] general characteristics of auto-parallelizing compilers. In particular, we focus on the Sun SPARC compiler suite, as it is closely related to our work.

The SPARC compilers attempt to parallelize *do-all* loops in Fortran and *for* loops in C. Parallel code is generated for loops with integer indices and iteration counts known at compile time. Serial and parallel code is emitted for loops with iteration counts that are not known at compile time. The serial code is executed at runtime if the iteration count is less than that required to overcome the overhead (due to thread creation and synchronization) of parallel execution. Our implementation attempts to parallelize all Fortran and C loops, irrespective of profitability. The SPARC compilers are also capable of performing loop interchange and strip-mine, but it is unclear under what conditions these are used in tandem with parallelization to reduce false-sharing on SMP machines.

7 Conclusion

We present an implementation of a parallelizing compiler optimization proposed by McKinley [6]. This optimization restructures loop nests to promote high reuse, while enabling maximally-grained parallel work. We analyze the performance of our implementation by applying it to several microbenchmarks and executing the resulting binaries on a 14-way SPARC-based SMP. Our results indicate that while the relationship between cache performance and parallel efficiency is complex, each clearly interferes with the other. Good locality can promote high parallel efficiency (MM), but as parallelism increases, it inhibits aggregate cache reuse (CG, FFT, and to a lesser degree MM). Strengthening the former effect, and reducing the latter, are key goals of this optimization. It, and others like it, offer the potential of increased utilization of large parallel machines and reduced time-to-solution for multiprocessor codes.

References

1. C. Aoki, P. Damron, K. Goebel, V. Grover, X. Kong, M. Lai, K. Subramanian, P. Tirumalai, and J. Wang. A parallelizing compiler for UltraSPARC. 1996.
2. B. Chandramouli, J.B. Carter, W.C. Hsieh, and S.A. McKee. A cost framework for evaluating integrated restructuring optimizations. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 131–141, Spain, September 2001.
3. K. Kennedy and K.S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the ACM International Conference on Supercomputing*, pages 323–334, Washington, DC, July 1992.
4. S. Leung. Array restructuring for cache locality. Technical Report UW-CSE-96-08-01, University of Washington, Department of Computer Science, August 1996.
5. W. Li and K. Pingali. Access normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems*, 11(4):353–375, November 1993.
6. K.S. McKinley. A compiler optimization algorithm for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 9(8):769–787, August 1998.
7. K.S. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.
8. M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, 1991.