

Combining Analytical and Empirical Approaches in Tuning Matrix Transposition

Qingda Lu
luq@cse.ohio-state.edu

Sriram Krishnamoorthy
krishnsr@cse.ohio-state.edu

P. Sadayappan
saday@cse.ohio-state.edu

Dept. of Computer Science and Engineering
The Ohio State University
Columbus, OH, USA

ABSTRACT

Matrix transposition is an important kernel used in many applications. Even though its optimization has been the subject of many studies, an optimization procedure that targets the characteristics of current processor architectures has not been developed. In this paper, we develop an integrated optimization framework that addresses a number of issues, including tiling for the memory hierarchy, effective handling of memory misalignment, utilizing memory subsystem characteristics, and the exploitation of the parallelism provided by the vector instruction sets in current processors. A judicious combination of analytical and empirical approaches is used to determine the most appropriate optimizations. The absence of problem information until execution time is handled by generating multiple versions of the code - the best version is chosen at runtime, with assistance from minimal-overhead inspectors. The approach highlights aspects of empirical optimization that are important for similar computations with little temporal reuse. Experimental results on PowerPC G5 and Intel Pentium 4 demonstrate the effectiveness of the developed framework.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—*code generation; compilers; optimization*

General Terms

Algorithms, Performance

Keywords

SIMD, bandwidth-limited, conflict misses, empirical search, matrix transposition, spatial locality, tiling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FACT'06, September 16–20, 2006, Seattle, Washington, USA.
Copyright 2006 ACM 1-59593-264-X/06/0009 ...\$5.00.

1. INTRODUCTION

Matrix transposition is an important kernel occurring in many applications. Although it essentially involves only memory copying, the required permutation results in complex problem-size dependent access patterns. It is a challenging task to develop transposition code that is optimized for different array sizes and alignments on different architectures.

In this paper, we address the development of a library for high-performance matrix transposition. We identify the optimization opportunities and parameters. We then employ both offline analysis and empirical search to determine the appropriate optimizations. When the exact optimization choices cannot be made at installation time, we generate multiple versions and choose the appropriate version at runtime.

We address various issues and optimization opportunities in achieving high-performance for this kernel. We present a two-level tiling strategy to improve spatial locality in the L1 cache. An analytical model to minimize TLB misses is presented. A model is developed to predict the presence of conflict misses for a subset of the problem instances. For other cases, we develop a low-cost inspector that can accurately detect the presence of conflict misses at runtime. Mechanisms to take into consideration other characteristics of the memory subsystem are included. We take advantage of the SIMD instruction sets widely available in modern computing systems. They have been shown to be beneficial in improving the memory bandwidth of bandwidth-limited computations such as memory copy. The parallelism in the SIMD hardware also enables efficient register-level matrix transposition. In exploiting these features, we address attendant issues such as data alignment.

A special code generator has been designed and implemented to automatically generate code using SSE/SSE2 on Intel architectures, AltiVec on PowerPC architectures, or just scalar instructions.

We show that the the cost of matrix transposition can be reduced significantly using our approach. The effectiveness of our approach in tuning for the target architecture is demonstrated by the performance improvement obtained in matrix transposition using just the scalar instruction set. Further optimizations using the SIMD instruction set result in the best performance of matrix transposition to be up to 43% and 98% of *memcpy* provided by the native operating system, on Pentium 4 and PowerPC G5, respectively.

The paper is organized as follows: Section 2 provides background on the matrix transposition problem. Section 3 elaborates on the matrix transposition problem and defines its input parameters. Optimizations targeting the memory hierarchy are addressed in Section 4. Section 5 details optimizations targeting the char-

```

1) for i = 0 to N1-1
2)   for j = 0 to N2-1
3)     B[i][j] = A[j][i]

```

Figure 1: A simple implementation of matrix transposition

acteristics of modern memory subsystems. Exploitation of SIMD instruction sets is described in Section 6. Section 7 presents the integrated framework that combines the various optimizations described to develop a matrix transposition library for a given target system. Experimental results are presented in Section 8. Section 9 discusses related work and Section 10 concludes the paper.

2. BACKGROUND AND MOTIVATION

Matrix transposition is widely used in many types of applications and is one of the most fundamental array operations. Despite its ubiquitous nature, matrix transposition is still commonly written in loop form as shown in Figure 1. Existing compilers are incapable of translating this implementation into an efficient executable code. For example, the program in Figure 1 was compiled using the Intel C compiler with “-O3” option. On an Intel Pentium 4 with a 533MHz front side bus, the executable achieved an average bandwidth of 90.3MB/s, for single-precision arrays with each dimension ranging from 3800 to 4200. This is only 4.4% of the sustained copy bandwidth reported by the STREAM memory benchmark [15].

Similar to many streaming multimedia workloads, matrix transposition lacks temporal locality (each element is accessed once) and has a large cache footprint. The data access pattern in matrix transposition makes it difficult to avoid performance problems due to the cache hierarchy. In particular, the following issues need to be addressed:

- In the absence of temporal locality, only spatial locality can be exploited to reduce cache misses.
- The strided access pattern potentially incurs high conflict miss penalties, due to the limited associativity of caches.
- For large array sizes, the strided access leads to heavy TLB miss penalties.

Matrix transposition can be characterized as a memory bandwidth-bound problem. A substantial portion of the total overhead is attributable to the memory subsystem (the memory bus and the main memory) in the following aspects:

- Since the output array never needs to read, a cache hierarchy with write-allocate policy wastes memory bandwidth by bringing the destination array into cache.
- Dead cycles between back-to-back read/write transactions are needed with many memory-bus designs. Previous studies [3] have demonstrated that significant bandwidth loss can be caused by *memory-bus turnaround*. Since matrix transposition consists of an equal number of read and write operations, there can be a significant loss of memory bandwidth even after careful optimization for the cache hierarchy.
- At the DRAM side, a row of data on a memory bank is read into a row buffer, providing an opportunity to exploit locality. A memory schedule may reuse the data in a row buffer efficiently eliminating additional *precharge* and *row activation*. In addition, accesses to data on different memory banks can be pipelined in popular DRAM designs.

Many modern processors have adopted multimedia extensions characterized as Single Instruction Multiple Data (SIMD) units operating on packed short vectors. Examples of these SIMD extensions include SSE/SSE2/SSE3 for Intel processors and VMX/AltiVec for PowerPC processors. When SIMD support is employed, programs such as matrix transposition can benefit from the following aspects:

- SIMD extensions often provide an independent large register file to reduce cache pressure. Data reorganization instructions provide an opportunity to efficiently reorganize data at register level.
- A SIMD load/store instruction often outperforms an equivalent sequence of scalar memory access instructions.
- *Cacheability control* is often provided by modern architectures as part of their multimedia extensions¹. The cacheability control instructions such as software prefetch and streaming writes enable software assisted data cache management and have a significant impact on streaming applications.

The above discussion is summarized in Table 1.

Table 1: Optimization challenges for matrix transposition

Category	Challenge
Cache hierarchy	Exploiting spatial locality
	Minimizing conflict misses
	Reducing TLB misses
Memory subsystem	Improving memory bandwidth utilization
	Reducing memory-bus turnaround overhead
	Exploiting efficient DRAM accesses
SIMD instructions	Exploiting intra-register data reorganization
	Accessing data with alignment constraints
	Exploiting cacheability control instructions

To illustrate the potential benefits of employing SIMD extensions in memory bandwidth-bound problems, Figure 2 shows the performance difference in memory copy performance using the scalar and SIMD instruction sets on an Intel Pentium 4 and a PowerPC G5. The configurations of these two processors are listed in Table 2.

3. INPUT PARAMETERS

Our objective is to generate an efficient implementation of the matrix transposition operation. We use a combination of offline analysis and empirical search to determine the appropriate choice of optimization parameters. The empirical search happens at installation time, and is similar to the ATLAS approach to generating an efficient BLAS library [18, 17].

The following inputs, which define the architecture, are available to the code generator at compile-time:

- Cache sizes C_1 and C_2 , in bytes, of L1 and L2 data caches, respectively.
- Cache line sizes L_1 and L_2 , in bytes, for L1 and L2 caches, respectively.
- Degrees of cache associativity K_1 and K_2 of L1 and L2 caches, respectively.
- Vector size S_v bytes. We have $S_v = 16$ in both SSE/SSE2 and AltiVec.

¹Some cacheability control instructions are not part of multimedia extensions, e.g. instructions *dcbt* and *dcbz* in PowerPC.

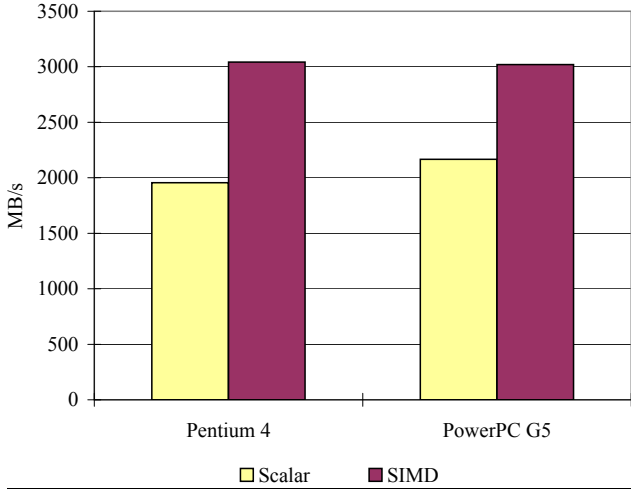


Figure 2: Improvement from using SIMD in memory copy

- Number of vector registers N_r .
- Number of TLB entries N_{TLB}^2 .

In systems with an L3 cache, L3 cache parameters are used in place of that of L2. In general, we are interested in the innermost and outermost levels of the cache hierarchy. Two platforms are used in our research, an Intel Pentium 4 with SSE/SSE2 and an Apple G5 with AltiVec. The configuration and architectural parameters of these two machines are listed in Table 2.

Table 2: Architectural parameters and configuration used for evaluation

	Pentium 4 2.4GHz	PowerPC G5 2.0GHz
C_1	8K	32K
L_1	64	128
K_1	4	2
C_2	512K	512K
L_2	64	128
K_2	8	8
S_v	16	16
N_r	8	32
N_{TLB}	64	1024
Front-side Bus	533 MHz	1 GHz
OS	Linux 2.4	Mac OS X 10.4
Compiler	icc 8.0	Apple gcc 3.3
Compiler Flags	-O3	-O3 -faltivec

The following inputs provide information on the specific matrix transposition required:

- Data element size: E bytes. Since scientific programs largely use either double-precision or single-precision floating point arrays, we consider $E = 4$ and $E = 8$ in this paper. All results in this paper are presented with $E = 4$, corresponding to single-precision arrays. Note that larger element sizes simplifies permutation and exploitation of locality.
- Array shape: (N_1, N_2) . In this work we consider row-major layout with the second dimension being the fastest varying dimension.
- Array base address: the base address of the source array $Addr_{src}$ and the base address of the destination array $Addr_{dst}$.

²We do not address the limited associativity of TLB entries on some systems.

Most architectures and compilers provide natural alignments, which means for any data element at address A with data size E bytes, it is guaranteed that $A \bmod E = 0$.

An array is said to be cache aligned if its base address and number of elements in a row are a multiple of cache line size.

The code generator takes as input the architecture parameters and generates multiple versions of code optimized for different categories of problem instances.

For brevity, we introduce the following notation:

- Vector size in terms of number of elements $V_e = S_v/E$.
- L1 Cache line size in terms of number of elements $L_{1e} = L_1/E$. Similarly, we have $L_{2e} = L_2/E$.

4. CACHE HIERARCHY OPTIMIZATIONS

In this section, we discuss the optimizations addressing the cache hierarchy in modern systems.

4.1 Tiling for Spatial Locality

We use tiling to exploit spatial locality. The tile size is chosen to be a multiple of L_{1e} , as other tile sizes lead to accessing partial cache lines, potentially increasing the cache misses over a tile size that is smaller but an integral multiple of L_{1e} . A tile size of L_{2e} is most beneficial as it helps bring each element into both L1 and L2 caches just once. Due to the absence of temporal locality, increasing the tile size beyond the L_{2e} does not improve performance, while increasing the possibility of both conflict and capacity misses.

The intra-tile loop indices will henceforth be referred to as i and j while the inter-tile loop indices will be referred to as iT and jT . Note that the row-major loop order (iT, jT) corresponds to contiguous access of the source array.

The appropriate tile sizes are chosen empirically from the range of (L_{1e}, L_{1e}) and (L_{2e}, L_{2e}) , such that the tile fits in the L1 cache. Note that non-square tile sizes can also improve spatial locality, depending on the loop order. For example, given the loop order (iT, jT) for the tiling loops, the tile size (L_{1e}, L_{2e}) allows the remainder of L2 cache lines of the source array to be accessed in the next tile, while improving spatial locality for the target array. The tile size along the fastest varying dimension for the destination array is denoted by $tile_{size_1}$, and the tile size along the other dimension is denoted by $tile_{size_2}$.

Note that tile size determination is problem-independent and can be performed once. When the architecture parameters do not dictate a particular tile size, we perform an empirical search of the tile sizes together with the associated loop order. The determination of the loop order is discussed in more detail in a subsequent section.

4.2 Two-level Tiling to Handle Cache Misalignment

The tiling described above can be insufficient when the arrays are not cache aligned. For example, consider the tiling shown in Figure 3. The tile size is chosen to be L_{2e} , and the tiles are accessed in a strided fashion, as depicted by the downward arrow. For larger problem sizes, this would result in all L2 cache lines except one to miss in the L1 cache, as the entire column has to be scanned before the partially accessed cache lines can be reused. This effectively doubles the number of L1 cache misses over the minimum possible.

Simply increasing the tile size greatly increases chances of conflict and capacity misses. A reordering of the processing of the tiles, as indicated by the numbering of the tiles, allows retaining of the partially accessed cache lines thus enabling reuse. In the best

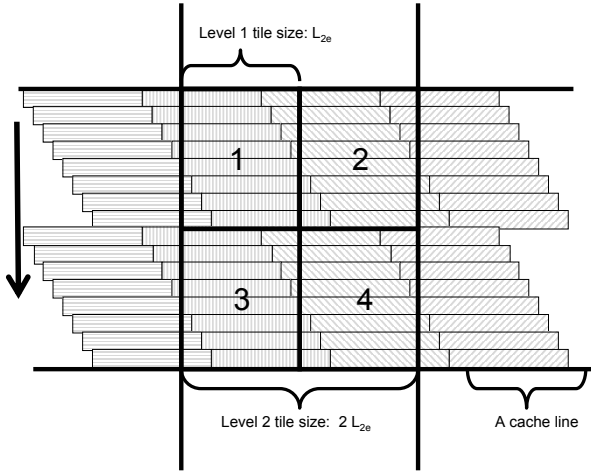


Figure 3: Two-level tiling to handle cache misalignment with tiling factor $T=2$. The arrow indicates strided inter-tile loop order

case, this reduces the number of cache misses per tile to eight from fifteen. This reordering is achieved by another level of tiling.

The tile size for the second level is chosen to be a multiple of the L2 cache line size, $T \times L_{2e}$. If a cache line is accessed in the outer tile but not accessed in the inner tile, it should be kept in L1 cache until it is fully accessed. Therefore, we have the following observations based on Figure 3:

1. If the writes for an inner tile are not buffered, explained in the next section, the L1 cache needs to hold at least $(2T+2)L_{2e}$ cache lines. Therefore, $1 \leq T \leq \frac{C_1}{2L_2L_{2e}} - 1$. We conservatively have $T = \frac{C_1}{2L_2L_{2e}} - 2$, to leave some L1 cache space for temporary variables.
2. Otherwise, only $(T+2)L_{2e}$ tiles need to be kept in the L1 cache since the data buffer is guaranteed to be cache aligned. We have $1 \leq T \leq \frac{C_1}{L_2L_{2e}} - 2$. Similar to the above case, we choose $T = \frac{C_1}{L_2L_{2e}} - 3$.

4.3 Minimizing TLB Misses

TLB misses can be severe given the memory access pattern in matrix transposition, despite tiling for the cache hierarchy. Another level of tiling can be introduced to alleviate the TLB misses. We select this tile size to be

$$tile_{size_{TLB}} = N_{TLB}/2 - \text{Max}(tile_{size_1}, tile_{size_2}).$$

This enables us to efficiently use about half the TLB entries without introducing further complexity of having conflict misses with in-cache buffers or worrying about the loop order for this outermost loop level.

On Pentium 4, which has $N_{TLB} = 64$, $tile_{size_{TLB}}$ is selected as 16, which is the same as $tile_{size_1}$ and $tile_{size_2}$, making tiling for TLB redundant. On PowerPC G5, because $N_{TLB} = 1024$ and TLB miss penalty is high, TLB tiling improves the performance by 37%.

4.4 Identifying and Handling Conflict Misses

Most existing literature on matrix transposition assumes the presence of conflict misses [4, 9]. While severe in the context of bit-reversal, matrix transposition of arrays with relatively large dimensions is not afflicted by conflict misses to the same degree. We simulated the conflict misses when the base addresses of the arrays are cache aligned and N_1 and N_2 are varied from 1 to $4 * C_2/E$. We found that conflict misses occur in the L1 cache for only 12.1% of the cases for the architecture parameters corresponding to Intel Pentium 4.

If both arrays are cache-aligned and both dimensions are multiples of L_{2e} , conflict misses can be predicted based on the following observations:

Let N be represented as $2^n \times (2s+1)$. Conflict misses can only occur in the following cases:

1. When $2^n < \frac{C_1}{K_1E}$ and $\frac{C_1}{2^nE} < tile_{size_1}$.
2. When $2^n \geq \frac{C_1}{K_1E}$.

When a tile in mini-transposition is not cache-aligned, any row in this tile of size $m \times L_{1e}$ crosses $m+1$ L1 cache lines. In practice, we have $m=1$ or $m=2$. For an arbitrary base address and array shape, it is not easy to analytically determine the presence of conflict misses. However, since the tile size is a multiple of L_{1e} , it is guaranteed that the cache footprints of all tiles have the same pattern. Figure 3 illustrates such an example, where the cache line offset patterns in different tiles are identical and the distance between two neighboring tile rows is guaranteed to be the same across tiles. Since one row in a tile crosses at most 3 cache lines in a tile, we can inspect no more than $3 * tile_{size_1}$ cache lines to determine whether there are conflict misses in one array. This inspection procedure has a very low overhead since it is only invoked once for all of the tiles in an array. Note that most modern processors have a virtually indexed L1 cache. Our inspection procedure is accurate on Pentium 4 because any given cache line equivalent of data in an array can only be mapped to a single L1 cache set (4 cache lines). In general, our procedure is inaccurate on processors with physically tagged L1 cache, such as PowerPC G5, because contiguity in cache lines depends on the virtual-to-physical mapping. Zhang and Zhang [23] observed that contiguous locations in virtual memory are usually mapped to contiguous locations in physical memory. Under this ‘‘contiguous allocation’’ assumption, our model is accurate.

We handle conflict misses by using buffers to copy the data corresponding to the innermost tile before processing. Depending on the order of the loops, one of the arrays is accessed contiguously, one row at a time. The other array is buffered to handle conflict misses. The effective detection of conflict misses is critical, since a conservative assumption of the presence of conflict misses would unnecessarily introduce copying, impairing performance. On the other hand, not detecting conflict misses can significantly worsen performance.

5. MEMORY SUBSYSTEM OPTIMIZATIONS

5.1 Buffering of Writes

Most modern architectures provide instruction-level mechanisms that enable cacheability control. For example, AltiVec provides instructions to zero out a cache line, without any read or write of the corresponding address in memory. This can be utilized to eliminate unnecessary bandwidth in reading a write-only array into cache. Going further, SSE/SSE2 provide non-temporal write instructions

that write an entire cache line, stored in a sequence of SIMD registers, directly into memory. This not only avoids wastage of bandwidth reading write-only data into cache, but also reduces cache pollution reducing the possibility of conflict and capacity misses.

Both mechanisms require the grouping of operations on the write array to operate on full cache lines. When tiling is used, in order to write a complete cache line, a buffer of size least L_{1e} elements is needed. For single-precision arrays, such an in-register buffer would require 16 registers on the Intel Pentium 4 and 32 registers on the PowerPC G5. Since there are only 8 SIMD registers on Pentium 4 and 32 SIMD registers on PowerPC G5, which are also needed for read and permutation operations, it is not feasible to maintain a buffer using the available registers. To overcome this limitation, we employ an in-cache buffer.

Buffering of writes also helps group read and write operations. Memory subsystems may have a turnaround time between read and write operations [3]. In addition, grouping of accesses to each array improves memory performance by allowing more contiguous access.

Note that buffering for writes increases the cache pressure for the L1 cache. The choice is architecture-dependent and is decided empirically at installation time.

5.2 Loop Order Selection

The choice of loop order determines which array is accessed contiguously across the processing of different tiles. This can result in reading or writing of contiguous cache lines. Different memory subsystems are biased towards either contiguous reads or contiguous writes. For example, on the Pentium 4, read of a cache line results in the prefetching of the subsequent cache line. This benefits biasing the loop order to processing contiguous source tiles.

The choice of loop order also interacts with the tile sizes chosen, as was described in Section 4. It is non-trivial to determine the appropriate loop order analytically. We empirically determine the appropriate loop order on a given machine at installation time. Note that the appropriate choice depends on the target architecture and is independent of the problem instance.

6. EXPLOITING SIMD INSTRUCTION SETS

The matrix transposition within the innermost tile is referred to as the micro-matrix transposition. On a given architecture, there are often many instruction sequences to do micro-matrix transposition. This is due to the various choices available in the SIMD instruction sets in doing the permutation operation. In addition, the order of instructions and the choice of memory access operations makes a difference. Less critical as most modern architectures support out-of-order execution. The different versions using the SIMD instruction sets correspond to different procedures to effect the transposition, and hence have different memory access patterns, instruction sequences, and register usage.

Data alignment constraints are imposed by most SIMD extensions. It implies that a memory access should be at an address divisible by the vector length. As one exception, SSE supports unaligned loads/stores. We need to obtain different versions of micro-transposition for the cases with and without perfect vector alignments.

Even if the base address of an array is aligned along vector boundaries, when dimensions are not multiples of the vector size, the vectors involved in a micro-transposition are not aligned for vector loads or stores.

Following the shifting approach [8], we can shift a data stream to handle vector misalignments. Therefore $n + 1$ aligned memory accesses and n data shifting operations are needed to obtain n con-

tiguous vectors in a stream. In our case, V_e rows in a data tile are treated as V_e data streams. If the intra-tile loop order for the innermost tile is selected to be (i,j) , $tilsize_2/V_e + 1$ loads are needed to read each row in the source tile. In this case, if the destination tile is misaligned, we cannot write it with the same strategy because $tilsize_1$ registers are needed to use the shifting procedure efficiently. When loop order (j,i) is selected, we have an opposite result that prefers vector writes but not vector reads. We assume the (i,j) order in the discussion below.

With Altivec, it is simple to conduct the shifting procedure because the permutation pattern can be computed at run time using instructions vsl and vsr . Only one micro-transposition is needed for all the misaligned cases of an array. For the other array, if it is not vector aligned, scalar instructions have to be used.

While with SSE, because there is no native support for inter-register shifts, the shifting patterns must be known at compile time or misaligned memory accesses in SSE must be utilized. Given that the tile sizes are always multiples of V_e , it is guaranteed that every complete tile shares the same alignment pattern as shown in Figure 3. The alignment of a tile row is decided by both the base address $Addr_{src}$ and $N2$. For any V_e , V_e^2 alignment patterns exist. We peel the first $(Addr_{src} \bmod E)$ loops to make the first row in every tile vector-aligned thus reduce the number of alignment patterns to V_e . For example, following this approach, when $V_e = 4$, all the 4 possible alignment patterns in a tile are $(0,0,0,0, \dots)$, $(0,1,2,3, \dots)$, $(0,2,0,2, \dots)$, $(0,3,2,1, \dots)$. For multimedia extensions such as SSE lacking flexible data reorganization support, this approach is necessary to control the number of code versions. Consequently, multiple versions of mini-transposition are necessary for SSE if misaligned memory accesses do not perform better.

7. INTEGRATED OPTIMIZATION FRAMEWORK

In this section, we discuss the generation of a matrix transposition library without information on the specific problem instance. The pseudo-code for the generated code with write buffering, TLB tiling, and one level of tiling for spatial locality is shown in Figure 4.

Figure 6 depicts the procedure used to generate the library. The best micro-transposition kernels are first selected by executing each candidate on in-cache arrays. Note that more than one kernel might have to be evaluated, each corresponding to a degree of misalignment.

The architectural parameters are then used to analytically determine the set of candidate tile sizes for the two levels of tiling and the TLB tiling. Then a version of code is generated for each degree of misalignment using the different micro-transpose kernels. Different variations of these versions are generated for all the optimization parameters to be determined at installation time. Variations corresponding to illegal parameter combinations are pruned away. For example, with misaligned inputs, using SIMD instructions on Altivec requires buffering on at least one array due to the lack of misaligned memory accesses. So it is not feasible to have a variation without buffering while using SIMD instruction set.

The different variations of each version are then empirically evaluated and the variation achieving the best average performance, in terms of bandwidth, for each version is chosen.

For each version, two implementations are generated, one with buffering for reads, to handle inputs with conflict misses, and another without. All the implementations thus generated comprise the matrix transposition library.

The runtime decision tree is shown in Figure 5. The presence

```

//Notation:
//tilesize_TLB : TsizeTLB
//tilesize_1   : T1
//tilesize_2   : T2
//in-cache buffer : BUF[T1 * T2]
//
1)for iTLB = 0 to N1-1 step tsizeTLB
2) for jTLB = 0 to N2-1 step tsizeTLB
3) for iT = iTLB to iTLB+tsizeTLB-1 step T1
4)  for jT = jTLB to jTLB+tsizeTLB-1 step T2
5)  // Mini-transposition
6)  for i = iT to iT+T1-1 step Ve
7)  for j = jT to jT+T2-1 step Ve
8)  // Micro-transposition
9)  Load Ve vectors
    A[i][j],...,A[i+Ve-1][j+Ve-1]
10) In-register matrix transposition
11) Store these Ve vectors to BUF
12) for j = jT to jT+T2-1
13) Load A[iT][j],A[iT+1][j],...,A[iT+T1-1][j]
    from BUF as T1/Ve vectors
14) Store these vectors using non-temporal
    writes from BUF to
    B[j][iT],B[j][iT+1],...,B[j][iT+T1-1]

```

Figure 4: Matrix transposition code with one-level tiling and buffering writes

of conflict miss is first verified either analytically or using the inspector described earlier. This is used to choose between the implementations with or without read buffering. Then the appropriate version of code is chosen depending on the alignment of the arrays with respect to vector size.

8. EXPERIMENTAL RESULTS

We evaluate the performance of the library generated on the Intel Pentium 4 and PowerPC G5. The code generation parameters that were determined analytically are shown in Table 3. Note that on both machines the L1 cache is large enough to hold two $L_{2e} \times L_{2e}$ tiles and $L_1 = L_2$, thus allowing the innermost tile size to be L_{2e} . Also note that there is no TLB tiling on Pentium 4 because $tile_{size_{TLB}} = tile_{size_1} \cdot 2$.

For the cache-aligned case, 8 legal parameter combinations were evaluated for Pentium 4 and PowerPC G5. For the misaligned

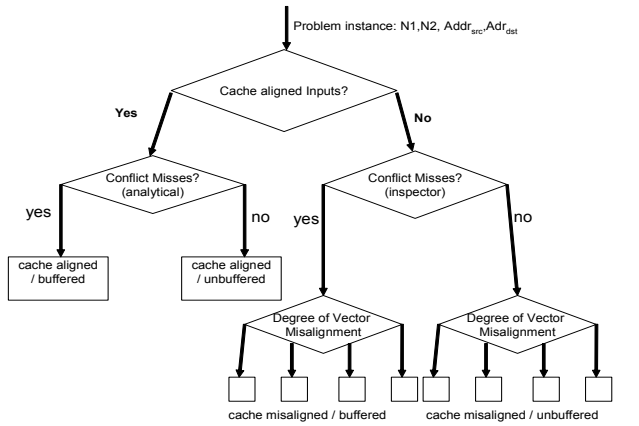


Figure 5: Runtime decision tree for implementation selection. The leaves of the tree correspond to the implementations

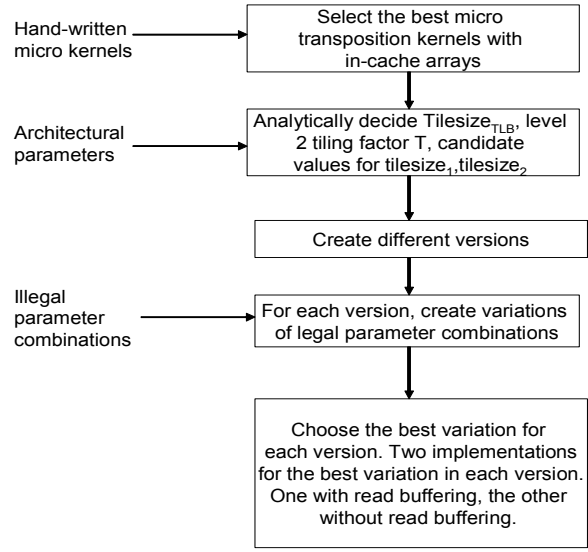


Figure 6: Library code generation procedure

cases, 8 legal combinations were evaluated for Pentium 4, but only 6 combinations on PowerPC G5. 2 combinations with (i,j) order but without copying for writes are not possible because there is no misaligned memory access on PowerPC. The empirically determined parameters are shown in Figure 4. Note that the presence or absence of write buffering does not affect the performance on Pentium 4 for cache-aligned inputs. The installation procedure took several minutes on both machines.

In Figure 7(a) and Figure 7(b), we demonstrate the contributions of individual optimizations to the overall improvement in performance. Since our optimization procedure is not incremental in nature, we chose an arbitrary order of applying the optimizations. For each set of optimization parameters the best version is chosen and is evaluated on a large number of randomly selected array sizes. With the same group of optimizations, the average performance of the best version over a large number of data points is shown. The evaluation of both the scalar and vector implementations is presented. The performance of the compiler generated code is presented in the captions to the figures. The effectiveness of memory hierarchy management is demonstrated by the performance improvement obtained in matrix transposition using only scalar instructions over the compiler-generated code. For example, *SIMD:2-level tiling + copy* denotes that we apply two-level cache tiling and include buffering for writes as an option but do not apply TLB tiling. We can observe that the SIMD version outperforms the scalar version with same optimization considerations. For example, on Pentium 4, the best SSE code has a speedup of 2.24 over the best scalar version. Several versions are not shown because some combinations of parameters are not legal or not considered with aligned cases. For example, we have to use copy to handle misaligned inputs on G5 therefore tiling without copying for writes can not be applied.

Table 3: Analytically-determined parameters for single-precision matrix transposition

	Pentium 4	PowerPC G5
tile _{size} ₁	16	32
tile _{size} ₂	16	32
second-level tile size	80	160
TLB tile size	16	480

Table 4: Empirically determined parameters for single-precision matrix transposition

	Misaligned input		Aligned input	
	Loop order	Write buffering	Loop order	Write buffering
Pentium 4	(j,i)(iT,jT)	no	(j,i)(iT,jT)	yes
PowerPC G5	(j,i)(jT,iT)	yes	(j,i)(iT,jT)	either

We observe that in most cases adding optimization components such as whether to copy for writes or to use an additional level of tiling improve performance, which shows the benefit of the additional optimization components. The only exception is with misaligned inputs, TLB tiling degrades performance after two-level cache tiling is applied. We are investigating the cause and hope to incorporate this consideration into our future work.

Figure 8(a) and Figure 8(b) show the performance of different versions when $N_1 = N_2 = N$ on Pentium 4 and PowerPC G5, respectively. Inputs are cache-aligned and all the optimization parameters are considered. Not all the versions are shown here due to space limitations, but the best version is always shown. *memcpy* is the performance of the function *memcpy* provided by the operating system. Our best version performs close to the *memcpy* provided on Pentium 4 because *memcpy* is implemented in the RedHat Enterprise Linux AS 3 release as a scalar copy. On the other hand, Mac OS X provided a highly tuned SIMD implementation of *memcpy* as part of its kernel thus it outperforms our best version by about a factor of 2.

In our experiments on Pentium 4, we have L_1 and L_2 reported as 64 bytes by PerfSuite [13], which uses the *CPUID* instruction to get hardware information of Intel processors. However, 128 bytes is often reported by micro-benchmarks as Pentium 4’s L_2 cache line size such as in [21]. This is because a BIOS feature called “adjacent sector prefetch” is enabled to prefetch the next cache line for a read. On our experimental platform, this feature is also enabled as on most desktop Pentium 4 systems. This explains why the version with (iT,jT) order outperforms the version with (jT,iT) order by 43% since the latter version does not use the prefetched cache line before it is evicted. By employing an empirical search strategy, we implicitly exploit this architectural feature without knowing its existence.

On PowerPC G5, the two best versions offer almost the same performance and the only difference between them is whether copying is used for writes. In Figure 7(b), we can also observe that any two versions, with or without copying for writes as an option, have the same average performance since adding copying only slightly degrades the performance. However, on Intel Pentium 4 the best version with buffer copy significantly outperforms the version without copying for writes. The reason is possibly that G5 has a dedicated load bus and a dedicated store bus and both buses work in a fully independent fashion. Also, the cacheability control on G5 is not as efficient as non-temporal writes on Pentium 4.

The (j,i) intra-tile order always performs better than (i,j) order on both platforms. We believe with (j,i) order there is better exploitation of DRAM’s pipelined parallelism because of the increased possibility of accessing different memory banks simultaneously.

Many architectural features such as memory-bus design or DRAM types vary greatly with architectures and even with specific machine configurations. The exploitation of these architecture-specific features are often out of the scope of most model-driven compiler optimizations. The above observations demonstrate the effectiveness of our empirical search in exploiting these “hidden” architectural features, especially for memory bandwidth-bound programs.

Figure 9(a) and Figure 9(b) demonstrate the performance of transposing $N \times N$ misaligned single-precision matrices on Pentium 4 and PowerPC G5, respectively. In the presence of cache misalignments, different parameter combinations are selected as the best as compared to the cache aligned case. Unlike the cache aligned case, the variation in the performance of different versions is not significantly different on Pentium 4. The version without buffering for writes outperforms the other versions. The misalignment precludes the use of non-temporal writes and complicates the memory bus traffic pattern. This possibly leads to the benefits from buffering for writes being overshadowed by the buffering overhead. On PowerPC G5, the (jT,iT)(j,i) combination is selected as the best version. In comparison, it underperforms the (iT,jT)(j,i) combination with cache-aligned inputs.

9. RELATED WORK

Theoretical study and empirical evaluation of optimizing matrix transposition with cache performance considerations was conducted in [4, 9]. The authors conclude that, assuming conflict misses are unavoidable, it is impossible to be both cache efficient and register efficient, and employ an in-cache buffer. Other memory characteristics are not taken into account. Zhang et al. [23] focus on how to write an efficient bit-reversal program with loop tiling and data padding. We do not have data padding as an option since we focus on generation of a library that cannot change layout of its inputs. The primary focus of that paper is on conflict misses, which is important in bit-reversal. We showed in Section 2 that conflict misses are not as widespread in the case of matrix transposition.

Different implementations of matrix transposition are investigated by Chatterjee et al. [5], with the conclusion that hierarchical non-linear layouts are inherently superior to the standard layouts for the matrix transposition problem. Optimizing for such layouts is beyond the scope of this paper.

There have been studies on how to achieve space-efficiency in matrix transposition or its more generalized forms [1, 14, 2, 7]. Our present work does not handle in-place transposition. We intend to handle in-place transposition by carefully ordering the transposition at the tile level and marking transposed tiles.

Several studies focus on how to generate or optimize intra-register permutations. The generation of register-level permutations is addressed in [12]. The algorithm optimizes data permutations at the instruction level and focuses on SSE instructions. Ren et al. [16] present an optimization framework to eliminate and merge SIMD data permutation operations with a high-level abstraction. Both studies propagate data organization along data-flow graphs and focus on reducing intra-register permutations. We manually generate various versions of micro-kernels and empirically choose the best one. While limiting, the manual process is only repeated once for every vector instruction set. The limited number of vector instruction sets allows this process to be applicable across a wide range of processor architectures.

Empirical search employed in library generators such as ATLAS [18, 17, 20] has drawn great interest because of the complexity of analytical modeling of optimal parameters for modern architectures. However, empirical global search is often too expensive to apply. Yotov et al. [22] present a strategy employing both model-driven analysis and empirical search to decide optimization parameters in matrix multiplication. Chen et al. [6] also present an approach to combining compiler models and empirical search, using matrix multiplication and Jacobi relaxation as two examples. Our work is similar in spirit but is applied to a computation that is bandwidth-limited and has no temporal locality. Matrix trans-

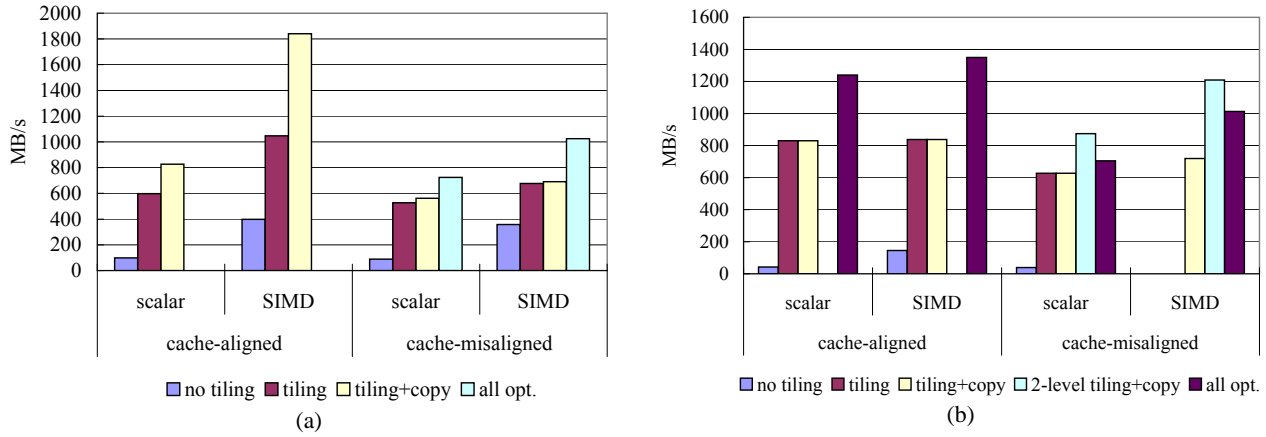


Figure 7: Effect of adding optimization considerations on (a) Pentium 4 and (b) PowerPC G5.

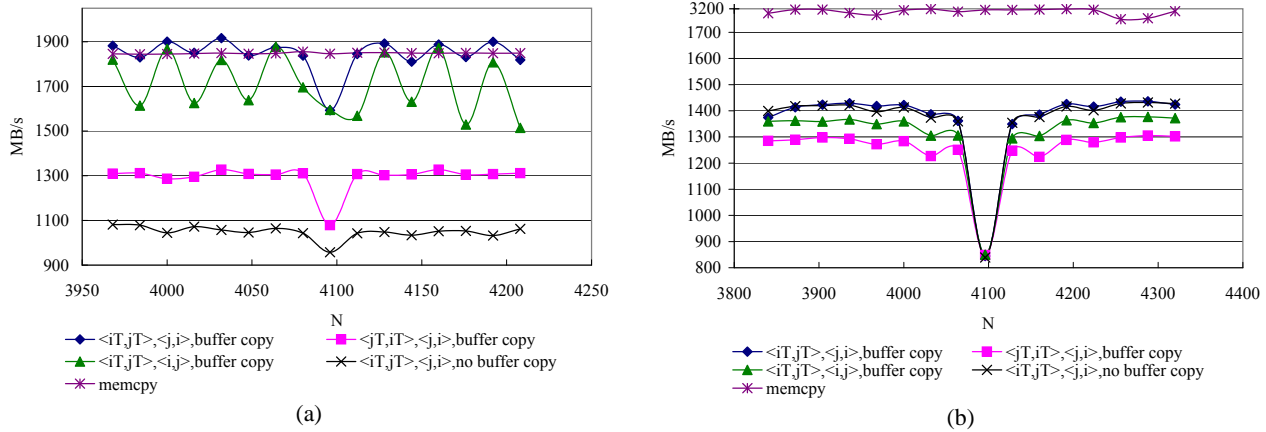


Figure 8: Performance of different versions with cache-aligned arrays on (a) Pentium 4 and (b) PowerPC G5

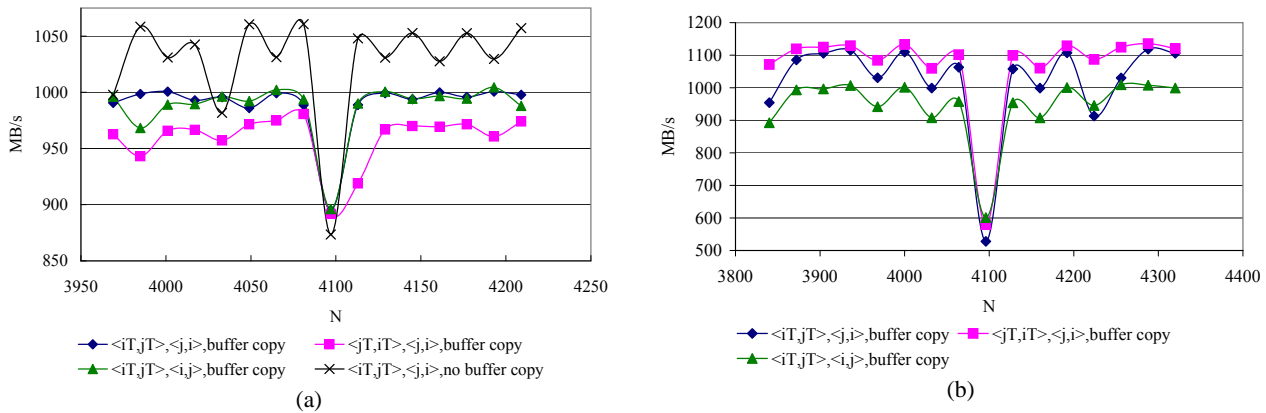


Figure 9: Performance of different versions with cache-misaligned arrays on (a) Pentium 4 and (b) PowerPC G5

position is similar to the level 1 BLAS kernels optimized by Whalley and Whalley [19] using an empirical search-based approach. But the presence of strided memory access in matrix transposition makes it harder to exploit spatial locality.

10. CONCLUSIONS AND FUTURE WORK

Extensive research has been conducted on optimizing matrix transposition and related problems because of its ubiquitous usage and unique memory access patterns. In this paper, we presented our approach employing both offline analysis and empirical search to decide optimization parameters for matrix transposition. We handle various alignments and conflict misses by generating multiple versions. Significant improvements are reported on Intel Pentium 4 and PowerPC G5 platforms with code generated by a special code generator. Several interesting observations demonstrate the effectiveness of our approach in exploiting hard-to-optimize architecture-specific features in the cache hierarchy and the memory subsystem. We believe our approach is very promising in optimizing other memory bandwidth-bound kernels and streaming applications.

We intend to generalize the approach in this paper to optimize arbitrary index permutations of multi-dimensional arrays, the generalized form of matrix transposition. Index permutation is a key operation in many scientific applications such as those in computational chemistry, weather modeling, computational oceanology, and data layout transformation to improve locality in programs [11, 10]. We are investigating the reduction in the number of versions to be generated while optimizing for the different permutations. The decrease in the size of each dimension with increasing dimensionality impairs the benefits achievable from optimizations such as loop tiling. In addition, the index calculation overhead must be effectively controlled to achieve high performance.

Acknowledgments

We would like to thank the Ohio Supercomputer Center (OSC) for the use of their computing facilities. We thank the National Science Foundation for the support of this research through award 0121676. We would also like to thank the reviewers for their insightful comments.

11. REFERENCES

- [1] M. F. Berman. A method for transposing a matrix. *J. ACM*, 5(4):383–384, 1958.
- [2] N. Brenner. Algorithm 467: matrix transposition in place. *Commun. ACM*, 16(11):692–694, 1973.
- [3] A. B. Brown and M. I. Seltzer. Operating system benchmarking in the wake of lmbench: a case study of the performance of netbsd on the intel x86 architecture. In *SIGMETRICS '97: Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 214–224, New York, NY, USA, 1997. ACM Press.
- [4] L. Carter and K. S. Gatlin. Towards an optimal bit-reversal permutation program. In *FOCS '98: Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, page 544, Washington, DC, USA, 1998. IEEE Computer Society.
- [5] S. Chatterjee and S. Sen. Cache-efficient matrix transposition. In *HPCA '00: Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 195–205, 2000.
- [6] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 111–122, Washington, DC, USA, 2005. IEEE Computer Society.
- [7] C. H. Q. Ding. An optimal index reshuffle algorithm for multidimensional arrays and its applications for parallel architectures. *IEEE Trans. Parallel Distrib. Syst.*, 12(3):306–315, 2001.
- [8] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD architectures with alignment constraints. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 82–93, New York, NY, USA, 2004. ACM Press.
- [9] K. S. Gatlin and L. Carter. Memory hierarchy considerations for fast transpose and bit-reversals. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 33, Washington, DC, USA, 1999. IEEE Computer Society.
- [10] M. Kandemir, P. Banerjee, A. C., J. Ramanujam, and E. Ayguadé. Static and dynamic locality optimizations using integer linear programming. *IEEE Trans. Parallel Distrib. Syst.*, 12(9):922–941, 2001.
- [11] M. Kandemir, J. Ramanujam, A. Choudhary, and P. Banerjee. A layout-conscious iteration space transformation technique. *IEEE Trans. Comput.*, 50(12):1321–1336, 2001.
- [12] A. Kudriavtsev and P. Kogge. Generation of permutations for SIMD processors. In *LCTES'05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 147–156, New York, NY, USA, 2005. ACM Press.
- [13] R. Kufirin. Measuring and improving application performance with perfuute. *Linux J.*, 2005(135):4, 2005.
- [14] S. Laffin and M. A. Brebner. Algorithm 380: in-situ transposition of a rectangular matrix. *Commun. ACM*, 13(5):324–326, 1970.
- [15] J. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society TCCA Newsletter*, Dec. 1995.
- [16] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for SIMD devices. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, to appear*, 2006.
- [17] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998. CD-ROM Proceedings.
- [18] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.
- [19] R. C. Whaley and D. B. Whalley. Tuning high performance kernels through empirical compilation. In *Proceedings of the 2005 International Conference on Parallel Processing (34th ICPP'2005)*, pages 89–98, Oslo, Norway, June 2005. IEEE Computer Society.
- [20] K. Yotov, X. Li, G. Ren, M.J.S. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2):358–386, 2005.
- [21] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters. In

SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pages 181–192, New York, NY, USA, 2005. ACM Press.

- [22] K. Yotov, K. Pingali, and P. Stodghill. Think globally, search locally. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 141–150, New York, NY, USA, 2005. ACM Press.
- [23] Z. Zhang and X. Zhang. Fast bit-reversals on uniprocessors and shared-memory multiprocessors. *SIAM J. Sci. Comput.*, 22(6):2113–2134, 2000.