International Journal of Parallel Programming, Vol. 26, No. 6, 1998

Quantifying the Multi-Level Nature of Tiling Interactions[1]

Nicholas Mitchell,[2] Karin Högstedt, Larry Carter, and Jeanne Ferrante

Optimizations, including tiling, often target a *single* level of memory or parallelism, such as cache. These optimizations usually operate on a level-by-level basis, guided by a cost function parameterized by features of that single level. The benefit of optimizations guided by these one-level cost functions decreases as architectures tend towards a hierarchy of memory and of parallelism. We have identified three common architectural scenarios where a single tiling choice could be improved by using information from multiple levels *in concert*. For each scenario, we derive **multi-level cost functions** which guide the optimal choice of tile size and shape, and quantify the improvement gained. We give both analysis and simulation results to support our points.

KEY WORDS: Tiling; compiler; memory hierarchy; parallelism; locality.

1. INTRODUCTION

If computers had only a single level of memory or parallelism, relatively simple cost functions could successfully guide optimization decisions. Such *one-level* cost functions commonly increase locality[1-4] and exploit parallelism.[5-9] For instance, a one-level cost function for a tiling might reflect only whether the tile fits in cache (perhaps by considering cache size, line size, and cache associativity[10]), but not the effect of the tiling on instruction level parallelism.

However, recent trends towards greater architectural complexity have increased the amount of information available to an optimizing compiler.

[1] This work supported in part by NSF CCR-9504150 and a UC MICRO grant in association with the Intel Corporation. A preliminary version of this paper appeared in the Tenth International Workshop for Languages and Compilers for Parallel Computing, August, 1997.

[2] Department of Computer Science and Engineering, University of California, San Diego.

0885-7458/98/1200-0641$15.00/0

Many machines now have multiple levels of memory and of parallelism arranged hierarchically. Memory appears as registers, several levels of cache and a translation look-aside buffer. Parallelism may occur as multiple functional units, multiple processors in a node sharing memory, multiple nodes sharing distributed memory, and so forth.

The *multi-level* aspect of memory and parallelism complicates optimizations. While a one-level cost function suffices to yield good performance at a single level of the memory hierarchy, it may not be globally optimal. In this paper, we strive to show that, as the amount of available information increases, the cost functions which guide tiling optimization choices must be similarly expanded.

Researchers have developed a number of solutions to this information expansion problem. Many have simply ignored the multi-level information, instead relying on one-level cost functions.[1, 10, 11] Others rephrase program optimization as a search problem and invent heuristics to prune the search.[12, 13]

We explore a different solution which first formulates the system to be optimized by *quantifying* both the effects of tiling choices and the interactions between such choices in a single formula, and then proceeds to minimize this formula. If the minimization is closed-form, this technique utilizes all the information necessary to achieve good performance without the nonoptimality of one-level cost functions or the expense of searches. Whether or not the minimization is closed-form, when tiling for a hierarchy of memory and parallelism this technique derives a *multi-level* cost function.

In this paper, we present evidence to support two claims about cost functions:

- **One-level cost functions may not globally optimize**. We show that tiling[1, 14] for a single level using a one-level cost function can lead to a globally *suboptimal* choice. In Section 3.1, we show for an example code that an optimal choice for *cache* leaves little instruction level parallelism, and an optimal choice for *instruction level parallelism* can cause poor cache usage. A globally optimal choice must consider the tradeoffs, given the architectural parameters from both levels. Section 3.3 reports a similar result with multiple levels of parallelism.

- **Multi-level cost functions optimize more effectively**. We show that multi-level cost functions, even when tiling at one level, can lead to a better overall choice. In Section 3.2 with matrix multiply, we obtain optimal cache and TLB miss rates for a given machine using a global, in-concert strategy that uses architectural parameters from multiple levels.

In Section 3, we consider several kernel codes and three different architectural scenarios (summarized in Fig. 1) involving multiple levels of memory or parallelism. Using these, we quantify the difference between using one-level and multi-level cost functions.

2. BACKGROUND

Given a loop nest, the *Iteration Space Graph* (*ISG*)[15] is a directed acyclic graph whose nodes represent the initial values and computations in the loop body, and whose edges represent data dependences.[16] [Note: Since the ISG has a distinct node for each value produced, *storage-related* dependences need not be represented.] Figure 4 shows the loop nest for the tiled program. The loop nest gives an order for executing the nodes of the ISG.

Tiling[1, 6, 14, 15, 17–22] can improve both data locality and parallel execution time. A tiling redefines the order of execution of the points within an ISG by specifying four pieces of information: the atomic units of execution, how to span each unit (in other words, a schedule for the points that comprise the unit), how these units are scheduled to span the iteration space, and the mapping of units to processing elements. Each unit, or *tile*, is a subset of the iteration space, typically of one size and shape (except tiles that intersect ISG boundaries). We refer to the method of spanning a tile as the *internal schedule* and the method of spanning the iteration space as the *external schedule*. Increasing the depth of a program's loop nest, with proper indices and bounds, implements a tiling.

2.1. Notation

In this paper, we will only consider internal and external schedules that are given by a total order of the iteration space axes. We represent each schedule by an ordered list of the index variables indicating loop order, from outermost to innermost. For example, Fig. 5 shows a scheduled tiling for matrix multiplication with (i, k) internal schedule and (k, i, j) external schedule.

A *module* is an architectural component, such as cache, registers, disk or network interconnections. The memory in a module is organized in *blocks*, the unit of transfer between a module and the next larger module. We denote modules by their first letter (r for register, c for cache, t for TLB, m for main memory, etc.). S_k denotes the block *size* of module k. For example, the block size S_c is the size of a cache line. We express block size in units of problem elements (for example, elements of the matrices in

matrix multiply), rather than bytes. The block *count*, C_k, is the number of blocks contained in module k.

We assume that TLB and cache use least-recently used (LRU) replacement policies.

2.2. Related Work

Related work falls into four categories: quantification of performance, determining tile characteristics for a single level, unifying optimizations for a single level and unifying optimization for multiple levels.

Quantifying performance: We employ similar counting arguments[10, 11, 20, 23, 24] to estimate the number of misses in a module. No previous work has applied these arguments to multiple levels of the memory hierarchy.

Single-level tile characteristics: Certain works[10, 20, 24] give methods for choosing tile size in a nested loop;[20] that used a "fits-in" constraint based only on memory capacity (not block size), Coleman and McKinley[10] use of "fits-in" constraint does not fully utilize block size information, though Agarwel *et al.*[24] limits block size to one. In contrast to these and other approaches to tiling size selection, our multi-level approach uses the block size at each level in a multi-level cost function.

Single-level unification: Unimodular transformations can guide loop transformations for locality[1] and parallelism.[6] These works unify only improvement-enabling transformations such as skewing, interchange, and reversal, and do not consider locality and parallelism in concert. The work[25] incorporates a larger set of transformations and unifies the transformation legality checks. AI search techniques on a decision tree of possible optimization may find a good sequence of transformations to parallelize a given program.[12] None of these works seeks to unify guidance for multiple levels of the memory hierarchy.

Multi-level unification: Unroll-and-jam can guide locality and instruction level parallelism in concert.[26] Loop fusion anal distribution affect both parallelism and locality.[27] These two works do not directly address tiling or the multi-level nature of the interactions. For matrix multiply,[28] tiles an arbitrary number of memory levels, one level at a time. It finds the optimal execution time by searching the space of tiling choices. Rather than use multi-level cost functions,[13] performs a pruned search on the space of possible combinations of minimizations of one-level cost functions. It is not clear whether this method of extending to multiple levels is equivalent to a multi-level cost function. In order to prune the search, the authors separate cache-level optimization (using a cache-specific cost function)

from scheduling choices (using a processor-specific cost function). In Section 3.1 we argue this separation is not always reasonable.

No previous work has studied the multi-level nature of program, and specifically tiling, optimizations. Our aim is to provide a system whereby a compiler (or human) may *consider* (quantitatively) the combined effect of memory hierarchy optimizations. In this sense, our work is similar to the "first-class" transformation representation of unimodular matrices and the framework found in Ref. 25.

3. MULTI-LEVEL COST FUNCTIONS YIELD BETTER PERFORMANCE

Figure 1 visualizes three architectural scenarios with multiple levels of memory or parallelism. The first, (a), represents any architecture where multiple children share a single memory, such as a cache shared by multiple processors. The second scenario, (b), represents an occurrence of two modules, where the module with the larger capacity has a smaller block count. This is typical for machines with a Translation Lookaside Buffer (TLB). Scenario (c) represents any architecture with multiple levels of parallelism. For each architectural scenario, we illustrate with example codes that multi-level cost functions yield improvements.

Each scenario involves multiple conflicting goals. To balance these goals, we need a single objective function that incorporates the competing costs; we choose total execution time. Unlike approaches that use cache misses or processor utilization, total execution time encapsulates the multi-level tradeoffs. However, constructing such a cost function requires more architectural information.

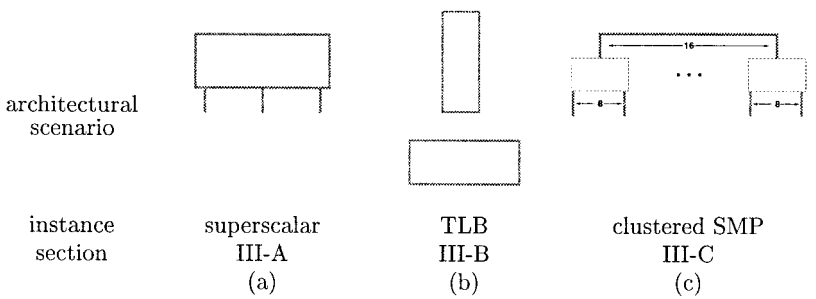


Fig. 1. Three architectural scenarios with deleterious tiling interactions. Higher modules represent larger, slower memory units; lower modules are smaller and faster caches or processors. Bold elements highlight the important features in each architecture.

3.1. Memory Shared Among Processors

To motivate multi-level cost functions, we start with a simple example. This example shows that any optimization strategy must trade off data locality against parallelism, and the best strategy depends on the program and details of the target machine.

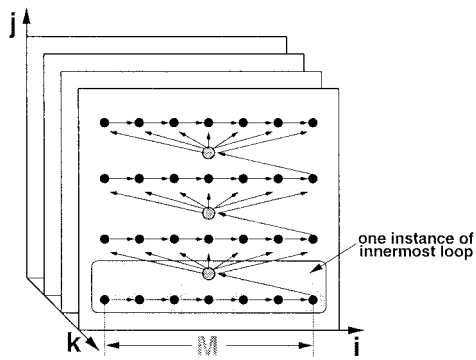
The general architectural scenario is that of Fig. 1a, where a parent module allocates its limited memory capacity among multiple children; we consider a particular instance in which a cache holds data for a superscalar or superpipelined processor. A different but equally valid instance is an L2 cache shared by multiple processors. We assume the processor can exploit instruction level parallelism (ILP) if there are sufficiently many independent operations.

Figure 2a shows our triply-nested example code, where f is an unspecified function. Figure 2b depicts the ISG for this problem. The dependences in any i-j plane sequentialize the computation in that plane. In order for a tile to introduce reuse, it must execute more than one instance of the full innermost loop. Having done so, there is no extra data movement required to complete the entire plane. Thus, the only significant tiling choice is how many such planes to include in the tile.

Note also there are no dependences *between* the $i-j$ planes, and so the computation is "embarrassingly parallel" in the k dimension. If a tile includes multiple planes, then standard transformations can create an

```
for k=1 to S do
  for j=1 to N do
    sum = 0
    for i=1 to M do
      A[k,i]=f(A[k,i-1],
               A[k,i],av)
      sum += A[k,i]
    av = sum / M
```

(a)



(b)

Fig. 2. Running sum code (a) and iteration space graph; (b) for Section 3.1: a set of independent planes. The lightly shaded nodes represent the computation of av and the black nodes represent inner loop computations. The choice of planes per tile affects both cache behavior and instruction level parallelism.

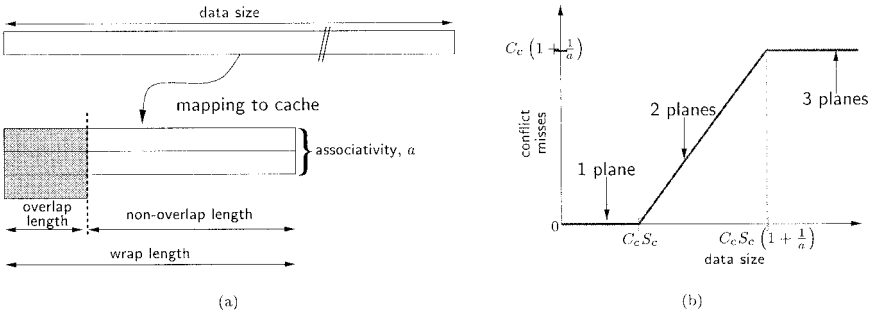


Fig. 3. The simple cache model used in Section 3.1 for a cache with C_c lines, line size S_c, and associativity a. (a) illustrates our derivation of the model which is shown in (b). For a tile of p planes, the data size (elements per cache tile) is pM, where M is the size of the input matrix in the **i** dimension. In (a) overlap length is $pM - C_cS_c$, cycle length is C_cS_c/a, and nonoverlap length is $C_cS_c/a - (pM - C_cS_c) = C_cS_c(1 + 1/a) - pM$.

innermost loop that executes points from several independent planes, allowing better instruction level parallelism.

To guide the choice of planes per tile, we develop a simple performance model. In this model, optimizations will be driven by information in a table; Table I shows the example we use in this section. Table I describes the effect of adding planes to a tile on both cache (second column) and processor utilization (third column); hence, it encapsulates the cache miss cost function, *CacheMiss*, and processor utilization cost function, *ILPTime* . Table I can be generated either analytically, experimentally (such as in Ref. 13), or by some combination of the two. For this paper, the processor

Table I. Tiling Choice Affects Cache Miss Penalty and Execution Time in Opposite Ways[a]

	(Cycles per point)	
Planes per tile	Cache miss penalty (*CacheMiss*)	Execution time (*ILPTime*)
1	0	10.4
2	1	5.3
3	4	5.1

[a] The input to our in-concert optimization is a table such as this one. The optimizer needs to know the effect of adding planes on cache and on instruction level parallelism. One can generate this table in many ways. We derived cache miss penalty using the cache model described in Fig. 3 and information about instruction level parallelism by running small enough versions of the input loop to allow measurements of execution time in the absence of cache misses. The benchmarks ran on an IBM POWER2 with $C_c = 256$, $S_c = 64$, $a = 4$.

utilization column, *ILPTime* , comes from actual executions of the loop nest with M sufficiently small so that there are no cache misses. The cache column, *CacheMiss*, comes from a static cache model, described next.

This cache model is shown in Fig. 3b and derived in Fig. 3a for a tile of p planes. Assume that the A matrix is stored so that the innermost loop accesses the data sequentially; to do so, the data from the p planes must be interleaved. [Notice that for $p = 1$, this corresponds to row-major order allocation of the A array.] Given that we have arranged the A array in this way, if the data for a tile fits in cache ($pM < C_cS_c$), we get no conflict misses; every cache line will be temporally reused. However, if $pM > C_cS_c$, the data for a tile doesn't entirely fit in cache. In an a-way set associative cache, data that is stored sequentially "wraps around" cache after C_cS_c/a items. This wrapping causes conflict misses in those regions of the tile data that wrap around more than a times, shown in the shaded portion of Fig. 3a. The length of this problematic region, the "overlap length" in Fig. 3a, is the length of data minus the capacity of cache, or $pM - C_cS_c$.

However, we are concerned with the number of elements which do not overlap, the "nonoverlap length." The nonoverlap length is just wrap length minus overlap length, or $C_cS_c/a - (pM - C_cS_c)$. We will get no temporal reuse when the nonoverlap length is zero, or when $pM = C_cS_c(1 + 1/a)$. We have now defined three regions by two endpoints: perfect temporal reuse when $pM < C_cS_c$, no temporal reuse when $pM > C_cS_c(1 + 1/a)$, and reduced temporal reuse inbetween. In the first region, the number of conflict misses is zero; in the third region we endure one conflict miss every line for pM accesses. In between the number of conflict misses grows linearly with pM: inspecting Fig. 3a, we see that the number of conflict misses (the shaded region) is the overlap length times $a + 1$ divided by S_c (because the code is scheduled for spatial reuse); more precisely, $(pM - C_cS_c)(a + 1)/S_c$.

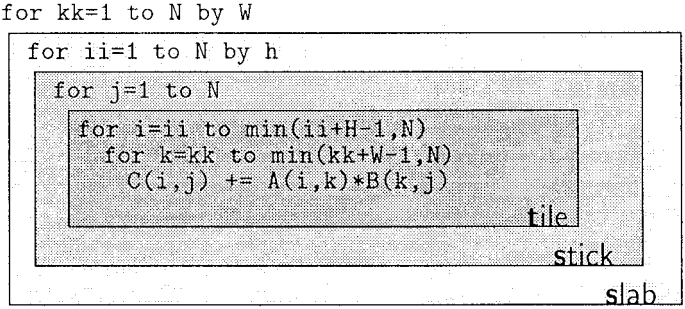


Fig. 4. Tiled code for matrix multiply.

To make this section more concrete, we generate an example cost function table. For this example, we assume a problem size that ensures an entire $i-j$ plane fits in cache, but two planes do not. For three planes and higher, the maximum cache miss penalty has been reached. This table further assumes the data for four iterations fit in one cache line and a cache miss costs sixteen cycles. The *ILPTime* cost function is for the IBM Power2 with values taken from Carter *et al.*[29]

Using our example cost function table, this section considers choosing the number of planes in three information domains: only considering cache characteristics, only ILP characteristics, and both. This final domain must balance the tradeoff between ILP and cache locality.

Cache alone: What tiling choice minimizes cache misses? Paying attention only to cache, we pick the row of Table I which minimizes the second column. For our example, we would pick the first row, corresponding to a tile of one plane. Unfortunately, this strategy, which minimizes cache misses, limits the instruction level parallelism to what is available in a single plane. The opportunity to execute iterations from independent planes has been lost.

Instruction Level Parallelism only: What tiling choice maximizes instruction level parallelism? As the number of planes increases, the rate of execution time improvement decreases and might even become negative. A compiler that only considers ILP would choose the minimum number of planes which realizes the minimum execution time, in this case $p = 3$. However, this choice yields the maximum average cache miss penalty.

In concert: The in-concert strategy must consider the effects of *both* ILP and cache misses. We use an execution time cost function of

$$E = \min_{p} (SNM(CacheMiss(p) + ILPTime(p)))$$

Note that it suffices to minimize $CacheMiss(p) + ILPTime(p)$ over the number of planes p.[3] To do this, the compiler need simply check the sum of the two entries for each p, line by line in both tables, and choose the p with the minimum sum. In this case, the choice would be $p = 2$, which is distinct from the cache-first strategy, $p = 1$, and the ILP-first strategy, $p = 3$. In this simple case, using the cache-specific cost function or the ILP-specific cost function did not yield the best solution, which is found by minimizing the *sum* of the two cost level-specific functions.

[3] Out-of order execution and nonblocking caches may complicate the formula; at worst the formula has a factor of the *maximum* instead of the sum of *CacheMiss* and *ILPTime*. Our argument works just as well in this case, since the two factors still must be considered in concert.

Table II. TLB and L1 Data Cache Characteristics for Various Workstation Processors

	TLB		L1 Cache	
	entries	kB/line	entries	bytes/line
Power2	512	4	1024	256
PA-8000	96	4+	16384+	16+
R10000	64	4+	1024	32
UltraSPARC	64	8+	512	32
Pentium II	64	4	512	32
21164	64	8	256	32

Others have proposed[13] a different solution to the ILP-cache tradeoff. They recognize the interdependence of optimization choices and use a search procedure to take into account many of these interactions. However, they prune the search procedure by making decisions for ILP *before* tiling decisions for cache locality. While this bottom-up procedure may work much of the time, our simple example illustrates that sometimes the best choice requires considering information from both cache and processor levels *in concert*.

3.2. Tall, Thin Modules

Our second architectural scenario consists of two modules, a and b, where a has fewer blocks, $C_a < C_b$, but greater capacity, $C_a S_a > C_b S_b$. We are particularly interested in the case that a is a *tall, thin module*;[4] i.e., when $C_a \ll S_a$, as in Fig. 1b. Tall, thin modules occur in virtually every contemporary workstation in the form of a translation look-aside buffer, as shown in Table II. For the remainder of this section, we will use the example of TLB[5] and cache.

In this section, we demonstrate the improvement seen by optimizing for TLB and cache *simultaneously*. We first quantitatively derive the optimal tiling for TLB, using a TLB-specific cost function and ignoring the cache completely. This tiling will lead to cache thrashing, so we then perform an additional level of tiling for cache. Next we pursue the reverse strategy: first tile for cache, show there is TLB thrashing, and then tile the

[4] We visualize modules as rectangles with height corresponding to the block size and width corresponding to the block count. The scenario of this section is visualized as a large module that is too narrow to contain a smaller module.

[5] We consider a data item to be "in the TLB" if the virtual to physical address translation for the item's virtual page is cached in the TLB.

cache-tiled code for TLB. Finally, we show that the best strategy balances TLB miss rate and miss cost with those of cache.

We explore the benefit of multi-level optimization using matrix multiply. We chose to look at matrix multiply for two reasons. First, much work has been put into it's optimization,[28] but none has explored optimizing both cache and TLB simultaneously. Second, the relatively simple data access patterns of matrix multiply allows straightforward locality analysis. However, this latter aspect of matrix multiply implies that matrix multiply is a compute-bound application. Being compute-bound, all tiling strategies which remove TLB *and* cache thrashing will have similar execution times. However, our goal is to provide an analytical framework for simultaneous optimization, rather than optimize any one application. Therefore, we nonetheless continue to analyze matrix multiply (in this paper) due to its simplicity.

To focus on the interaction of TLB and cache tiling choices (as opposed to all the other choices affecting performance), optimized versions use the external schedule (k, i, j) and the internal schedule (i, k). Figure 4 gives the code and defines our naming convention: tiles stacked along the j dimension are *sticks* and sticks stacked along the i dimension are *slabs*; Fig. 5 illustrates the tiling. The only difference between optimized versions will be the choice of the *tile size*, given by H and W.

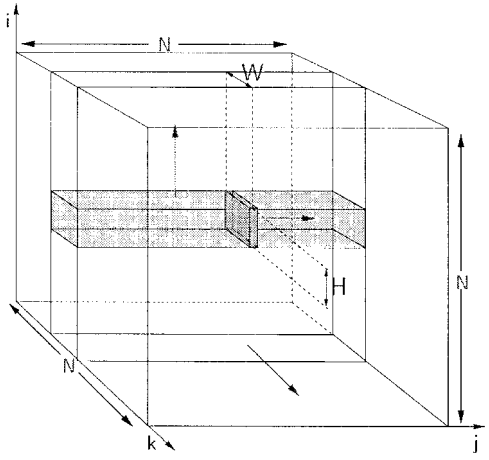


Fig. 5. Our scheduled tiling for matrix multiply. This tiling uses the (i, k) internal schedule and (k, i, j) external schedule. The intensity of the shading measures nesting depth; regions with the darkest shadings correspond to inner loops. The picture remains the same whether we are tiling for TLB or cache, but the height and width (H and W) change.

To determine the optimal tile height and width, we use a somewhat simplistic execution time cost function of miss cost times miss count. Let i_k be the idle time due to a miss at level k and M_k be the miss count at level k. Using these, we define two cost functions: a *single-level* cost function E_k^{single} and a *multi-level* cost function E^{multi}:

$$E_k^{single} = i_k M_k(H, W)$$

$$E^{multi} = i_t M_t(H, W) + i_c M_c(H, W)$$

Notice that minimizing the single-level cost function is equivalent to minimizing the miss count. In contrast, minimizing the multi-level cost function balances the relative costs of TLB and cache misses.

Thus, to determine the optimal tile height and width, we need two formulas. The first, M_k predicts the number of mandatory misses on A, B, and C in a module k. The second predicts the memory requirements of the tile; the optimal tiling must satisfy the capacity requirements of TLB and cache. As we assume a fully associative cache, we do not factor in self- and cross-interference. [Note: In the future, we plan to model associativity.]

To derive these two formulas, we prove a lemma which closely bounds the expected number $B_k(H, W)$ of blocks that must be brought into module k to hold an $H \times W$ submatrix. We use this lemma to generate both the miss count formula and the capacity requirements of a tile. First, a formal definition:

Definition 1. Suppose A is an $N \times N$ matrix[6] partitioned into $H \times W$ submatrices, and k is a module. Let $B_k(H, W)$ be the expected number of distinct (full or partial) blocks of module k included in a submatrix, where the average is over all submatrices in the partitioning.

The exact value of $B_k(H, W)$ depends on the alignment of the matrix columns in module k; for instance, even if $H = 2$, the columns of A might be allocated in a way that each column spans two blocks of k. Rather than making assumptions about the alignment, we derive an upper bound on $B_k(H, W)$.

Lemma 1. Let k be a module and A be an $N \times N$ matrix stored, without loss of generality, in column-major order; that is, each column of

[6] Treating nonsquare matrices required nominal modifications to the formulas in this section.

A is allocated in contiguous storage. Suppose that A is partitioned into submatrices of size $H \times W$ Then,

$$B_k(H, W) < \frac{N(\lceil N/H \rceil + \lceil N/S_k \rceil)}{\lceil N/H \rceil \lceil N/W \rceil}$$

Proof. Define a *piece* to be the set of memory addresses that represent the intersection of one of the submatrices with a single block of module k. Also define a *block address* to be the address of the first element of a block of k. We will first derive an upper bound on the total number of pieces, from which the lemma will follow easily.

Each piece has a unique identifier, its smallest memory address. The key insight is that each identifier is either a block address or the first address in a submatrix column. Thus, the total number of pieces is bounded by the number of submatrix columns, $N\lceil N/H \rceil$, plus the number of block addresses contained in A. To bound this latter number, observe that the column alignment that includes the maximum number of block addresses has each matrix column start at a block address. In this case, the number of block addresses per column is $\lceil N/S_k \rceil$.

Thus an upper bound on the total number of pieces is $N(\lceil N/H \rceil + \lceil N/S_k \rceil)$ Dividing by the number of submatrices, $\lceil N/H \rceil \lceil N/W \rceil$ yields the desired bound on the average number of pieces per submatrix. ∎

We first apply this lemma to the problem of ensuring a tile remains resident. For matrix multiply, the memory must contain W columns of height H from A, one column of height W from B and one column of height H from C. However, this memory bound is not sufficient to avoid capacity thrashing, dale to the nature of LRU (least-recently used policy) caches. Because our external schedule runs along the j dimension, the very next tile reuses the columns from A, but does not need the B and C columns for a long time. However, either possible internal schedule, (i, k) or (k, i), leaves the first columns of A the least recently used. Therefore, to ensure no thrashing due to an LRU policy, we must leave room for *two* columns from B and *two* columns from C. Therefore, for the tile to remain resident, it is necessary that $B_k(H, W) + 2B_k(W, 1) + 2B_k(H, 1) < C_k$.

Next, we apply the lemma to determine the number of mandatory misses. We denote this quantity for level k by M_k. The total number of mandatory misses on A, B, and C, is the product of the number of sticks with the misses incurred on B and C in a single stick, together with the cost of bringing in the A matrix exactly once. Each $H \times W \times N$ stick intersects an average of $B_k(H, W)$ blocks of the A matrix, $B_k(W, N)$ blocks of the B matrix and $B_k(H, N)$ of the C matrix. Thus, executing the stick will incur $B_k(H, W) + B_k(W, N) + B_k(H, N)$ misses; we refer to the number of misses

for a stick of height H and width W as $\mu(H, W)$. We must also count partial sticks, in the event that W does not divide N or H does not divide N. We now have a formula which counts an upper bound to M_k (We further assume a sufficiently large N so that the lines of the B and C matrix used in each stick do not remain resident in the TLB.):

$$M_k = \lfloor N/H \rfloor \lfloor N/W \rfloor \mu(H, W) + \lfloor N/W \rfloor \mu(N \bmod H, W) + \lfloor N/H \rfloor \mu(H, N \bmod W) \mu(N \bmod H, N \bmod W)$$

Yet, this miss count formula does not bound as tightly as it might. When $N - H < S$, M_k needlessly overcounts cache misses. In this case, one TLB[7] line touches the tile in two or more columns. Our derivation of B_k amortized *two* sources of line-tile intersections: the start of a tile column and the start of a line. When $N - H < S$ we need only amortize the second source of intersections. Next observe that when $N - H < S$, the number of misses is independent of tile height H. Thus, over the entice contiguously allocated matrix, we get $\lceil N^2/S_k \rceil$ line starts for $\lfloor N/W \rfloor$ vertical tile swaths. We now have a piece-wise defined, but more tightly bounding, B_k:

$$B_k(H, W) = \begin{cases} \dfrac{N(\lceil N/H \rceil + \lceil N/S_k \rceil)}{\lfloor N/H \rfloor \lfloor N/W \rfloor} & N - H \geqslant S \\ \dfrac{\lceil N^2/S_k \rceil}{\lfloor N/W \rfloor} & \text{otherwise} \end{cases}$$

Now that we have a tightly-bounding miss count formula, we need to simplify it to allow closed-form minimization (rather than integer programming). We refer to former as the **complex** miss count formula and the latter as the **simple** miss count formula. In the **simple** formula, we ignore misses on the A matrix (as they are only $O(N^2)$ compared to $O(N^3)$ misses on each of B and C). We also assume that $\lceil N/H \rceil \approx N/H$ and likewise for N/W; in doing so, we ignore the effect of partial tiles. Next, we simplify the residency requirements by only counting lines from A (and ignoring the much smaller number of lines from B and C). The **simple** formula also ignores the effect of an LRU policy. Finally, the simplified formula ignores wrap-around. Though not a guaranteed upper bound on the number of misses, this particular formula allows easy minimization. We could alternatively derive a continuous but more complicated upper bound. We could also potentially refine the **simple** formula by making fewer assumptions but still ensuring that it is continuous. These extensions would have only

[7] Cache lines will "wrap around" only for very small matrices.

needlessly complicated matters: our simulation next results show that the **simple** cost function predicts quite well. The resulting **simple** formulas are:

$$B_k^{simple} = HW\left(\frac{1}{H} + \frac{1}{S_k} + \frac{1}{N}\right)$$

$$M_k^{simple} = \frac{N^2}{HW}(B_t(W, N) + B_t(H, N))$$

$$= N^3\left(\left(\frac{1}{H} + \frac{1}{W}\right)\left(\frac{1}{N} + \frac{1}{S_t}\right) + \frac{2}{HW}\right)$$

When using this **simple** miss count formula, we must scale back the size of TLB and cache. The **simple** miss count only counts lines from A towards capacity (and ignores many other issues, as well). Therefore, similar to Sarkar *et al.*[11] and the *effective cache size*,[13] we modify the "fits in" constraint with a "fudge factor" of 75%: $B_k(H, W) \leqslant 0.75C_k$. Again, we need only use this fudge factor when using M_k^{simple}, as this formula ignores many architecture-code interactions.

Minimizing the M_k^{simple} subject to a "fits-in" constraint of $B_k(H, W) = 0.75C_k$ produces the following equations for the optimal TLB tile size. These equations apply when optimizing for any level *in isolation*.

$$N'_k = 1 + S_k/N$$

$$\mathscr{H} = N'_k H = \sqrt{0.75C_k S_k N'_k + 2S_k^2}$$

$$W = \frac{0.75C_k S_k}{S_k + \mathscr{H}}$$

TLB decisions first: Our first tiling strategy optimizes for TLB first. As an example, consider the MIPS R10000 where $C_t = 64$ and $S_t = 1024$, the standard UNIX page size measured in words. For 1200×12000 matrices, a tile of size $H = 800$ and $W = 20$ minimizes TLB misses. Interestingly enough, the optimal tile is quite far from square.

Unfortunately, this optimal TLB tile leads to poor cache performance. The tile needs HW, or about 16,000, words from the A matrix, but the R10000 L1 cache holds only $C_c S_c = 1024 \times 8 = 8192$ words. Therefore, the tile's submatrix of A must be reloaded into cache for each TLB tile. The other systems in Table II suffer the same problem. (See Table III.)

Cache next: To overcome the cache problem, the "TLB first, cache next" optimization order considers the entire *stick* of TLB tiles as an iteration space, and optimizes it for cache performance. It turns out that cache

Table III. Tile Height and Width Picked by the Three Strategies Using the R10000 and UltraSPARC Cache and TLB Parameters Given in Table II

strategy	R10000 $H \times W$	UltraSPARC $H \times W$
TLB-first	297×20	145×20
cache-first	79×42	56×45
in-concert	177×33	77×36

misses can be minimized by splitting the stick lengthwise. This is equivalent to simply using a shorter tile in the original five-loop program of Fig. 4.

The optimal cache tile size, given that the TLB considerations have already chosen the tile width to be $W = 20$, is obtained by choosing the largest value of H that satisfies the "fits-in" constraint for cache, $B_c(H, W) = 0.75C_c$. In our R10000 example, this gives $H = 297$.

Cache first: Now we apply the reverse strategy, considering cache first, instead of TLB. Using the method described in the "TLB first" section, but using the cache parameters, gives $H = 79$ and $W = 70$ using R10000 parameters. But now, TLB degrades performance. Since the TLB has only 64 entries and each column of an A-submatrix uses distinct TLB blocks, there will be TLB misses on many or all (depending on the TLB's associativity) columns of each cache tile.

TLB next: To remedy the problem with TLB, we subtile each stick, this time for TLB by reducing the width of the cache tiles in the five-loop program to satisfy the "fits-in" constraint, $B_t(H, W) = 0.75C_t$. For the R10000, this gives $H = 79$ and $W = 42$.

In-concert: Can we do better by taking advantage of the interactions between TLB and cache? To find out, we minimize E^{multi} subject to the two constraints $B_t(H, W) \leqslant 0.75C_t$ and $B_c(H, W) \leqslant 0.75C_c$. It turns out that the former suffices to constrain width while the latter suffices to constrain height. Solving this minimization problem yields the following equations. These formulas apply to any level for which we have considered the characteristics of two consecutive levels.

$$\mathscr{H}' = N'_c H' = \left(S_t + \sqrt{0.75C_t S_t N'_t + 2S_t^2 \frac{1 + m_{t/c}}{S_{c/t} N'_{c/t} + m_{t/c}}} \right) C_{c/t} - S_t$$

$$W' = \frac{0.75C_c S_c}{S_c + \mathscr{H}'}$$

Table IV. Simulated and Predicted Misses for the Three Tiling Strategies Using Both (a) R10000; and (b) UltraSPARC Hardware Parameters[a]

		(A) R100000			(B) UltraSPARC		
		simulated	predicted (% error)		simulated	predicted (% error)	
	strategy	misses/10^6	simple	complex	misses/10^6	simple	complex
TLB	TLB-first	0.781	−4.10	+1.79	0.762	+12.2	+0.92
	cache-first	1.182	−1.02	+10.4	0.838	+12.9	+1.91
	in-concert	0.688	+2.32	+5.23	0.768	+12.8	+1.43
Cache	TLB-first	12.89	−5.51	+3.41	14.04	−3.42	+6.05
	cache-first	9.713	−7.64	+4.40	10.86	−7.09	+3.68
	in-concert	9.178	−8.38	+11.14	10.62	−4.80	+7.16

[a] The simple formula allows analytical minimization while the complex formula is a discontinuous upper bound.

Using the same TLB and cache parameters as before and assuming a TLB miss is three times as costly as a cache miss yields $H = 177$ and $W = 33$.

Which is best? To verify the analytical formulas for misses, we counted cache and TLB misses on the SimpleScalar 2.0[30] cache simulator, **sim-cache**. The simulation is parametrized by characteristics of the target architecture: C_k, S_k and associativity for each module k. As our formulas do not yet account for associativity, we ran the simulations on fully associative caches.

We simulated the three tiling strategies using the characteristics of the R10000 and the UltraSPARC given in Table II. The simulations demonstrate the complex formula accurately overestimates both TLB and cache misses. Relative to the predicted figures, the **complex** cost function[8] overestimates TLB miss counts by 1–10% and overestimates cache miss counts by 4–11%. The **simple** cost function, as expected, sometimes underestimates miss counts; it also overestimates TLB misses for the UltraSPARC as it does not factor in wrap-around. Table IV presents these comparative figures.

Figure 6 compares the simulated data access times of the three optimization strategies. This chart assumes a TLB miss is three times more expensive than a cache miss. The simulation results indicate that, using R10000 parameters, the cache-first and TLB-first optimization strategies

[8] The figures in Table IV include an extra $3B_k(N, N)$ misses to account for the initialization of the three matrices.

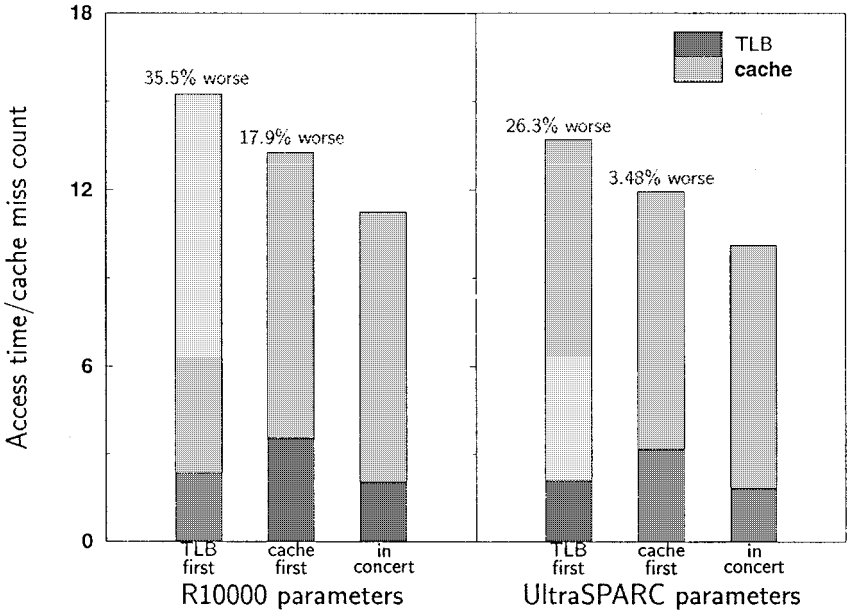


Fig. 6. Simulated data access times for the three tiling strategies using R10000 parameters and UltraSPARC parameter. Tiling for TLB and cache in concert does best.

have 17.9% and 35.5% higher data access times than an in-concert strategy (3.48% and 26.3% for the UltraSPARC). The cache-first tiling produces many TLB misses while the TLB-first tiling has many cache misses. The in-concert strategy actually does at least as well for TLB than a TLB-first strategy (and likewise for cache) because the "first" strategies are handicapped by the need to subtile to avoid thrashing. The cache-first optimization ran into problems by making tiles too wide for TLB; the TLB-first approach ran into cache problems by making tiles too high for cache.

We have also performed preliminary full-processor simulation studies using **sim-outorder** from SimpleScalar 2.0. The simulations indicate, not surprisingly for matrix multiply, that the three tiling strategies perform similarly. Each strategy ensured no thrashing in TLB and cache. Therefore, other factors, before negligible, now dominate cache and TLB effects: branch mispredictions, number of loads and stores and number of branches. This does not imply that in-concert TLB and cache tuning is always unnecessary. Rather, it may be unimportant for applications like matrix multiplication that are compute-bound in the absence of thrashing.

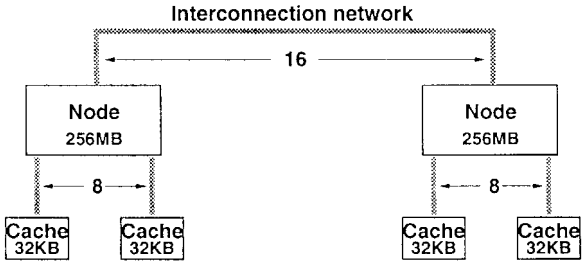


Fig. 7. Model of a sixteen-node message passing computer, where each node has eight processors, each with a 32 kilobyte cache, that share a common memory of 256 megabytes.

3.3. Multiple Levels of Parallelism

Finally, we consider optimizing for two levels of parallelism. Our example shows that optimizations driven by single-level cost functions can result in 50% slower execution than in-concert optimization.

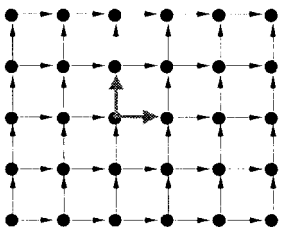
We use a computer similar to a SGI Power Challenge Array for our example of a machine with multiple levels of parallelism. This system has sixteen nodes connected with a high-speed network. Each node contains eight processors (each with a 32 kilobyte cache) and a 256 megabyte shared memory. Figure 7 depicts this architecture.

For this target architecture, we explore various tiling strategies of a common four-point stencil problem. Figure 8 shows the code and a portion of its iteration space. Iteration (i, j) is dependent on values from the two earlier iterations $(i-1, j)$ and $(i, j-1)$.

We consider a restricted set of possible tilings as shown in Fig. 9. In particular, we consider tilings that correspond to a block distribution of the columns of array A to the processors. The iteration space is divided into

```
for i=1 to T
    for j=1 to S
        A(i,j)=(A(i,j-1)+A(i-1,j)+
                +A(i,j+1)+A(i+1,j))*.25
```

(a)



(b)

Fig. 8. (a) Code; and (b) a portion of its iteration space. The stencil of the loop is highlighted.

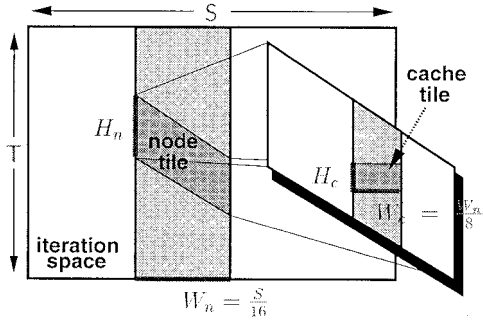


Fig. 9. Tiling for multilevel parallelism. The iteration space is partitioned into 16 stacks of node tiles, and each node tile is further partitioned into 8 stacks of cache tiles.

sixteen vertical swaths, one corresponding to each node of the computer. Each swath is partitioned into potentially non-rectangular *node tiles* of height H_n. Parallelogram-shaped tiles correspond to rectangular tilings after a unimodular transformation[6] of the iteration space. After the iterations in a given node tile are executed, a message containing the values on the right-land boundary is sent to the next node. This allows the receiving node to begin executing the tile that needs these values.

Each node tile is further partitioned into eight vertical swaths, one for each processor, and these are partitioned into *cache tiles* of height H_c.

We now develop a performance model to guide tiling choices. We begin by considering a simple loop. If each iteration of the loop takes time E_0 then our model of the execution time E_1 for the entire loop is

$$E_1 = N_1 E_0 + O_1$$

where N_1 is the number of iterations and O_1 is the overhead of initializing and terminating the loop. The body of the loop might be a single step of the computation, or it might be a tile. We make the assumption that the execution time of a partial tile is proportional to its area, so if the loop executes a stack of two-dimensional tiles, the number of tiles in the stack is the area of the stack divided by the area of a full tile.

The execution for a doubly-nested loop E_2 executed sequentially by a single processor is given by applying this formula twice:

$$E_2 = N_2 E_1 + O_2 = N_2 N_1 E_0 + N_2 O_1 + O_2 \tag{3.1}$$

Now consider instead a parallel outer loop, where a set of processors each execute a stack of tiles. If there are no dependences between the tiles

and all stacks take the same time, then the total execution time E_2 is the time of any one stack plus the parallel loop overhead:

$$E_2 = E_1 + O'_2 = N_1 E_0 + O_1 + O'_2 = N_1 E_0 + O_2$$

Now consider the case where there are tile dependences. It has been shown by Högstedt *et al.* that this can be done by including another term, the *idle time* I_2, which is the time that last processor spends waiting for data from other processors. [In Ref. 31, idle time is actually defined for any processor as the time it spends idle waiting for data from other processors plus the time it is idle after it has completed its work, before the last-completing processor has finished. In our case, the idle times for all processors are the same. The formula used here also assumes that communication cannot be overlapped by computation. The paper by Högstedt *et al.*[31] handles the general case as well.] The formula for idle time is:

$$I_2 = (N_2 - 1)(1 + r_2) E_0$$

where there are N_2 processors and r_2 is the *rise*, which reflects the amount of skewing of the tiles. Specifically, $r_2 = W_0/H_0(s_2 - s_1)$, where s_2 is the slope of the iteration space, s_1 is the slope of the tile, and H_0 and W_0 are the tile dimensions.[31] [Note: The formula for I_2 is valid when $r_2 \geqslant -1$; we won't consider $r_2 < -1$ since idle time cannot be negative.] When we incorporate the idle time into our formula for the parallel execution time of a set of stacks of tiles, we have:

$$E_2 = N_1 E_0 + O_2 + (N_2 - 1)(1 + r_2) E_0 \tag{3.2}$$

This discussion presents our model for execution time for a single-level tiling, where a two-dimensional iteration space is partitioned into a set of stacks of tiles. We now apply the formulas iteratively to the multiple levels of tiling for our example. We will use the subscripts i, n, and c to refer to features of the tilings of the entire iteration space, the node tiles, and the cache tiles, respectively. The formula are:

$$E_i = N_i E_n + O_i + (P_i - 1)(1 + r_i) E_n \tag{3.3}$$

$$E_n = N_n E_c + O_n + (P_n - 1)(1 + r_n) E_c \tag{3.4}$$

where N_i is the number of node tiles in a stack (corresponding to N_1 in formula Eq. (3.2)), P_i, is the number of nodes (corresponding to N_2 in formula Eq. (3.2)), N_n is the number of cache tiles in a stack (corresponding to N_1 in formula Eq. (3.2)), and P_n is the number of processors per node [corresponding to N_2 in formula Eq. (3.2)].

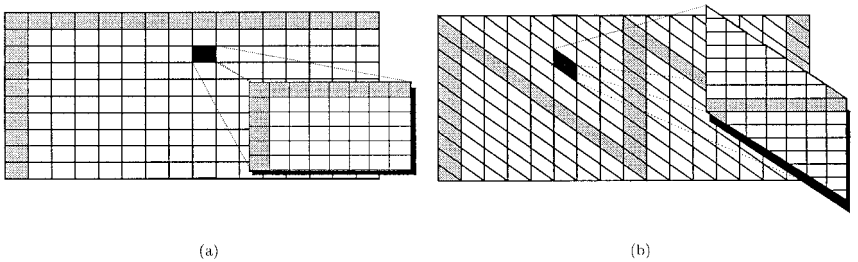


Fig. 10. Two tilings of the iteration space which differ in their choices of rise. In (a) $r_n = r_i = 0$; and (b) $r_n < 0$ and $r_i > 0$.[31] defines rise as a relationship between the shape of the iteration space and the shape of the tiles. Notice that decreasing the rise at a particular level of parallelism reduces the idle time due to a shorter critical path of dependent tiles at that level. The highlighted tiles in each tiling corresponds to these critical paths.

Figure 10 illustrates the multi-level tilings and the significance of rise. Note that the critical path of the tiling of the node tile by cache tiles in Fig. 10a (where the rise $r_n = 0$) is longer than the corresponding critical path in Fig. 10b (where $r_n = -1$). Intuitively, the idle time is the time required to execute the critical path minus the execution time for the last stack of tiles. Thus, the parallelogram-shaped node tile has less idle time. There is an interesting tradeoff between idle times at the two levels of parallelism. The tiling of the iteration space into node tiles in Fig. 10a has a smaller idle time than the corresponding tiling in Fig. 10b, whereas the tilings of node tiles into cache tiles have the reverse relationship.

A single-level tiling is determined by three parameters, the height of the individual tiles, their width, and the rise. For the two-level tiling of our example, there are the six parameters, H_n, W_n, r_i, H_c, W_c, and r_n.

Given T and S, the height and width of the full iteration space, and our architectural parameters $P_n = 16$ and $P_c = 8$, we know that $W_n = S/16$, $W_c = S/(16 \times 8)$, $N_i = T/H_n$, and $N_n = H_n/H_c$. To model the execution time of a cache tile, E_c, we somewhat arbitrarily assume that executing an iteration takes 5 cycles. This cost is intended to model both the computation time and the amortized cost of cache misses. [In this model, we don't count the number of cache misses, but instead adopt the rule that a cache tile must be small enough to fit comfortably in cache. We interpret "comfortably" to mean that the portion of the A matrix corresponding to a cache tile may be only 85% of the cache capacity, and assume the data values are 4 bytes each.] Using Formula (3.1) we get:

$$E_c = 5W_cH_c + W_cO_c + O'_c$$

where, O_c is the per-column loop overhead of the cache tile, and O'_c is the additional (nonloop) overhead of the cache tile.

Summarizing and rewriting all of our formulas in terms of the unknown tile parameters (r_i, H_n, r_n, and H_c) yields:

$$E_i = T/H_n E_n + O_i + 15(1 + r_i) E_n$$

$$E_n = (H_n/H_c) E_c + O_n + 7(1 + r_n) E_c$$

$$E_c = 5(S/128) H_c + (S/128) O_c + O'_c$$

We henceforth ignore the top-level overhead O_i since it is a (typically small) constant added to all execution times independent of the tiling choice.

Given this discussion, there are four unspecified parameters, r_i, H_n, r_n, and H_c. We will only consider rectangular cache tiles (i.e., not parallelograms), since using a positive slope for a tile would violate the dependences of the iteration space, and there is no advantage in using a negatively-sloped cache tile. [Note: If we were concerned with instruction-level parallelism in this example, then there might be an advantage to nonrectangular cache tiles.[32]] With the slope of the cache tile fixed, once r_i is chosen the value of r_n is completely determined (and vice versa). Thus, a multi-level tiling is given by a choice of values for three *tiling parameters*.

We consider four tiling strategies for the loop shown in Fig. 8. The first three strategies make single-level tiling choices in a level-by-level manner. These strategies choose the tiling parameters H, W, and r at one level at a time, making estimates of execution times that depend on tiling choices that haven't yet been made. We describe the three single-level strategies, **bottom-up**, **top-down** and the **flattened**, later. The fourth strategy *In-concert* uses a multi-level cost function to determine the three tiling parameters.

Bottom-up: The **bottom-up** strategy begins by minimizing $E_n/H_n W_n$. We minimize execution time per iteration point, instead of E_n, to avoid the trivial optimal value of $H_n = 0$. The only tiling parameters that affect $E_n/H_n W_n$ are H_c and r_n. We then choose the remaining node tile parameter H_n to minimize the formula for E_i, subject to the "fits-in" constraints $H_c W_c \leqslant 0.85 C_c$ and $H_n \geqslant H_c$.

Top-down: In the **top-down** strategy, we start by first choosing the tiling of the iteration space into node tiles. To do so in a single-level manner, we estimate E_n and choose the optimal values of H_n and r_i. The estimate we use assumes a node tiles takes 5 cycles per iteration point per processor, plus node tile overhead and an estimate of the processor idle time based on the largest tile that fits comfortably in cache. [The actual formula we used is $E_n = 5H_n S/128 + O_n + 8 \times 0.85 C_c$.] Having chosen H_n and r_i, we then choose the remaining cache tile parameter H_c, subject to the same fits-in constraints, to minimize formula in Eq. (3.4).

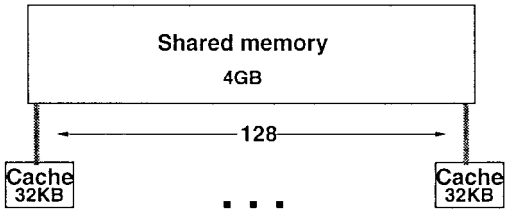


Fig. 11. The model for the architecture of the flattened approach.

Flattened: In this strategy we flatten the two levels of parallelism of our computer model into one. We choose the optimal one-level tiling of the iteration space for the flattened model, use the resulting tile height for both H_n and H_c of the original model. Finally, we choose the optimal values of the rises r_i and r_c to minimize formula in Eq. (3.3).

The **flattened** model has 128 caches sharing a large memory as in Fig. 11. We model the overhead in the **flattened** model by the weighted average of the cache-tile and node-tile overheads. This gives the formula for E_f, the execution time in the **flattened** model:

$$E_f = T/H_c E_c + O_i + 127(1 + r_f) E_c$$

$$E_c = 5(S/128) H_c + (S/128) O_c + O'_c + O_n/8$$

The rise r_f must be zero since the cache tiles and iteration space are both rectangular. We choose the optimal value for H_c subject to the fits-in constraint to minimize E_f. Proceeding according to strategy described earlier, we then set $H_n = H_c$, and find the optimal rises.

In-concert: The **in-concert** strategy makes the optimal simultaneous choice of the three tiling parameters to minimize the total execution time E_i.

Which is best? We evaluated the four tiling strategies using the performance model described earlier. As the **in-concert** strategy optimized this performance model, it is guaranteed to be the best in these evaluations.

We compare the predicted execution times of the four tiling strategies in two ways: fixed overheads with varying problem size and fixed problem size with varying overheads. Table V and Fig. 12a show the results for the former; Table VI and Fig. 12b show the results for the latter. The tables also include the tiling parameter choices for each of the strategies. In both figures, the y-axis shows the predicted total execution time relative to the **in-concert** strategy, which is represented by the line $y = 0$. In Fig. 12a the x-axis represents varying problem sizes using fixed values of O_c, O'_c and O_n. In Fig. 12b the problem size is fixed at $S = T = 8192$ (the smallest problem size represented in Fig. 12a) for varying values of O_n. The dashed

Table V. Choices of the Tiling Parameters H_n, H_c, r_i and r_n when Tiling Fig. 8 with $O_n = 1000$, $O_c = 10$ and $O'_c = 225$

$T \times S$	Strategy	H_n	H_c	r_i	r_n	Predicted exec. time
8192×8192	top-down	300	11	0.00	0.00	6.39 Mcycles
	bottom-up	101	101	8.00	−1.00	7.39
	flattened	40	40	8.00	−1.00	4.98
	in-concert	41	14	2.73	−1.00	4.26
4096×16384	top-down	144	7	0.00	0.00	7.20
	bottom-up	46	46	8.00	−1.00	7.16
	flattened	26	26	8.00	−1.00	5.60
	in-concert	21	9	3.43	−1.00	4.69
16384×8192	top-down	424	13	0.00	0.00	10.7
	bottom-up	101	101	8.00	−1.00	10.2
	flattened	57	57	8.00	−1.00	8.49
	in-concert	58	19	2.62	−1.00	7.48
8192×16384	top-down	203	8	0.00	0.00	11.9
	bottom-up	46	46	8.00	−1.00	10.0
	flattened	36	36	8.00	−1.00	9.26
	in-concert	29	13	3.59	−1.00	8.05
16384×16384	top-down	288	10	0.00	0.00	20.4
	bottom-up	47	46	9.95	−1.00	15.7
	flattened	46	46	8.00	−1.00	15.7
	in-concert	41	18	3.51	−1.00	14.3
8192×32768	top-down	131	6	0.00	0.00	23.5
	bottom-up	22	19	6.91	−1.00	15.9
	flattened	19	19	8.00	−1.00	15.9
	in-concert	21	12	4.57	−1.00	15.5
32768×16384	top-down	407	12	0.00	0.00	36.0
	bottom-up	62	46	5.94	−1.00	27.0
	flattened	46	46	8.00	−1.00	27.1
	in-concert	58	25	3.45	−1.00	26.3
16384×32768	top-down	185	8	0.00	0.00	40.8
	bottom-up	30	19	5.07	−1.00	27.9
	flattened	19	19	8.00	−1.00	28.0
	in-concert	29	17	4.69	−1.00	27.9
49152×16384	top-down	498	13	0.00	0.00	50.8
	bottom-up	74	46	4.97	−1.00	38.3
	flattened	46	46	8.00	−1.00	38.4
	in-concert	72	31	3.44	−1.00	37.9
16384×49152	top-down	135	6	0.00	0.00	62.9
	bottom-up	22	10	3.64	−1.00	42.5
	flattened	10	10	8.00	−1.00	43.0
	in-concert	22	10	3.64	−1.00	42.5
65536×16384	top-down	575	14	0.00	0.00	65.0
	bottom-up	84	46	4.42	−1.00	49.5
	flattened	46	46	8.00	−1.00	49.8
	in-concert	83	36	3.47	−1.00	49.3
32768×32768	top-down	262	9	0.00	0.00	72.6
	bottom-up	41	19	3.71	−1.00	51.7
	flattened	19	19	8.00	−1.00	52.3
	in-concert	41	19	3.71	−1.00	51.7
16384×65536	top-down	101	5	0.00	0.00	87.7
	bottom-up	18	6	2.67	−1.00	60.9
	flattened	6	6	8.00	−1.00	62.2
	in-concert	18	6	2.67	−1.00	60.9

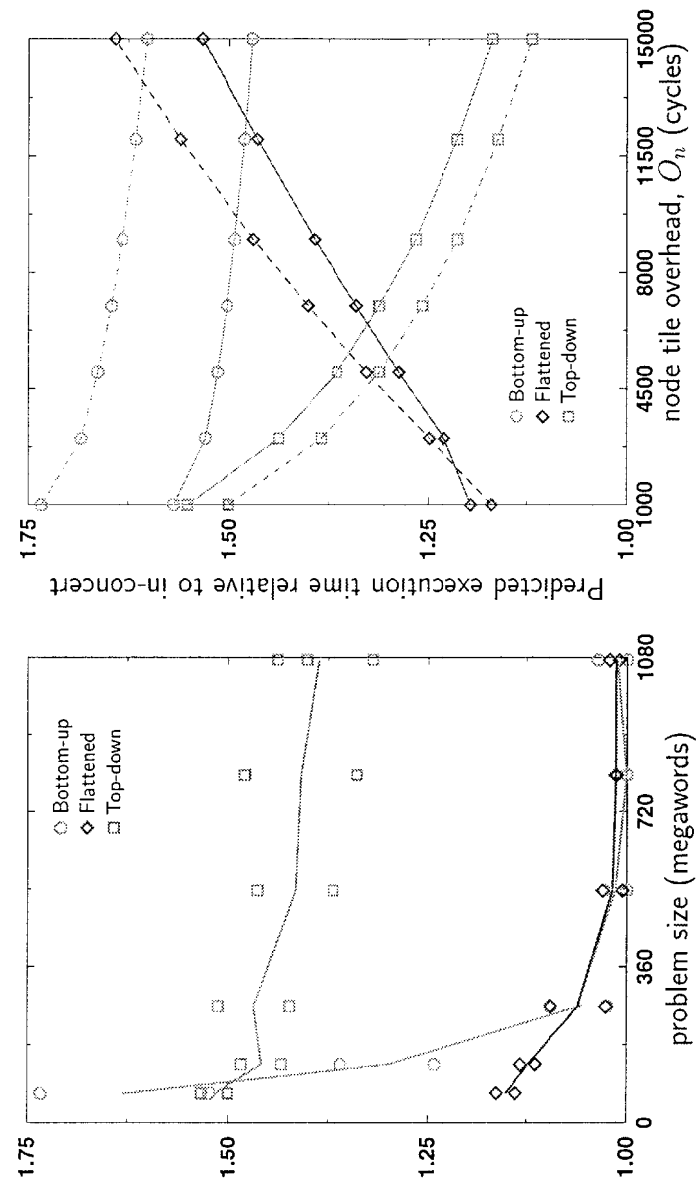


Fig. 12. Exactly how much worse than **in-concert** and the relative ranking of the three single-level strategies depends on the size of the problem (a) and node-tile and cache-tile overheads (b). In (a), for each problem size we tried several different $T \times S$ configurations (Table V shows the tiling choices for selected datapoints). The dashed lines in (b) use cache tile overheads twice as large as in the solid lines. (Tables VI and VII shows the tiling choices for selected datapoints.)

Table VI. Choices of the Tiling Parameters H_n, H_c, r_i and r_n when Tiling Fig. 8 with $S = T = 8192$, $O_c = 10$ and $O'_c = 225$

O_n	Strategy	H	H_c	r_i	r_n	Predicted exec. time (Mcycles)
1000	top-down	300	11	0.00	0.00	6.39
	bottom-up	101	101	8.00	−1.00	7.39
	flattened	40	40	8.00	−1.00	4.98
	in-concert	41	14	2.73	−1.00	4.26
3000	top-down	305	11	0.00	0.00	6.42
	bottom-up	111	101	7.28	−1.00	7.81
	flattened	45	45	8.00	−1.00	5.79
	in-concert	72	14	1.56	−1.00	4.64
5000	top-down	311	11	0.00	0.00	6.43
	bottom-up	143	101	5.65	−1.00	8.16
	flattened	49	49	8.00	−1.00	6.51
	in-concert	92	14	1.22	−1.00	4.91
7000	top-down	316	11	0.00	0.00	6.45
	bottom-up	170	101	4.75	−1.00	8.45
	flattened	53	53	8.00	−1.00	7.19
	in-concert	109	14	1.03	−1.00	5.13
9000	top-down	322	11	0.00	0.00	6.47
	bottom-up	192	101	4.21	−1.00	8.71
	flattened	57	57	8.00	−1.00	7.83
	in-concert	124	14	0.90	−1.00	5.33
12000	top-down	329	11	0.00	0.00	6.51
	bottom-up	192	101	3.64	−1.00	9.05
	flattened	62	62	8.00	−1.00	8.74
	in-concert	143	14	0.78	−1.00	5.60
15000	top-down	337	11	0.00	0.00	6.53
	bottom-up	248	101	3.26	−1.00	9.35
	flattened	66	66	8.00	−1.00	9.58
	in-concert	160	14	0.70	−1.00	5.84

lines in Fig. 12b use cache tile overheads O_c and O'_c twice as large as in the solid lines.

The second best strategy varies with the problem size and the size of the overheads, O_c, O'_c and O_n. As shown in Fig. 12a, the **bottom-up** strategy does very well for large problems, but as the problem size decreases it can be over 70% slower than **in-concert**. **Top-down** is 30–50% slower, and only improves slightly with larger problems. The **flattened** approach appears to be closest to **in-concert** for the overhead values used in Fig. 12a. As shown in Fig. 12b the **bottom-up** and **flattened** approaches degrade with increased values of O_c and O'_c. The **flattened** approach is also the only strategy that degrades with increased values of O_n.

Table VII. Choices of Tiling Parameters H_n, H_c, r_i and r_n when Tiling Fig. 8 with $S = T = 8192$, $O_c = 20$ and $O'_c = 450$

O_n	Strategy	H_n	H_c	r_i	r_n	Predicted exec. time (Mcycles)
1000	top-down	300	15	0.00	0.00	7.49
	bottom-up	101	101	8.00	−1.00	7.57
	flattened	55	55	8.00	−1.00	5.77
	in-concert	41	19	3.71	−1.00	4.83
3000	top-down	305	15	0.00	0.00	7.52
	bottom-up	110	101	7.35	−1.00	8.00
	flattened	58	58	8.00	−1.00	6.43
	in-concert	72	19	2.11	−1.00	5.23
5000	top-down	311	15	0.00	0.00	7.52
	bottom-up	142	101	5.69	−1.00	8.36
	flattened	62	62	8.00	−1.00	7.10
	in-concert	92	19	1.65	−1.00	5.52
7000	top-down	316	16	0.00	0.00	7.54
	bottom-up	168	101	4.81	−1.00	8.65
	flattened	65	65	8.00	−1.00	7.71
	in-concert	109	19	1.39	−1.00	5.75
9000	top-down	322	16	0.00	0.00	7.55
	bottom-up	190	101	4.25	−1.00	8.91
	flattened	68	68	8.00	−1.00	8.30
	in-concert	124	19	1.23	−1.00	5.96
12000	top-down	329	16	0.00	0.00	7.58
	bottom-up	219	101	3.69	−1.00	9.25
	flattened	72	72	8.00	−1.00	9.15
	in-concert	143	19	1.06	−1.00	6.25
15000	top-down	337	16	0.00	0.00	7.60
	bottom-up	245	101	3.30	−1.00	9.56
	flattened	76	76	8.00	−1.00	9.97
	in-concert	160	19	0.95	−1.00	6.50

These numbers only take parallelism into account; there is also a complicated interaction between parallelism, locality and other overheads. For instance, the worst performing cache tile for parallelism is rectangular, but this shape might have the least loop overhead.

4. CONCLUSIONS

Simple cost functions guide simple situations. As the architectural landscape grows increasingly complex, simple cost functions will no longer suffice to guide program optimizations.

We have shown that single-level cost functions may not optimally guide tiling decisions. Instead, an optimizing compiler should have a more *global* perspective; an optimization with a seemingly single-level goal may in fact be better regarded as multi-level.

We conjecture that one-level techniques suffice in the presence of a chained nest of caches (that is, ignoring parallelism and TLBs). Prior work concentrating on this scenario should optimize well. We further conjecture that, in the presence of two or more of the architectural features presented here, independent optimizations for the features (even using multi-level cost functions) is not the best strategy.

ACKNOWLEDGMENTS

We would like to thank Jim Demmel of UC Berkeley for his constructive criticisms on a draft of this paper.

REFERENCES

1. Michael E. Wolf and Monica S. Lam, A data locality optimizing algorithm, *Progr. Lang. Design Implementation* (1991).
2. Steve Carr and Ken Kennedy, Compiler blockability of numerical algorithms, *J. Supercomputing*, pp. 114–124 (November 1992).
3. Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng, Compiler optimizations for improving data locality, *Sixth Int'l. Conf. Archit. Support Progr. Lang. Oper. Syst.*, San Jose, California, Oct. 1994.
4. Steve Carr and Ken Kennedy, Improving the ratio of memory operations to floatingpoint operations in loops, *Trans. Progr. Lang. Syst.* **16**(6):1768–1810 (November 1994).
5. Corinne Ancourt and François Irigoin, Scanning polyhedra with DO loops, *Principles and Practice of Parallel Progr.*, pp. 39–50 (April 1991).
6. Michael E. Wolf and Monica S. Lam, A loop transformation theory and an algorithm to maximize parallelism, *IEEE Trans. Parallel Distrib. Syst.* **2**(4):452–471 (1991).
7. Paul Feautrier, Some efficient solutions to the affine scheduling problem, Part I, one-dimensional time, *IJPP* **21**(5):xx–xx (October 1992).
8. Wayne Kelly and William Pugh, A unifying framework for iteration reordering transformations, *IEEE First Int'l. Conf. Algorithms and Architectures for Parallel Processing* (April 1995).
9. Daniel Lavery and Wen-mei Hwu, Unrolling-based optimizations for modulo scheduling, 28*th Int'l. Symp. Microarchit.*, pp. 126–141 (December 1995).
10. Stephanie Coleman and Kathryn S. McKinley, Tile size selection using cache organization and data layout, *Progr. Lang. Design and Implementation* (June 1995).
11. Vivek Sarkar, Guang R. Gao, and Shaohua Han, Locality analysis for distributed shared-memory multiprocessors, *Lang. Compilers for Parallel Computing* (1996).
12. Dennis Gannon and Ko-Yang Wang, *Applying AI Techniques to Program Optimization for Parallel Computers*, Chap. 12, McGraw Hill Co. (1989).
13. Michael E. Wolf, Dror Maydan, and Ding-Kai Chen, Combining loop transformations considering caches and scheduling, 29*th Int'l. Symp. Microarchit.* (December 1996).

14. Michael J. Wolfe, Iteration space tiling for memory hierarchies, *Parallel Processing for Sci. Comput.*, pp. 357–361 (1987).
15. J. Ramanujam and P. Sadayappan, Tiling multidimensional iteration spaces for nonshared memory machines, *Supercomputing* (November 1991).
16. David A. Padua and Michael J. Wolfe, Advanced compiler optimizations for supercomputers, *Commun. ACM* **29**(12):1184–1201 (December 1986).
17. Dennis Gannon, William Jalby, and Kyle Gallivan, Strategies for cache and local memory management by global program transformation, *J. Parallel and Distrib. Comput.*, Vol. 5, No. 5 (October 1988).
18. François Irigoin and Rémi Triolet, Supernode partitioning, *Principles of Progr. Lang.*, pp. 319–328 (January 1988).
19. Michael J. Wolfe, More iteration space tiling, *Supercomputing*, pp. 655–664 (1989).
20. Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf, The cache performance and optimizations of blocked algorithms, *ASPLOS-IV*, Palo Alto, California (April 1991).
21. Utpal Banerjee, Unimodular transformations of double loops, in *Progr. Lang. Compilers for Parallel Computing*, Irvine, California (August 1990).
22. Ken Kennedy and Kathryn S. McKinley, Optimizing for parallelism and data locality, *Int'l. Conf. Supercomputing* (July 1992).
23. Jeanne Ferrante, Vivek Sarkar, and Wedy Thrash, On estimating and enhancing cache effectiveness, *Lang. Compilers for Parallel Computing* (1991).
24. Anant Agarwal, David Kranz, and Venkat Natarajan, Automatic partitioning of parallel loops and data arrays for distributed shared memory multiprocessors, *Int'l. Conf. Parallel Computing* (1993).
25. Vivek Sarkar and Radhika Thekkath, A general framework for iteration-reordering loop transformations, Technical Summary, *Progr. Lang. Design and Implementation* (1992).
26. Steve Carr, Combining optimization for cache and instruction-level parallelism, *PACT* '96, pp. 238–247 (1996).
27. Ken Kennedy and Kathryn S. McKinley, Maximizing loop parallelism and improving data locality via loop fusion and distribution, *Lang. Compilers for Parallel Computing* (1993).
28. Jeff Bilmes, Krste Asanović, Chee-Whye Chin, and Jim Demmel, Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology, *Int'l. Conf. Supercomputing* (1997).
29. Larry Carter, Jeanne Ferrante, Susan Flynn Hummel, Bowen Alpern, and Kang Su Gatlin, Hierarchical tiling: A methodology for high performance, Technical Report CS96-508, UCSD, Department of Computer Science and Engineering (November 1996).
30. Doug Burger and Todd Austin, The SimpleScalar architectural research tool set, Version 2.0, http://www.cs.wisc.edu/m̃scalar/simplescalar.html
31. Karin Högstedt, Larry Carter, and Jeanne Ferrante, Determining the idle time of a tiling, *Principles of Progr. Lang.* (1997).
32. Larry Carter, Jeanne Ferrante, and S. Flynn Hummel, Hierarchical tiling for improved superscalar performance, *Int'l. Parallel Processing Symp.* (April 1995).