Exploiting Wavefront Parallelism on Large-Scale Shared-Memory Multiprocessors

Naraig Manjikian, Member, IEEE, and Tarek S. Abdelrahman, Member, IEEE

Abstract—Wavefront parallelism, in which parallelism is limited to hyperplanes in an iteration space, can arise when compilers apply tiling to loop nests to enhance locality. Previous approaches for scheduling wavefront parallelism focused on maximizing parallelism, balancing workloads, and reducing synchronization. In this paper, we show that on large-scale shared-memory multiprocessors, locality is a crucial factor. We make the distinction between intratile and intertile locality and show that as the number of processors grows, intertile locality becomes more important. We consider and experimentally evaluate existing strategies for scheduling wavefront parallelism. We show that dynamic self-scheduling can be efficiently used on a small number of processors, but performs poorly at large scale because it does not enhance intertile locality. By contrast, static scheduling strategies enhance intertile locality for small tiles, maintaining parallelism and resulting in better performance at large scale. Results from a Convex SPP1000 multiprocessor demonstrate the importance of taking intertile locality into account. Static scheduling outperforms dynamic self-scheduling by a factor of up to 2.3 on 30 processors.

Index Terms—High-performance compilers, wavefront parallelism, cache locality, locality-enhancing loop transformations, tiling, large-scale shared-memory multiprocessors.

1 INTRODUCTION

Large-scale shared-memory multiprocessors are increasingly used as platforms for high-performance computing [6]. Several commercial and research systems of this type have been developed, including the HP/Convex SPP1000 [10], the SGI/Cray Origin 2000 [25], the Stanford FLASH [13], and the University of Toronto NUMAChine [12]. The architecture of large-scale shared-memory multiprocessors supports coherent shared memory in hardware. However, the shared-memory is physically distributed to attain scalability, as shown in Fig. 1. As a result, memory accesses are nonuniform. Access latency for remote memory is considerably higher than for local memory. Caches are relied upon to mitigate the latency of both local and remote accesses. Consequently, attaining high levels of performance necessitates both effectively exploiting parallelism and effectively utilizing the caches.

Application programs that benefit from high-performance computing often consist of loop nests that reuse large data arrays. The total amount of the data typically exceeds available cache capacity. Consequently, data that is loaded once into the cache is often replaced by other data before it is used again. The resulting cache misses to reload data reduce cache effectiveness and lead to poor performance. To address this problem, high-performance compilers employ loop transformations to reorder the computation performed in loop nests and increase the likelihood of retaining data in the cache between uses. Examples of such locality-enhancing loop transformations include permutation, fusion, distribution, and tiling [3].

However, locality-enhancing transformations can also adversely affect available parallelism in a program. For example, applying loop permutation may move an outer parallel loop to an inner loop level, which increases synchronization overhead [3]. A more serious effect is the loss of parallelism. For example, applying a loop fusion to a sequence of parallel loops may introduce new dependences that force the resulting loop to be executed sequentially [19]. Indeed, one study shows that commercial compilers that automate loop transformations for locality enhancement can result in poor overall performance in some applications by favoring locality over parallelism [7]. Such conflicts between parallelism and locality must be addressed to achieve scaling performance on large-scale multiprocessors. This paper addresses these conflicts for the specific case of tiling.

Tiling is a common transformation used to enhance temporal locality in loop nests [10], [17], [27]. In many cases, tiling must first be enabled by a transformation known as loop skewing [29]. The effect of skewing is to limit parallelism among tiles to hyperplanes, called wavefronts, in the tiled iteration space. Only tiles in the same wavefront may be executed in parallel. The skewed loops—an example of DOACROSS loops—must be scheduled such that a tile in a wavefront is not executed until the execution of tiles on which it depends in earlier wavefronts is complete.

Previous work on scheduling DOACROSS loops, or the execution of wavefront parallelism, has focused on maximizing available parallelism, balancing the workload, and reducing synchronization (see Section 6 for literature...
review). For example, dynamic self-scheduling [29] is used to balance workload and maximize parallelism. Similarly, static strip scheduling [9] is proposed to reduce synchronization overhead. In this paper, we show that, in addition to the above factors, locality is an important consideration on large-scale multiprocessors. We make the distinction between intratile locality that results from the reuse of data within one tile, and intertile locality, which results from the reuse of data between tiles. We show that as the tile size becomes smaller, which is often necessary when the number of processors is large, intertile locality becomes more important. We compare dynamic and static strategies on the basis of their ability to exploit intertile reuse in addition to run time overhead and synchronization requirements. We show that for a small number of processors, wavefront parallelism can be efficiently exploited using dynamic self-scheduling with a large tile size. Such a strategy enhances intratile locality, but does not necessarily enhance intertile locality, resulting in poor performance for a large number of processors. By contrast, we show that static scheduling exploits intertile reuse and performs better when the number of processors is large and smaller tiles become necessary to provide sufficient parallelism.

The remainder of this paper is organized as follows: Section 2 gives background on tiling loop nests and wavefront parallelism. Section 3 considers data reuse in tiled loop nests that require loop skewing. Section 4 describes the three scheduling strategies considered in this paper and analytically evaluates their benefits. Section 5 provides a comparative experimental evaluation of the three scheduling strategies on a large-scale HP/Convex SPP1000 multiprocessor for a number of applications. Section 6 reviews related work. Finally, Section 7 gives concluding remarks.

2 BACKGROUND

2.1 Tiling

Tiling combines strip-mining with loop permutation to enhance temporal locality [3, 28]. Strip-mining a loop introduces a new control loop to iterate between the original loop bounds in steps of $B$ (the strip size), as shown in Fig. 2b. The control loop is then permuted to the outermost level, as shown in Fig. 2c. This permutation changes the original order of the computation to make iterations that reuse the same element $a[i]$ execute successively. Although Fig. 2 illustrates one-dimensional tiling, where only one loop is strip-mined and permuted, multidimensional tiling is more common.

The legality of tiling is dictated by dependences. Strip-mining is always legal. However, permutation of the control loop to the outermost level may violate dependences, making tiling illegal. In such cases, loop skewing [3] may be used to enable legal permutation, as shown in Fig. 3. Loop skewing is always legal since it preserves the order in which iterations are executed.

The SOR loop, in Fig. 4a, is used to illustrate the necessity of loop skewing to enable tiling. This loop nest can benefit from tiling because the outermost loop carries the dependence $(1,0,0)$ that reflects temporal reuse of $a[i,j]$ carried by loop $t$. The complete set of dependence distance vectors for this loop nest is: $\{(1,0,0), (1,1,0), (1,0,1), (0,1,0), (0,0,1)\}$. Unfortunately, the tiling of loops $i$ and $j$ to exploit the temporal reuse is not legal because of the $-1$ that would prevent permutation after strip-mining. Loop skewing must be applied, as shown in Fig. 4b, in order to remove the negative elements in the distance vectors. Both inner loops $i$ and $j$ are skewed by one iteration with respect to loop $t$, resulting in the transformed distance vectors: $\{(1,1,1), (1,0,1), (1,1,0), (0,1,0), (0,0,1)\}$. The loop nest can then be tiled legally by first strip-mining the skewed $i$ and $j$ loops by a factor of $B$, as shown in Fig. 4c, and then by permuting the resulting $ii$ and $jj$ control loops to the outermost level, as in Fig. 4d.

2.2 Wavefront Parallelism

Although loop skewing enables tiling to exploit outer-loop reuse, it also transforms dependences in a manner that limits the degree of parallelism in the resulting tiled loop nest. Consider the distance vector $(t,j,i) = (1,0,0)$ from the set of dependences for the original loop nest of SOR. Skewing both inner loops transforms the original

---

**Fig. 1.** Large-scale shared-memory multiprocessor architecture.

**Fig. 2.** Tiling a loop nest with strip-mining and loop permutation to exploit temporal reuse. (a) Original loop nest. (b) After strip-mining. (c) After permutation.
vector into \((t, j, i) = (1, 1, 1)\). Now, strip-mining results in \((t, jj, ji, ii) = (1, B, 1, B, 1)\), and permutation finally produces \((jj, ii, t, j, i) = (B, B, 1, 1, 1)\). The resulting outermost loop now carries a dependence as a result of skewing the original inner loops. Since all of the resulting vector components are nonzero, permutation of any other loop to the outermost position also results in a loop-carried dependence that prevents parallel execution.

Similar skewing transformations that target the remaining distance vectors for the SOR loop nest introduce additional loop-carried dependences in the outer loops after tiling (although these are redundant in relation to the primary dependence discussed above). Hence, both \(ii\) and \(jj\) loops in the tiled loop nest of Fig. 4d carry dependences. These loop-carried dependences are represented graphically by the arrows in Fig. 5. This figure is a two-dimensional representation of the five-dimensional iteration space of the tiled loop nest. Each square corresponds to an iteration \((jj, ii)\) from the outer loops, and represents a \(B \times B \times T\) tile of iterations from the original \(jj\), and \(t\) loops. Since the two outer loops carry dependences, they are not parallelizable.

However, there does exist exploitable parallelism along the diagonal wavefronts shown in Fig. 5. Tiles within each wavefront are independent from one another and may be executed in parallel, although the wavefronts must be executed in proper sequence to satisfy the loop-carried dependences.

### 2.3 Exploiting Wavefront Parallelism: DOALL vs. DOACROSS

There are two general approaches for exploiting wavefront parallelism. The first is to apply a wavefronting transformation to obtain one or more inner DOALL loops [28]. This transformation corresponds to applying additional loop skewing at the outer loop levels to align independent tiles in each wavefront with corresponding DOALL loop iterations. Inner loops that execute iterations within a tile are left intact since each tile is executed in its entirety by one processor. For the SOR example, applying additional skewing to the tiled iteration space shown in Fig. 5 yields the iteration space shown in Fig. 6.

The alternative approach for exploiting wavefront parallelism is to treat the two outer loops as DOACROSS loops and introduce explicit synchronization between the dependent tiles. This approach avoids global synchronization and effectively utilizes idle processors by allowing portions of different wavefronts to proceed concurrently, albeit local synchronization is now required between tiles. Since the DOACROSS approach provides the opportunity for improved processor utilization, it will be assumed in the remainder of this paper for scheduling the execution of tiled loop nests.

### 3 Data Reuse in Tiled Loop Nests

#### 3.1 Intratile and Intertile Reuse

The data reuse in the transformed loop nest after tiling may be categorized as intratile or intertile reuse. Intratile reuse

---

**Fig. 3. Example of loop skewing.** (a) Original loop nest. (b) Skewing inner loop with respect to outer loop.

**Fig. 4. Steps in tiling to exploit outer loop reuse in the SOR loop nest.** (a) Original SOR loop nest. (b) Skewing inner two loops. (c) strip-mining inner two loops. (d) Permuting control loops.
results from capturing the reuse in the original outer loop within a single tile. In the tiled loop nest, data referenced in each tile is ideally loaded only once into the cache, then reused from the cache for locality within the same tile. However, when loop skewing is required to enable tiling, the data access patterns in the original loop nest are modified. When the skewed loop nest is tiled, there is still reuse of data within tiles, but the modified data access patterns also result in reuse between different tiles. Hence, there is also intertile reuse.

These two categories are illustrated in Fig. 7 for the SOR loop nest. The original iteration and data spaces are shown in Fig. 7a. With loop skewing and tiling, successive iterations of the original outer loop that are executed within the same tile access overlapping regions of the array, as shown in Fig. 7b. This constitutes intertile reuse. However, iterations from adjacent tiles also access overlapping regions in the data space as a result of loop skewing, as shown in Fig. 7c, and it is this overlap between tiles that results in intertile reuse. Note that skewing the extent of intertile reuse is considerably more than a small number of data elements at tile boundaries. Elements accessed in the interiors of adjacent tiles are reused as well.

\[
\text{do } j = 2.2^*(N-1+T)-B,B \\
\text{doall } i = \max(2,j-(N-1+T)+1), \min(j,j(N-1+T)),B \\
\text{do } t = 1:T \\
\text{do } j = \max(j-j,2+1), \min(j+j-B-1,N-1+T) \\
\text{do } i = \max(i,2+1), \min(i+j+B-1,N-1+T) \\
\text{a}[i-t,j-t] = (a[i,j]+a[i+1-t,j-t]+a[i-1-t,j-t] +a[i-t+1-t]+a[i-t,j-1-t]) / 5
\]

Fig. 5. Dependences and wavefronts.

Fig. 6. Exploiting parallelism with inner DOALL loops. (a) Inner DOALL loop from wavefronting transformation. (b) Wavefront parallelism in tiled iteration space.

Fig. 7. Data reuse in a tiled loop nest that requires skewing. (a) Iteration and data spaces for original SOR loop nest. (b) Skewed data access patterns within each tile. (c) Overlapping array regions swept by adjacent tiles.

Intratile reuse is converted to intertile locality in the cache of a processor if the reused data remains cached during the execution of the tile. When adjacent tiles are executed by the same processor, and data in the overlapping regions for those tiles is retained in the cache between tiles, intertile reuse is converted to intertile locality. That is, data in the overlapping regions is loaded only once into the cache, then reused from the cache not only within the same tile for intratile locality, but also in adjacent tiles. On the other hand, when adjacent tiles are executed by different processors, cache misses are incurred by each processor to load all the data referenced within each tile, including the data in the overlapping regions. In this case, there is no intertile reuse and the opportunity to convert the reuse into locality is lost.

Fig. 7 illustrates the skewing and tiling of a three-dimensional loop nest to exploit temporal reuse of a two-dimensional array. The discussion of the intertile reuse that results from loop skewing applies equally for tiling of higher-dimensional loop nests.

3.2 Tile Size, Parallelism, and Locality

The tile size has a significant impact on the performance of a tiled loop nest because it determines both the degree of parallelism and the extent to which locality is exploited. A smaller tile size increases the number of wavefronts and, consequently, the number of independent tiles in each wavefront. Hence, the degree of parallelism increases with smaller tile sizes.

The impact of tile size on intratile and intertile locality is illustrated in Fig. 8. The shaded regions represent the data accessed by each tile, as in Fig. 7c. The overlapping regions correspond to the intersection of the data accessed by
adjacent tiles. For a given number of iterations in the original outer loop, the amount of data in the overlapping regions is relatively small compared to the total amount of data accessed by the tile when the tile size is large. Consequently, a large tile size enhances intratile locality and diminishes the impact of intertile locality. By contrast, for the same number of iterations and a small tile size, the amount of data in the overlapping regions is a much larger fraction of the total amount of data accessed by the tile. Hence, a small tile size increases the importance of intertile locality.

4 Scheduling Strategies for Wavefront Parallelism

This section compares three scheduling strategies—namely dynamic self-scheduling, static cyclic scheduling, and static block scheduling—for exploiting wavefront parallelism in tiled loop nests when the outer loops are treated as DOACROSS loops. The strategies are compared on the bases of run time overhead, synchronization requirements, and locality enhancement for tiled loop nests with wavefront parallelism. Although the scheduling strategies discussed in this section will be illustrated in a tile space of two dimensions (where each component tile is three-dimensional) the discussion applies equally to higher-dimensional tile spaces.

4.1 Dynamic Self-Scheduling

In normal dynamic self-scheduling of DOALL loops, processors obtain iterations in some arbitrary order from a shared work pool. In DOACROSS dynamic scheduling, iterations in the work pool represent individual tiles and explicit synchronization is needed to enforce dependences between tiles in different wavefronts. Prior to executing a tile, interprocessor synchronization is required to ensure that tiles in the preceding wavefront have been executed.

Dynamic self-scheduling is adequate for exploiting wavefront parallelism on small-scale shared-memory multiprocessors. A large tile size generally provides an adequate degree of parallelism. Intratile locality is enhanced because a large tile size captures most of the reuse within a single tile, and intertile locality has little impact on performance.

However, dynamic self-scheduling is not appropriate for large-scale shared-memory multiprocessors, which requires a relatively smaller tile size to result in sufficient parallelism. A small tile size reduces intratile locality and places greater importance on intertile locality. Dynamic self-scheduling is not likely to enhance intertile locality since tiles are assigned arbitrarily to idle processors. Furthermore, cache misses that result from the reduced intertile locality are likely to be incurred for remote, rather than local, memory due to the arbitrary assignment of tiles to processors. The performance degradation resulting from these misses may be significant.

4.2 Static Cyclic Scheduling

In static scheduling for DOALL loops, the assignment of iterations to a processor is determined a priori and remains fixed. Static cyclic scheduling for DOACROSS loops assigns rows of horizontally-adjacent tiles to the same processor, as shown in Fig. 9. Intertile reuse along each row of tiles is exploited by a processor. The cyclic mapping of rows of tiles to processors distributes the workload in each wavefront evenly among processors to fully exploit available parallelism, but requires explicit synchronization between dependent tiles.

Static cyclic scheduling improves over dynamic self-scheduling in three ways. First, cyclic scheduling enhances intertile locality for horizontally-adjacent tiles by statically assigning them to the same processor, whereas dynamic self-scheduling does not necessarily exploit any intertile reuse due to the arbitrary assignment of tiles. Second, interprocessor synchronization to enforce loop-carried dependences is required only for vertically-adjacent tiles, since horizontally-adjacent tiles are executed in the correct order by the same processor. Third, the scheduling overhead is reduced since the assignment of tiles to processors is determined statically. However, cyclic scheduling still requires synchronization for each tile to enforce dependences, and not all of the intertile reuse is exploited.

4.3 Static Block Scheduling

Static block scheduling for the DOACROSS loop iterations in a tiled loop nest assigns contiguous blocks of tiles to the same processor, as shown in Fig. 10. In this manner, all of the intertile reuse within a block of horizontally- and vertically-adjacent tiles is exploited by one processor to enhance intertile locality. However, the tiles must be executed in an order that respects the dependences. The available parallelism on each wavefront is not exploited efficiently for the original wavefronts shown in Fig. 10a because some of the processors are left idle for the few initial and few final wavefronts. The block assignment of
tiles to processors precludes the use of additional processors even when there is a sufficient number of tiles that can be executed in parallel. Consequently, it takes longer for all processors to become active, and it takes longer for execution to complete.

Hence, we modify block scheduling by rotating wavefronts such that the number of independent tiles in the largest wavefront is exactly equal to the number of processors, as shown in Fig. 10b. This rotation corresponds to the selection of a different scheduling vector. The scheduling vector is (1, 1) for the original wavefronts in Fig. 10a. In fact, this is the optimal scheduling vector to minimize parallel execution time for dynamic and cyclic scheduling (neglecting the impact of locality). The scheduling vector for the modified wavefronts in Fig. 10b is given by \( \left\lceil \frac{(N + T)}{(B \cdot P)} \right\rceil, 1 \), where \( N + T \) is the number of iterations (with skewing) along the blocked iteration space dimension, \( B \) is the tile size in that dimension, and \( P \) is the number of processors. The new scheduling vector preserves the loop-carried dependences in block scheduling, but it also reduces the time before all processors become active in parallel execution, which in turn reduces the parallel execution time.

Static block scheduling improves over both dynamic and cyclic scheduling in two ways. First, block scheduling exploits all tile reuse, except at block boundaries. Second, interprocessor synchronization to enforce loop-carried dependences is required only for tiles on block boundaries. No synchronization is required for adjacent interior tiles because they are executed in the correct order by the same processor. Scheduling overhead is also reduced since the assignment of tiles to processors is determined statically.

### 4.4 Comparison of Scheduling Strategies

The scheduling strategies are compared on the bases of run time overhead, synchronization, and intertile locality enhancement. The comparison is summarized in Table 1.

#### 4.4.1 Runtime Overhead for Scheduling

Dynamic self-scheduling incurs run time overhead in order to assign tiles from the work pool to processors as they become idle. The overhead has two components. The first is maintaining the set of iterations to be assigned, and only two scheduling counters are required for this purpose. The first identifies the current wavefront, and the second identifies the most recent tile assigned in that wavefront. The second component arises from contention for access to these counters. By contrast, static cyclic and static block scheduling incur no run time overhead for scheduling since the assignment of tiles to processors is determined a priori.

#### 4.4.2 Synchronization Requirements

DOACROSS loops require explicit synchronization between dependent iterations. To provide the necessary synchronization in tiled loop nests with wavefront parallelism, each row of tiles is assigned a synchronization counter that is incremented as each tile in the row is completed. These counters track the execution of wavefronts across the tiled iteration space. No more than one tile can be executed in each row at any given time. Hence, no locking is needed to increment the counters.

Dynamic self-scheduling requires synchronization for both horizontally- and vertically-adjacent tiles from different wavefronts. Prior to executing a tile in a given row, a processor must wait until the counters for both the given row and the preceding row have indicated that the dependences have been satisfied. By contrast, static cyclic scheduling only requires checking the counter for the preceding row because all tiles in the given row are executed in order by the same processor.

Finally, static block scheduling requires interprocessor synchronization only for vertically-adjacent tiles on block boundaries. As a result, the number of synchronization counters required is equal to the number of processors, rather than the number of rows. The counters between two blocks must only be checked before executing tiles at the block boundary.
Fig. 11. Number of elements within a tile. (a) Elements per tile. (b) Elements per overlap region.

4.4.3 Locality Enhancement

The extent of intertile locality enhancement for each scheduling strategy is shown in Table 1. The importance of enhancing intertile locality, when skewing is required, can be demonstrated by estimating the latency for cache hits and misses that occur during the execution of a single tile. For a tile size $B \times B$, and $T$ iterations in the original outer loop of the loop nest being tiled, the total number of data accesses to the cache within each tile is given by $B^2T$. This number is conservative because register locality reduces the number of cache accesses. Each access to the cache has a latency of $C$ clock cycles. Some fraction of these references miss in the cache and incur the additional cache miss latency $M$. For dynamic self-scheduling, there is no intertile locality, and in the worst case, misses are incurred for all data elements accessed for the first time within the tile. The number of such elements is given by $B^2 + (2B - 1)(T - 1)$, as shown in Fig. 11a. This number must then be divided by $L$ (the cache line size) to arrive at an estimate for the number of cache misses. The total latency in clock cycles for memory accesses is then given by multiplying by the cache miss latency $M$. Finally, the total memory access latency, including the cache accesses, is given by $B^2TC + (B^2 + (2B - 1)(T - 1))(M/L)$. To measure the extent of locality enhancement for different values of $B$ and $T$, it is useful to express the fraction $f$ of the total memory access latency per tile that is due to cache misses, which is given by

$$f_{\text{calc}} = \frac{(B^2 + (2B - 1)(T - 1))(M/L)}{B^2TC + (B^2 + (2B - 1)(T - 1))(M/L)}.$$  

A similar derivation can be made for cyclic and block scheduling. Because there is intertile locality for adjacent tiles, fewer misses are incurred per tile. The reduction in the number of misses is determined by the number of elements in one or both of the overlap regions shown in Fig. 11b. Once again, the fraction of the latency due to misses can be determined. Hence,

$$f_{\text{calc}} = \frac{(B^2 + BT - 2B - T + 1)(M/L)}{B^2TC + (B^2 + BT - 2B - T + 1)(M/L)}$$

and

$$f_{\text{calc}} = \frac{(B^2 - 2B + 1)(M/L)}{B^2TC + (B^2 - 2B + 1)(M/L)}.$$  

Fig. 12 plots the fraction $f$ for different tile sizes $B$ and different values of $T$. The cache line size is $L = 4$ elements. The cache access latency is $C = 1$ clock cycle. And the cache miss latency is $M = 50$ clock cycles. As $T$ increases, $f$ decreases for all three strategies because reuse carried by the original outer loop is captured within each tile through intratile locality. However, $f$ decreases far more rapidly for block scheduling. This is because block scheduling benefits from enhancing intertile locality by reducing the number of cache misses by an amount proportional to the overlap regions in Fig. 11b. Furthermore, for a given value of $T$, $f$ is further reduced with a smaller tile size for block scheduling because intertile locality is more critical when the tile size is small (see Fig. 8). By contrast, for a given value of $T$, $f$ increases when the tile size is reduced for both dynamic and cyclic scheduling. This is because dynamic and cyclic scheduling do not enhance intertile locality to the same extent for small tile sizes as block scheduling.

5 Experimental Evaluation

This section presents experimental results obtained on a Hewlett-Packard/Convex SPP1000 multiprocessor [10] to evaluate the scheduling strategies described in this paper. The SPP1000 multiprocessor consists of up to 16 hypernodes, each containing 8 processors with a crossbar connection to 512 Mbytes of common memory, as shown in Fig. 13. The crossbar provides uniform access to the local memory for processors within a hypernode. Each processor is a superscalar Hewlett-Packard PA7100 RISC microprocessor running at 100 MHz with separate 1-Mbyte instruction and data caches. The cache access latency is 1 clock cycle.

Fig. 12. Fraction of miss latency per tile.

Fig. 13. Architecture of the Convex SPP1000.
or 10 nsec, and the cache line size is 32 bytes. 

Hypernodes are connected together with the Coherent Toroidal Interconnect (CTI), a system of rings based on the SCI standard interconnect, clocked at 250 MHz. The CTI permits processors to access memory in any hypernode through coherent global shared memory. Cache misses to retrieve data from the local hypernode memory incur a nominal latency of 40 cycles, or 400 nsec. However, misses to retrieve data from remote hypernode memory through the CTI incur a latency of approximately 200 cycles, or 2μsec. A unique feature of the Convex SPP1000 is the CTI cache, which is a portion of the memory in each hypernode reserved for caching data from other hypernodes in order to reduce the effective memory latency for remote memory accesses. Our experiments were conducted on a 32-processor Convex SPP1000 consisting of 4 hypernodes. The CTI cache size in each hypernode is 16 Mbytes. Two processors are reserved for system use, leaving 30 processors available for experimentation.

5.1 Results for SOR

We first report results obtained for the SOR loop nest with array size $1,024 \times 1,024$, where each array element is an 8-byte floating point value. The number of iterations in the original outer loop is $T = 40$. We use tile sizes of $32 \times 32$, $16 \times 16$, and $8 \times 8$. Larger tile sizes are not considered because they do not provide sufficient parallelism for a large number of processors. Fig. 14 shows the average number of cache misses and corresponding miss latencies per processor on 16 processors. Both the number of misses and the latencies are broken down into local and remote. Fig. 14a indicates that block scheduling incurs far fewer misses for a given tile size than dynamic or cyclic scheduling, which agrees with our observations in Section 4.4.3. The fraction of misses to remote memory is small for block and cyclic scheduling (4 percent and 5 percent respectively for a tile size of 8). This fraction is significantly larger for dynamic self-scheduling (27 percent for a tile size of 8). Hence, the impact of the remote misses on the total cache miss latency shown in Fig. 14b is more pronounced for dynamic self-scheduling. As the tile size is reduced, both the number of cache misses and the total miss latencies increase dramatically for both dynamic and cyclic scheduling. In particular, the resulting miss latency for dynamic self-scheduling with a tile size of 8 is 30 times larger than for block scheduling. This clearly demonstrates the detriment of failing to provide intertile locality when the tile size is small.

The effect of the cache behavior on execution time for the tiled SOR loop is shown in Fig. 15 for 16 and 30 processors. Static scheduling performs better than dynamic scheduling for a large number of processors, but only block scheduling improves consistently when the tile size is reduced to provide greater parallelism. Although cyclic scheduling
Fig. 16. Speedup for tiled SOR.

with an intermediate tile size may perform better than block scheduling, it is difficult to predict an optimal tile size for cyclic scheduling that achieves the appropriate balance between sufficient parallelism and sufficient locality. Hence, we focus on comparing block scheduling with dynamic scheduling for the largest and smallest tile sizes.

Finally, the parallel speedup of tiled SOR for various numbers of processors over the sequential untilted loop nest executed on a single processor is shown in Fig. 16. The speedup of block scheduling and dynamic self-scheduling are compared for the largest and smallest tile sizes. When the number of processors is eight or less, all memory accesses are confined within single a hypernode. Dynamic self-scheduling with a large tile size generates sufficient parallelism for the relatively small number of processors, and maximizes intratile locality. The larger tile size and the uniform memory access within a hypernode diminish the impact of intratile locality. Consequently, dynamic self-scheduling with the largest tile size performs the best. However, as the number of processors increases, a large tile size limits the speedup of dynamic self-scheduling due to insufficient parallelism. In addition, memory accesses span hypernodes and become nonuniform, which limits the speedup of dynamic self-scheduling, particularly when a smaller tile size is used to provide greater parallelism. Intratile locality is critical for small tile sizes, and dynamic self-scheduling does not exploit intratile locality. By contrast, block scheduling with a small tile size provides sufficient parallelism while enhancing intratile locality, improving the speedup by a factor of 1.4 over dynamic self-scheduling at 30 processors.

5.2 Results for Jacobi

The Jacobi loop nest is shown in Fig. 17. There is reuse between the inner two loop nests in addition to the reuse carried by the outer loop. Tiling requires fusion [3] of the inner two loop nests to produce a single loop nest. However, there are fusion-preventing dependences between the inner two loop nests and it is necessary to apply the shift-and-peel transformation [19] to enable legal fusion. Once a single loop nest is obtained with fusion, loop skewing is required to tile. It is necessary to skew the inner loops by two iterations with respect to the outer loop, rather than one as required for SOR. The loop nest is then tiled to exploit the reuse carried by the outer loop. The final code is not shown due to space limitations, but is similar to the tiled SOR loop nest.

Fig. 18 shows the average number of cache misses and corresponding miss latencies per processor for parallel execution of tiled Jacobi with the different scheduling strategies on 16 processors. The array sizes are $2,048 \times 2,048$, and the number of iterations in the original outer loop is $T = 10$. The number of misses and the latencies are broken down into local and remote. Block scheduling incurs the fewest cache misses as well as having the smallest fraction of remote misses. The cache latency for block scheduling is also the lowest. Dynamic self-scheduling incurs the greatest number of cache misses and a larger

Fig. 17. The Jacobi loop nest sequence.

Fig. 18. Cache misses for tiled Jacobi. (a) Average cache misses for 16 processors. (b) Average miss latency for 16 processors.
fraction of remote misses, which results in a dramatic increase in cache miss latency as the tile size is reduced.

Normalized execution times for tiled Jacobi on 16 and 30 processors are shown in Fig. 19. All execution times are normalized with respect to the parallel execution time for the original code. The parallel performance for the original code is indicative of the capabilities of the Convex native compiler because it would only identify and exploit outermost parallel loops as we have done for our baseline results. We then apply transformations that the native compiler does not have. The normalized execution time for fusion of the inner loops without tiling is also shown, since parallel execution of the fused loops is enabled by the shift-and-peel transformation. Once again, the results for tiling are similar to those obtained for SOR. The dramatic increase in execution time for dynamic self-scheduling correlates with the increase in the cache miss latency. Block scheduling with a small tile size performs far better. Fusion exploits reuse between the inner two loops, but tiling goes further to exploit the reuse carried by the outer loop. To ensure that the full benefit of tiling is realized, the tiled loop nest must be scheduled appropriately.

Finally, the parallel speedup of tiled Jacobi for various numbers of processors over the original code executed on a single processor is shown in Fig. 20. The speedup for block scheduling and dynamic self-scheduling is compared to the speedup from parallel execution of the original code (i.e., with two DOALL loops) and the fused version. The speedup for cyclic scheduling is not shown because its performance for small tile sizes is worse than block scheduling. Dynamic self-scheduling with a large tile size is only effective in the absence of remote memory accesses (i.e., when the number of processors is 8 or less). By contrast, block scheduling with a small tile size improves the speedup by a factor of 1.8 over dynamic self-scheduling at 30 processors, and consistently outperforms even the parallel versions of the original and fused code.

5.3 Results for LL18

The LL18 kernel from the Livermore Loops consists of three loop nests surrounded by an outer loop. A total of nine arrays are used, and there is reuse between the inner loop nests in addition to the reuse carried by the outer loop. Once again, the shift-and-peel transformation [19] is needed to enable legal fusion. It is then necessary to skew the inner loops by three iterations to tile.

The tiled LL18 loop nest is scheduled with the different strategies just as for SOR and Jacobi. Normalized execution times for 16 and 30 processors are shown in Fig. 21 for array sizes of 1024 x 1024 and T = 10 iterations in the original outer loop. The results are similar to those obtained for Jacobi. Fusion improves performance by exploiting reuse between the inner loop nests, but tiling with an appropriate scheduling strategy exploits all the reuse for the best performance. Once again, only block scheduling is successful in enhancing locality when the tile size is reduced to provide sufficient parallelism for a large number of processors. The speedups for LL18 shown in Fig. 22 also agree with the trends observed for Jacobi. Block scheduling improves the speedup at 30 processors by a factor of 2.3 over dynamic self-scheduling.

6 RELATED WORK

An extensive formal treatment of tiling is given by Wolf [28], based on earlier work by Porterfield [23], Irigoin and Tricolet [16], and Abu-Sufah et al. [1]. Wolf presents a theoretical framework that combines tiling with unimodular transformations, and also presents algorithms for deriving the skewing required to produce a fully-permutable loop nest for tiling. Desprez et al. [11] develop a theoretical model for determining idle time in tiled loops.
More recently, Son and Li [26] propose tiling techniques for imperfectly nested loops and show their effectiveness in the Panorama compiler. However, these works do not study the effects of loop skewing on data reuse, and make no distinction between intratile and intertile locality. Furthermore, experimental results are limited to small-scale multiprocessors with uniform memory access. By contrast, this paper demonstrates that intertile locality is crucial on large-scale multiprocessors that require the use of small tile sizes to provide sufficient parallelism on a large number of processors.

There exists a large body of work dealing with scheduling of parallel, or DOALL, loops on shared-memory multiprocessors. Many scheduling strategies such as static scheduling [3], self-scheduling [3], guided self-scheduling [22], and factoring [15] have been proposed to strike a balance between load balance and scheduling overhead. Some scheduling strategies, such as affinity-based scheduling [20], also consider memory locality for nonuniform memory access by attempting to distribute loop iterations in a manner that matches an a priori-known distribution of data. However, all these strategies address individual DOALL loops where there are no restrictions on the manner in which iterations are distributed and executed among multiple processors and, hence, do not distinguish between intratile and intertile data reuse for cache locality.

There is also a large body of work on scheduling DOACROSS loops. Chen and Yew [9] tackle the execution of DOACROSS loops with nonuniform dependences. They propose theoretical frameworks for “uniformizing” the dependences by determining uniform dependences that “cover” nonuniform ones. More relevant to our work, Chen and Yew [9] propose a static strip scheduling scheme for the execution of a DOACROSS loop. They use analysis to model the performance of their scheduling strategy, but are motivated by extracting parallelism and reducing synchronization rather than by exploiting locality. Furthermore, they conduct no experimental evaluation of their techniques. By contrast, we focus on locality, and conduct an experimental evaluation of scheduling strategies.

Chen et al. [8] propose an algorithm for run time parallelization of DOACROSS loops based on an inspector-executor paradigm. The compiler inserts the inspector and executor code into the loop. The inspector determines dependence relations among data accesses, and the executor uses this information to execute iterations in parallel when possible. They evaluate their algorithm on the Cedar multiprocessor. However, their work does not address cache locality. By contrast, we focus on cache locality and on waveform parallelism for which dependences allow static scheduling.

Li [18] discusses an affinity tiling algorithm aimed at enhancing spatial locality for reuse of cache lines at the boundaries of adjacent tiles. This is a different notion of intertile reuse than the one considered in this paper. Affinity tiling does not exploit temporal reuse carried by an outer loop. Reported performance improvements due to affinity tiling are limited. Loop permutation to enhance cache line spatial locality prior to affinity tiling accounts for most of the reported improvements. By contrast, tiling, as discussed in this paper, exploits temporal reuse in an inner loop within a loop nest, which is substantially more than the spatial reuse of cache lines at tile boundaries.

Hodzic and Shang [14] present an analysis for tiled loop nests with loop-carried dependences that require interprocessor communication on message-passing multiprocessors. They assume that communication startup cost is the dominant factor affecting performance and derive an optimal scheduling vector that minimizes execution time. By contrast, this paper considers shared-memory multiprocessors that must account for communication startup cost on processors with cache locality, rather than communication startup cost, having the greatest impact on performance.
We show that a suboptimal scheduling vector is required in order to enhance locality and provide the best performance.

Barua [4], Barua et al. [5] extend earlier work by Agarwal et al. [2] to reduce the amount of nonlocal communication and the number of cache misses on large-scale shared-memory multiprocessors. They partition parallel loops to minimize the number of data elements in the data tile accessed by a processor, and partition arrays across the physical memories in the system to reduce remote memory accesses. They use an iterative search supported by a cost model of cache and memory accesses to derive loop and data partitions across multiple loop nests. However, their framework is limited in that they only reduce the number of first-time or compulsory cache misses, which is not sufficient since cache locality is adversely affected by noncompulsory misses that are caused by failing to exploit data reuse. By contrast, our work aims specifically to exploit such data reuse and prevent the other cache misses that are ignored by their framework. Furthermore, data partitioning can only ensure that misses from a processor’s cache are satisfied from the local memory of the processor. By contrast, we exploit intratile and intertile reuse to reduce the number of cache misses, regardless of whether a miss is to local or to remote memory. In this respect, data partitioning is complementary to the scheduling strategies we examine. With data partitioning, the reduced number of cache misses that still occur with our approach may become local instead of remote.

In general, when starting with the same initial set of DOALL loops, our approach enhances cache locality considerably more than the approach of Barua et al. For example, their framework ignores data reuse across loop nests in a program [4, p. 24], [5]. Nonetheless, data reuse across loops provides significant opportunities for enhancing locality [21]. Our approach is to exploit such reuse using shift-and-peel [19], resulting in superior performance over their approach of partitioning the original loops and data, as shown by the results for the Jacobi example in Section 5. Furthermore, Barua et al. also rely only on the choice of loop partitions to optimize cache locality and only reduce first-time compulsory misses. This approach foregoes cache performance improvements through techniques such as tiling. Indeed, they report that their loop partitioning approach for cache locality does not provide significant improvements because the initial loop partitions derived by their approach are almost always the same as default ones in the absence of their approach [4, pp. 39-42]. By contrast, our approach uses tiling to exploit temporal reuse and can result in significant improvements, as shown with Jacobi in Section 5.

Rastello and Robert [24] also extend the work of Agarwal et al. [2] to determine the best tile shape to minimize the amount of data accessed, given a fixed tile size, and to more accurately quantify the amount of data accessed in a tile. However, their results are limited to estimates produced by their tile shape selection algorithm and they point out that further experimentation is required to assess the true value of their approach. By contrast, we provide experimental results that show the true impact of a selected tile shape and size on actual performance.

7 Concluding Remarks

Large-scale shared-memory multiprocessors have become increasingly viable platforms for high-performance computing. Faced with nonuniform memory access latencies in such multiprocessors, high-performance compilers must aggressively enhance locality as well as exploit parallelism in order to provide good performance. Tiling is a particularly effective locality-enhancing technique that exploits temporal data reuse carried by an outer loop of a loop nest. However, legal tiling often requires loop skewing, which results in loop-carried dependences that limit the available parallelism to wavefronts in the tiled iteration space.

In this paper, we considered scheduling of tiled loop nests to exploit wavefront parallelism on large-scale multiprocessors. We distinguished between intratile and intertile locality to evaluate the extent to which locality is enhanced, and we showed the impact of tile size selection on locality as well as parallelism. We evaluated existing dynamic and static scheduling strategies on the basis of their ability to exploit locality on large-scale multiprocessors in addition to parallelism, load balance, and synchronization. We showed that dynamic self-scheduling of tiles performs poorly for a large number of processors because it cannot enhance intertile locality with a small tile size, leading to poor performance. By contrast, static scheduling strategies enhance intertile locality and overcome the shortcomings of dynamic self-scheduling. Results of experiments conducted on a 30-processor HP/Convex SPP1000 with three representative applications indicate that static scheduling outperforms dynamic self-scheduling by enhancing intertile locality while providing sufficient parallelism for a large number of processors. Furthermore, the performance of static block scheduling consistently improves with reductions in tile size, unlike static cyclic or dynamic scheduling.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council of Canada and the Information Technology Research Centre of Ontario. Access to the HP/Convex SPP1000 multiprocessor was provided by the Center for Parallel Computing at the University of Michigan. The authors also thank the anonymous referees for their comments.

References


Naraig Manjikian (S’87-M’97) received the BASc degree (computer engineering) in 1991 and the MASc degree (electrical engineering) in 1992, both from the University of Waterloo. He received the PhD degree (electrical engineering) from the University of Toronto in 1997. From 1992 to 1997, he was also a participant in the NUMachine Multiprocessor Project at the University of Toronto. Since 1997, he has been an assistant professor in the Department of Electrical Engineering at Queen’s University, Kingston, Ontario. Dr. Manjikian’s research interests include computer architecture and multiprocessing, compilers for parallel systems, and applications of parallel processing. His current research projects are examining processor-memory integration and application-specific computing architectures for telecommunications. He is a member of the IEEE and the IEEE Computer Society.

Tarek S. Abdelrahman received the PhD degree in computer science and engineering from the University of Michigan at Ann Arbor in 1989. He is currently an associate professor of electrical and computer engineering at the University of Toronto. His current research interests are in compiler optimizations for parallelism and large-scale shared-memory multiprocessors. Dr. Abdelrahman is a member of the ACM and of the IEEE and the