Reducing Cache Conflicts by Partitioning and Privatizing Shared Arrays

Zhiyuan Li
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
E-mail: li@cs.purdue.edu

Abstract

Parallelizing compilers for shared-memory multiprocessors typically generate fork/join programs in which multiple threads may access different sections of shared arrays. Each thread tends to access discontiguous addresses due to the gap between sections which belong to different arrays. Such interarray discontiguity *often causes cache-set conflicts even when the private cache is large enough to hold the working set. A compiler technique, called* array partition and privatization *(or APP), is proposed to correct this problem. The APP technique partitions a shared array into sections, one for each computing thread, and then maps each section to the private stack of the corresponding thread. Interarray discontiguity is thus removed and the cache performance improved.*

This paper presents a compiler algorithm for APP and preliminary experimental results based on four benchmark programs running on an SGI Origin 2000 multiprocessor.

1. Introduction

In recent years, an increasing number of shared-memory multiprocessor systems become available in the commercial market. In order for a wide range of users to take advantage of the increasingly predominant multiprocessors, the state of the art of parallelizing compilers must continue to improve.

Memory latency is well recognized as a severe performance bottleneck and it will become more so in the foreseeable future. On modern commercial computers, the most important hardware technology for reducing the average memory latency is the cache. For programs whose working sets are large, the secondary off-chip cache is particularly important because it is larger than the primary on-chip cache, often by one or two orders of magnitude. On multiprocessors, efficient use of caches is even more important because the penalty for cache misses is greater. Unfortunately, the issue of efficient cache use is also much more complex than on uniprocessor machines.

Several types of cache misses are known to occur on shared-memory multiprocessors. *Coherence misses*, due to interference between shared variable references from different processors, have been studied extensively in the last decade. *Set-conflict misses* are caused by accessing different memory addresses which are mapped to the same cache set. Set-conflict misses are not unique to multiprocessors. However, parallel programs, especially compiler-generated parallel programs, may exhibit a particular memory-reference behavior, called *address discontiguity*, which can cause severe set-conflict misses.

Parallelizing compilers for shared-memory multiprocessors typically generate fork/join programs in which multiple threads may access different sections of shared arrays. If one thread accesses several discontiguous sections of a shared array, the thread is said to have *intraarray* address discontiguity. *Array permutation* has been proposed to remove intraarray discontiguity [3]. Address gaps can also exist between sections which belong to different arrays. In this case, we say that the thread has *interarray discontiguity*. The problem of interarray discontiguity has not been previously reported. However, in the experiments reported in this paper, this problem is found to cause severe cache-set conflicts and hence poor cache performance. We present a compiler technique, called *array partition and privatization* (or APP), to solve this problem. The APP technique partitions a shared array into sections, one for each computing thread. It then maps each section to the private stack of the corresponding thread. Interarray discontiguity is thus removed and the cache performance improved, resulting in shorter execution time.

One should note that message passing programs for distributed memory multicomputers normally do not suffer address discontiguity as shared-memory programs do. However, transforming a sequential program to a message passing parallel program requires domain decomposition throughout the whole program and it also requires precise array data flow analysis to ensure correct send/receive communication [1]. On the other hand, the APP approach can be applied to an arbitrary program region as long as it is beneficial. Moreover, memory-based data dependence analysis, which is considerably simpler than data flow analysis, is sufficient for inserting the necessary data communication and synchronization in the shared-memory code. Hence, we believe that the APP approach is more flexible than complete transformation of programs into message passing codes, and we expect it to be more applicable to a wide range of programs.

We organize the rest of the paper as follows. In Section 2, we present an example to illustrate the problem of interarray discontiguity and the idea of the APP technique. In Section 3, we present a compiler algorithm to automate the technique. We compare our work with related works in Section 4. In Section 5, we present preliminary experimental results which are collected from four benchmark programs executing on an SGI Origin-2000 multiprocessor. We conclude with Section 6.

2. Interarray Address Discontiguity

In this section, we use an example to illustrate how interarray address discontiguity can cause cache-set conflicts and how the APP (array partition and privatization) technique can eliminate such a problem.

Figure 1 shows an imperfectly-nested loop nest which resembles the kernel of the tomcatv program in the SPEC95 benchmark suite [22]. The SGI MIPSpro compiler [20] parallelizes the code into what is shown in Figure 2. The MIPSpro is representative of the most advanced commercial parallelizing compilers. In an attempt to reduce cache conflicts, MIPSpro performs *array padding* to change each array from the shape of (513,513) to the shape of (525,513). The OpenMP PARALLEL DO directives mark the two j loops as parallel loops. Suppose p threads are created, by default or by the user's specification, to execute the whole loop nest, then the 511 iterations of each j loop will be divided in p consecutive chunks and assigned to the p threads evenly. For convenience, we can assume that each thread is executed by a distinct processor. The data dependences in the loop nest prevent fusion of the two inner loop nests and the interchange of the sequential k loop with the inner loops. Notice that the two arrays, a and b, are marked as shared and that each array has a size of 2MB. On the other hand, the array sections accessed by different threads are disjoint except for the overlap of borderlines. Therefore, suppose each processor has a 2MB secondary cache, then $p \geq 2$ would in theory be sufficient for the working set of each thread to fit in the secondary cache. Unfortunately, it is not so in practice.

If the cache is virtual-indexed, or if the cache is real-

```
real*8 a(513,513),b(513,513)
do k = 1, 1000
  do j = 2, 512
    do i = 2, 512
       . = a(i-1,j-1) + a(i+1,j+1)
       . = b(i-1,j-1) + b(i+1,j+1)
    enddo
  enddo
  . . .
  do j = 2, 512
    do i = 2, 512
      a(i,j) = .
      b(i,j) = .
    enddo
  enddo
enddo
```

Figure 1. A sample loop nest.

```
real*8 a(525,513),b(525,513)
do k = 1, 1000
  !$OMP PARALLEL DO private(i,j), shared(a,b)
  do j = 2, 512
    do i = 2, 512
       . = a(i-1,j-1) + a(i+1,j+1)
       . = b(i-1,j-1) + b(i+1,j+1)
    enddo
  enddo
  . . .
  !$OMP PARALLEL DO private(i,j), shared(a,b)
  do j = 2, 512
    do i = 2, 512
      a(i,j) = .
      b(i,j) = .
    enddo
  enddo
enddo
```

Figure 2. The sample loop nest parallelized by MIPSpro.

indexed but the OS uses the *page coloring* page-mapping scheme [12] as on the SGI machines, then the virtual addresses of a(i,j) and b(i,j) will map to the same cache set. Suppose the secondary cache is direct mapped, then nearly every array reference will be a cache miss! (Caches with higher set associativity will have the same problem if the program contains more arrays.)

The wasteful utilization of the cache by this sample program is due to the address gaps between the array sections referenced on each processor. The idea of the APP technique is to partition each shared array into p sections, one for each thread. It then declares a thread-private copy for each array partition. The private array has the same number of rows but $1/p + 2$ columns. The extra two columns are used to store array elements accessed in the first and the last iterations assigned to the thread, which are mapped to the borderlines of the neighboring array partitions. Due to the data flow carried by the serial k loop, the borderline data must be transferred between the neighboring threads after each iteration of the k loop. The resulting code is shown in Figure 3. For clarity, the number of threads, p, is assumed to be statically determined, but it does not need to be. The OpenMP library function OMP_GET_THREAD_NUM() returns the thread identification number (between 0 to $p - 1$). The OpenMP PARALLEL and END PARALLEL directives define a parallel region which is executed by each of the p threads. The private arrays, `a_p` and `b_p`, are allocated on the private stack of each thread. Four new shared arrays, `a_top, a_bot, b_top, b_bot`, are used to transfer data between different threads. We call these arrays *buffer arrays*. Notice that the original shared arrays a and b do not need to be reshaped.

After the APP transformation, each thread has four private arrays as its working set and they are allocated consecutively on the private stack. Assuming $p \geq 2$, the private arrays will be mapped to different cache sets and they fit in a 2MB secondary cache throughout the computation of the whole loop nest. The set conflicts are thus removed.

```
real*8 a(513,513),b(513,513)
parameter(nchunk=512/p)
real*8 a_p(513,0:nchunk+1),b_p(513,0:nchunk+1)

real*8 a_top(513,p),a_bot(513,p),b_top(513,p),b_bot(513,p)
CALL OMP_SET_NUM_THREADS(p)
!$OMP PARALLEL private(i,j,k,jpr,jlo,jhi,a_p,b_p),
      shared(a,b,a_top,a_bot,b_top,b_bot)
jpr = OMP_GET_THREAD_NUM() + 1
 ! Code to copy shared arrays to private arrays omitted.
 ! Code to copy borderline shared array elements to buffer arrays omitted.
if (jpr.eq.1) then
  jlo = 2
else
  jlo = 1
endif
jhi = nchunk
do k = 1, 1000
 ! Code to copy buffer arrays to borderline private array elements omitted.
  do j = jlo, jhi
    do i = 2, 512
      . . .
    enddo
  enddo
  . . .
  do j = jlo, jhi
    do i = 2, 512
      . . .
    enddo
  enddo
 ! Code to copy borderline private array elements to buffer arrays omitted.
 ! BARRIER
enddo
 ! Code to copy private arrays to live shared arrays omitted .
!$OMP END PARALLEL
```

Figure 3. The sample loop nest parallelized with the APP technique.

3. A Compiler Algorithm for APP

In this section, we present a compiler algorithm for APP, using OpenMP as the parallel programming interface. Since most of current OpenMP implementations support single-level parallel regions only, in this paper we also consider single-level parallel loops.

In this paper, we consider applying APP to a program region, R, which

- has a single entry and a single exit,
- contains a number of single-level parallel loops, named J_1, J_2, $\ldots$, J_m respectively, which may be doall loops or reduction loops,
- does not contain procedure calls,
- does not contain I/O statements, and
- may contain serial loops and structured IF statements both outside and inside the parallel loops. To simplify the discussion, we assume that all J_i loops are embedded in the same serial loops in the program region and that there exist no serial loops between the J_i loops.

We assume that conventional array privatization [9, 15, 18, 24, 8] has already been performed for the parallel loops.

Algorithm: Simple-APP.

Input: (1) A program region, R, defined above. (2) A number of shared arrays referenced in R. Without the loss of

generality, consider just one of such arrays, A. (3) The number of threads, p, which participate in the computation in R.
Output: A parallel region, R', to be executed by each of the p threads in which the original shared-array references are replaced by private-array references.
Procedure:

- Let L_i and U_i be the lower and upper bound of J_i, respectively. Let $L = min(L_i)$ and $U = max(U_i)$ for all i. Let *nchunk* equal $(U - L + 1)/p$. (A fractional number is rounded to the nearest integer.) Let C_t be the iteration range assigned to thread t such that $C_0 = [L, L + nchunk]$, $C_1 = [L + nchunk + 1, L + 2 * nchunk]$, and so on. Let $C_t^i = C_t \cap [L_i, U_i]$, $0 \leq t \leq p$, $1 \leq i \leq m$.

- Compute MOD_t^i, the regular section of A which is modified within J_i in the range C_t^i. (For descriptions of regular sections, see [6, 10].) Compute USE_t^i, the regular section of A which is read within J_i in the range C_t^i. Let $REF_t^i = MOD_t^i \cup USE_t^i$. Let MOD_t be the union of MOD_t^i and USE_t be the union of USE_t^i for all $1 \leq i \leq m$. Let $REF_t = MOD_t \cup USE_t$. Determine the maximum size, s_d, for each dimension d of REF_t, $0 \leq t \leq p$.

- Declare a private array, A_p, for the new parallel region, R'. A_p has the bounds of (0:$s_d - 1$) in the d-th dimension.

- For each J_i, compute the address mapping between REF_t^i and A_p which is a translation ϕ_i: $\vec{j_i} \longrightarrow \vec{j_i} - \vec{l_i}$, where $\vec{l_i}$ is a constant vector.

- Replace the references to REF_t^i by references to A_p, using the translation function ϕ_i computed above to determine the array subscripts.

- Replace the loop bounds of J_i by $[1, nchunk] \cap [L_i, U_i]$. We denote the new loop by J_i'.

- All serial computations whose results are not live outside R remain unchanged within R' except that the modified variables are declared private to each thread.

- All serial computations whose results are live outside R will be performed by one thread only. Insert IF statements such that only thread 0 performs such computations.

- Apply *Procedure Synch()* below to insert code for data communication and synchronization.

- For each t, determine the A sections which are upward-exposed [9] to the outside of R and insert copy-in code to copy the exposed A elements to A_p, A_{top} and A_{bot} (see Procedure Synch() below).

- For each t, determine the modified A section which is live outside R (*c. f.* [14, 8]). Insert copy-out code to copy A_p back to A.

Procedure Synch()

- Compute data dependence distances between A references in different J_i loops. If there exists a dependence from J_{i_1} to J_{i_2} with a nonzero distance and no simple paths from J_{i_1} to J_{i_2} contain a back edge, then insert a *barrier* at the exit of J_{i_1}'. If there exists any simple path from J_{i_1} to J_{i_2} which contains a back edge, then insert a barrier at the end of the iteration of the innermost serial loop which contains the J_i' loops, $1 \leq j \leq m$.

- Of the dependence distances computed above, determine the maximum positive distance, D^+, and the minimum negative distance, D^-. (For example, in the sample code in Figure 1, $D^+ = 1$ and $D^- = -1$.) If there exist no dependences of positive (or negative) distances, then let $D^+ = 0$ (or $D^- = 0$). If either $D^+ > nchunk$ or $D^- < -nchunk$, then the Simple-APP Algorithm aborts. (As a matter of fact, unless the absolute values of both positive and negative distances are far smaller than *nchunk*, then it is not worth while to apply the APP technique, and the algorithm should abort.)

- Let L_t^i and U_t^i be the lower and the upper limit of C_t^i, respectively. Let $C_t^i \mid_{D^+} = [L_t^i, L_t^i + D^+ - 1]$ and $C_t^i \mid_{D^-} = [U_t^i + D^- + 1, U_t^i]$. Let $USE_t^i \mid_{D^+}$ be the regular section of A which is read within loop J_i in the range $C_t^i \mid_{D^+}$. Let $USE_t^i \mid_{D^-}$ be the regular section of A which is read within loop J_i in the range $C_t^i \mid_{D^-}$. Let $USE_t \mid_{D^+}$ be the union of $USE_t^i \mid_{D^+}$ for all loop J_i. Let $USE_t \mid_{D^-}$ be the union of $USE_t^i \mid_{D^-}$ for all loop J_i. Determine the maximum size, s_d^+, for each dimension d of $USE_t \mid_{D^+}$, and determine the maximum size, s_d^-, for each dimension d of $USE_t \mid_{D^-}$, $0 \leq t \leq p$.

- Declare two shared buffer arrays, $A_{top}(s_1^+, s_2^+, \ldots, s_m^+, p)$, which is mapped to $USE_t \mid_{D^+}$ and $A_{bot}(s_1^-, s_2^-, \ldots, s_m^-, p)$, which is mapped to $USE_t \mid_{D^-}$.

- If there exists a dependence from J_{i_1} to J_{i_2} with a positive distance, then insert code before J_{i_2}' to copy $A_{top}[\ldots, t]$ to A_p, $1 \leq t \leq p - 1$. Also, insert code after J_{i_1}' to copy A_p to $A_{top}[\ldots, t+1]$, $0 \leq t \leq p-2$. If there exists a dependence from J_{i_1} to J_{i_2} with a negative distance, then insert code before J_{i_2}' to copy $A_{bot}[\ldots, t]$ to A_p, $0 \leq t \leq p - 2$. Also, insert code after J_{i_1}' to copy A_p to $A_{bot}[\ldots, t-1]$, $1 \leq t \leq p-1$.

- For each recurrence loop J_i, declare private scalar variables to store the partial results computed in each thread and insert a critical section (using the OpenMP critical directive [17]) to update the shared recurrence variables.

The Simple-APP algorithm presented above can be improved in several obvious aspects. For example, the J_i loops can be *aligned*, via re-indexing, to reduce data dependences between different threads. Secondly, the number of writes to the buffer arrays and the number of barriers can be reduced by array reaching definition analysis. Similarly, the number of reads from the buffer arrays can be reduced. These analyses are nonetheless nontrivial. We leave them for our future work.

4. Related Works

Several compiler algorithms have been proposed to align parallel tasks with array data (see [2, 3, 13, 16], for examples). Where data dependence directions permit, loop fusion can be performed to reduce the number of barriers [25, 11]. In certain cases, barriers may also be removed without conducting loop fusion [23]. Array permutation [3] is a recently proposed data reorganization technique which permutes array dimensions in order to increase intraarray address contiguity. Subscripts in array references are permuted accordingly. Recent work by Chandra et al. [7] provides compiler support for programmer-specified data reshaping on CC-NUMA multiprocessors. Such compiler support aims to reduce page-level false sharing by eliminating intraarray address discontiguity. These techniques are complementary to the APP technique. With the APP technique, loop fusion, loop interchange and array permutation are all unnecessary for the removal of the interarray discontiguity. However, data-task alignment is performed before APP to insure that the J_i parallel loops are well aligned and hence the dependence distances between J_i loops are short. Barrier removal techniques [23] may be performed after the APP, although we have not experimented with such techniques. The SGI MIPSpro compiler performs array permutation and data-task alignment automatically. The APP technique is shown to outperform MIPSpro generated code in most configurations for the chosen benchmarks.

On machines with large real-indexed data caches, cache set conflicts may be reduced by the operating system which conducts *careful page mapping* [12]. *Page coloring* and *bin-hopping* are two best-known careful mapping techniques. Silicon Graphics and Microsoft have adopted page-coloring in the Irix and the NT operating systems, respectively. DEC has implemented bin-hopping in the OSF/1 system. With *Compiler-directed page coloring* (or CDPC), the OS takes array reference information from the compiler and it then chooses a smart page mapping to avoid bin conflicts [5]. Our experiments show the APP technique to significantly improve cache performance under the page-coloring scheme, but we have not experimented with other page mapping schemes.

The *array padding* technique modifies array dimensions in order to change the way array elements are mapped to the cache lines [4, 19]. Padding with a small offset may change the composition of individual cache-lines, and padding with a large offset may also change the virtual bin number. Padding patterns, however, are not always easy to determine based on array subscripts. They also depend on the cache size and the number of processors. Since the APP technique selects the size of each private array to be the size of the array partition plus the borderlines, it has a side effect of padding the array. In two of our four benchmarks, such padding causes a significant reduction in the cache misses even when a single thread is used for the computation. However, the important fact is that APP works well where existing padding algorithms do not perform effectively. The MIPSpro compiler performs array padding.

There exist extensive studies of loop tiling to improve cache reuses. No existing tiling algorithms can tile the imperfectly-nested parallel loops in the sample program. Neither can they tile the benchmarks used in our experiments. However, it is interesting to investigate more advanced tiling techniques to handle these benchmarks and their interactions with APP.

There exist rich literature of compiler techniques to transform a sequential program (possibly with data partitioning directives) into message passing parallel codes for distributed memory multicomputers. Automatic generation of efficient message passing is known to be extremely difficult for the compiler, partly due to the difficulties with choosing the message length and with the interleaving of send and receive library calls. Enormous efforts have been spent on reducing the overhead of message passing library calls. On the other hand, one should note that message passing programs for distributed memory multicomputers normally do not suffer address discontiguity as shared-memory programs do. However, transforming a sequential program to a message passing parallel program requires domain decomposition throughout the whole program. In comparison, the APP approach presented in this paper can be applied to an arbitrary program region so far as it is beneficial. Hence, the APP approach seems more flexible than complete transformation of programs into message passing codes. We expect the APP approach to be more applicable to a wide range of sequential programs.

5. Experiments

We have implemented a preliminary version of the simple-APP algorithm within the Panorama parallelizing compiler [9]. This version can handle relatively simple Fortran subroutines which do not contain subroutine calls in the target program regions. We applied the automated APP to four benchmark programs: sobel and kirsch are from an image processing benchmark suite provided by Computing Devices International (CDI), and tomcatv and swim are from the SPEC95 floating point benchmarks [22]. (The subroutine calls in swim are in-lined before applying APP.)

Program `tomcatv` is a mesh generation solver. The main data consists of seven arrays of 513 × 513 8-byte real numbers. Program `swim` (shallow water model) is a weather prediction program. The main data consists of 14 arrays of 513 × 513 4-byte real numbers. For $p \geq 4$ threads, the APP technique makes each thread's working set fit in the 4MB secondary cache. The main parallel loop nests in both programs are two-level deep. One sequential loop contains the parallel loops. The sequential loop iterates 750 times in `tomcatv` and 900 times in `swim`, respectively. For both programs, the SGI MIPSpro compiler parallelizes the correct loop level such that the data and the threads are well aligned. MIPSpro also performs array permutation and loop interchange for `tomcatv` in order for data-task alignment to work.

Programs `sobel` and `kirsch` implement the namesake algorithms. Each program contains three arrays of 1024 × 1024 4-byte integers. Each consists of a perfect nest of two doall loops. Only the outer doall loop is parallelized.

Both APP-transformed codes and the codes parallelized by SGI MIPSpro compiler (using vendor advised the "pfa -O3 -mp" switch) are executed on an SGI Origin-2000 multiprocessor [21] at NASA Ames. For `swim` and `tomcatv`, additional switches, as published by the vendor's SPEC report, are set when using the MIPSpro compiler to parallelize the codes. Jobs are submitted through the Portable Batch System (PBS) to claim exclusive access to a desired number of processors. Each processor contains a MIPS RISC R10K 64-bit CPU running at 250 MHz, a 32KB two-way set-associative on-chip instruction cache and a 32KB two-way set-associative on-chip data cache. Origin 2000 multiprocessors are shipped with 1MB or 4MB unified secondary caches which are two-way set-associative. This machine comes with an 4MB secondary cache for each processor. Due to the relatively modest problem sizes of our benchmarks, we have used up to 16 processors only.

Tables 1 through 4 show the secondary cache misses generated by different codes, measured by using the hardware counters through the SGI perfex utility. Where multiple processors are used, the number shows the sums over all processors. Figure 4 shows the speedup (over the sequential time) of the parallelized codes. Notice that, for `sobel` and `kirsch`, the APP technique reduces secondary cache misses dramatically even for $p = 1$. As mentioned in the last section, this is because the APP has a side effect of padding the private arrays, which eliminates cache set conflicts within the inner loop. However, we see that for $p = 1, 2$, the `sobel` program does not get speedup by APP. In this case, the overhead of copy-in and copy-out code seems to cancel the gain from reducing cache misses, but we are yet to investigate further.

Table 1. Secondary Cache Misses for tomcatv

Codes	1 proc.	2 proc.	4 proc.	8 proc.	16 proc.
MIPSpro	214M	200M	187M	161M	60M
APP	220M	108M	16M	4.7M	0.3M

Table 2. Secondary Cache Misses for swim

Codes	1 proc.	2 proc.	4 proc.	8 proc.	16 proc.
MIPSpro	167M	162M	157.M	148M	128M
APP	95M	95M	3.8M	0.9M	0.9M

We compared the timing results which we measured for MIPSpro-generated `tomcatv` and `swim` with the official SPEC results reported by the vendor. Only the results for 16 processors and one processor are published. Our results agree with those published results except for `swim` running on one processor, where we consistently measured 94 seconds vs. 200 seconds reported by the vendor.

The code parallelized by MIPSpro for `tomcatv` runs slower than the APP-transformed code, except in the case of 16 processors where both codes execute at roughly the same speed. Notice that the MIPSpro-parallelized code incurs significantly more secondary-cache misses. It seems that, for 16 processors, the cache invalidation overhead begins to dominate the performance. More investigation is needed to improve APP's performance in order to capitalize its low cache misses.

As we mentioned previously, the sequential time we measured for `swim` is less than half of what appears in the SPEC report. Using this base time, we calculated the speedup of MIPSpro-generated code for 16 processors over a single processor as 9.4. Note that we use the same sequential time to calculate the speedup of the APP-transformed code (about 11) and that of MIPSpro-generated code.

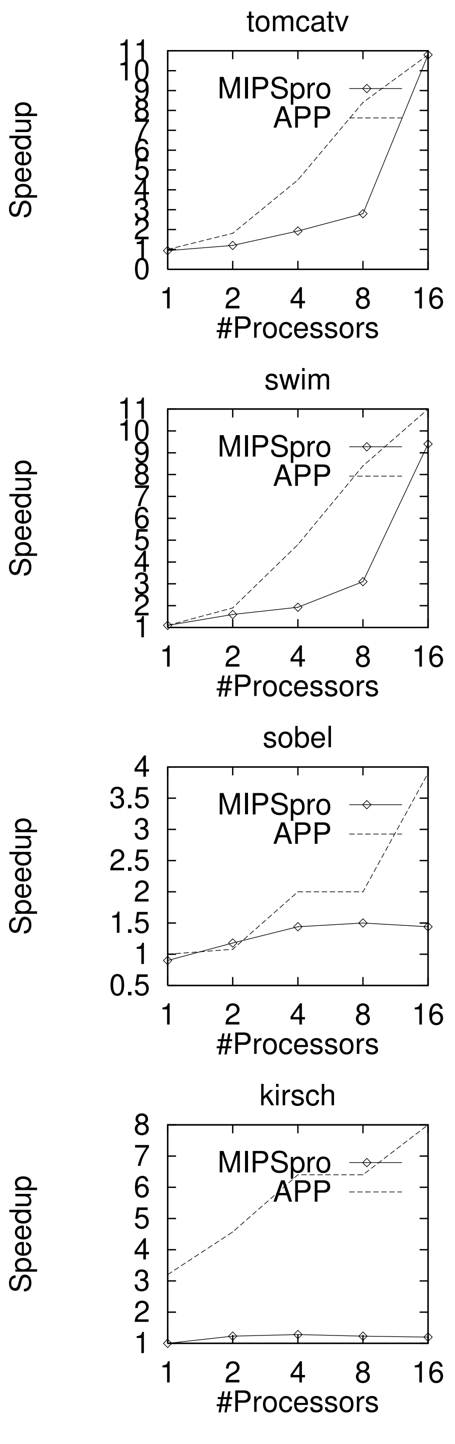

Figure 4. Speedup of different parallel versions of four benchmarks over the sequential code.

Table 3. Secondary Cache Misses for sobel

Codes	1 proc.	2 proc.	4 proc.	8 proc.	16 proc.
MIPSpro	6.7M	7.0M	7.0M	7.0M	7.0M
APP	0.5M	0.3M	0.3M	0.3M	0.3M

Table 4. Secondary Cache Misses for kirsch

Codes	1 proc.	2 proc.	4 proc.	8 proc.	16 proc.
MIPSpro	6.2M	6.2M	6.2M	6.2M	6.2M
APP	0.5M	0.4M	0.3M	0.3M	0.3M

6. Summary

In this paper, we have proposed the array partition and privatization (APP) technique which eliminates interarray address discontiguity in compiler generated parallel programs. The preliminary experiments show that the APP technique reduces cache misses significantly and the programs run considerably faster.

The program model for our simple-APP algorithm is rather simplistic. Our next step is to study how to extend the algorithm for program regions which contain subroutine calls and more complex control flows. More extensive experiments will be conducted afterwards. It is important to study the interaction between APP and other cache-performance enhancement techniques, such as padding and tiling. Moreover, a good cost model is needed in order to determine when the APP can guarantee a performance gain. As with any other cache-performance enhancement techniques, these issues are difficult but are important.

Acknowledgement

This work is sponsored in part by National Science Foundation through grants CCR-950254, MIP-9610379 and by Purdue Research Foundation. Computing Devices International provided the image processing benchmark suite used in our experiments. The AIMS group at NASA Ames provided an account on SGI Origin 2000 multiprocessors. The timing results are verified on a newly-installed SGI Origin 2000 in the Department of Computer Science, Purdue University, which is funded by National Science Foundation. These generous supports are gratefully acknowledged. The author also thanks the reviewers for their helpful comments.

References

[1] V. Adve and J. Mellor-Crummy. Using integer sets for data-parallel program analysis and optimization. In *SIGPLAN PLDI'98*, Montreal, Canada, June 1998.

[2] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *SIGPLAN PLDI'93*, pp. 112-125, June 1993.

[3] J. M. Anderson, S. P. Amarasinghe and M. S. Lam. Data and computation transformations for multiprocessors. In *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 19-21, 1995.

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

[5] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *ASPLOS VIII*, October, 1996.

[6] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *ACM SIGPLAN '86 Symp. Compiler Construction*, pages 162–175, June 1986.

[7] R. Chandra, D.-K. Chen, R. Box, D.E. Maydan, N. Nedeljkovic, and J.M. Anderson. Data Distribution Support on Distributed Shared Memory Multiprocessors. In *ACM SIGPLAN PLDI'97*, pp. 334–345, June 1997.

[8] B. Creusillet and F. Irigoin. Interprocedural array region analyses. *Int. Journal of Parallel Programming*, 24(6):513–546, December 1996.

[9] J. Gu, Z. Li, and G. Lee. Experience with efficient array data flow analysis for array privatization. In *Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, June 1997.

[10] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):350–360, 1991.

[11] K. Kennedy and K. S. McKinley. Loop fusion with arbitrary control flow. *Supercomputing '90*, pp. 407–416, 1990.

[12] R.E. Kessler and M.D. Hill, Page Placement Algorithms for Large Real-Indexed Caches, *ACM Trans. on Computer Systems*, 10(4), November 1992, 338–359.

[13] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *J. Par. and Dist. Computing*, 13:213–221, 1991.

[14] Z. Li. Array privatization for parallel execution of loops. In *ACM Int. Conf. on Supercomputing*, pages 313–322, July 1992.

[15] D.E. Maydan, S.P. Amarasinghe, and M.S. Lam. Array data-flow analysis and its use in array privatization. In *Proc. of the 20th ACM Symp. on Principles of Programming Languages*, pages 2–15, January 1993.

[16] T.N. Nguyen and Z. Li, Interprocedural Analysis for Loops Scheduling and Data Allocation, *Parallel Computing, Special Issue on Languages and Compilers for Parallel Computers*, 24(3), 1998, pp. 477–504.

[17] OpenMP. *OpenMP Fortran Application Program Interface, Version 1.0*, October, 1997. http://www.openmp.org.

[18] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *LNCS 768: Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*. Portland, OR, August 1993. Springer-Verlag.

[19] G. Rivera and C.-W. Tseng. Eliminating Conflict Misses for High Performance Architectures. In *Proc. of the 1998 ACM International Conference on Supercomputing*, Melbourne, Australia, July 1998.

[20] Silicon Graphics, Inc. *MIPSpro Power Fortran 77 Programmer's Guide*. Document Number 007-2362-002. http://techpubs.sgi.com.

[21] Silicon Graphics, Inc. *Origin-2000 Technical Specifications.* http://www.sgi.com/origin/2000_specs.html.

[22] Standard Performance Evaluation Corporation, *SPEC Newsletter*, Vols. 1–9, 1989–1997.

[23] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. *Proc. of Fifth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pp. 144–155, Santa Barbara, CA, July, 1995.

[24] P. Tu and D. Padua. Gated ssa-based demand-driven symbolic analysis for parallelizing compilers. In *International Conference on Supercomputing*, pages 414–423, July 1995.

[25] Michael Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley, 1994.