Applying Data Copy to Improve Memory Performance of General Array Computations

Qing Yi

Department of Computer Science, University of Texas at San Antonio *

Abstract. Data copy is an important compiler optimization which dynamically rearranges the layout of arrays by copying their elements into local buffers. Traditionally, array copy is considered expensive and has been applied only to the working sets of fully blocked computations. This paper presents an algorithm which automatically applies data copy to optimize the performance of general computations independent of blocking. The algorithm automatically decides where to insert copy operations and which regions of arrays to copy. In addition, when specialized, it is equivalent to a general scalar replacement algorithm on arbitrary array computations. The algorithm is fully implemented and has been applied to optimize several scientific kernels. The results show that the algorithm is highly effective and that data copy can significantly improve the performance of scientific computations, both when combined with blocking and when applied alone without blocking.

1 Introduction

Most scientific applications operate on large multi-dimensional arrays that cannot fit in the caches of modern computers. Such computations typically include sequences of loop nests, with each loop selectively accessing elements of arrays. When a loop accesses a non-continuous collection of array elements, that is, when the array elements accessed together close in time are far from each other in the memory, the loop demonstrates poor spatial locality and additionally could incur conflict misses in the cache.

Data copy is an important compiler optimization that can dynamically rearrange the layout of arrays. At the beginning of each computation phase, the transformation can choose to copy a subset of array elements into local buffers. All the relevant array accesses within the computation phase can then be changed to instead operate on the local buffers. At the end of the computation phase, if the selected elements are modified, the local buffers are copied back to the original arrays. Because the local buffers store working sets of computations continuously, data copy optimization can significantly improve the spatial locality of computations.

Data copy was first proposed by Lam, Rothberg and Wolf [9] to reduce cache conflict misses for blocked computations. As an example, Figure 1(a) shows a

* The work was developed when the author was under employment by Lawrence Livermore National Laboratory

```
int _j,_k,_i,j,k,i;
double alpha, *A, *B, *C;
......
for (_j=0; _j<n; _j+=16)
  for (_k=0; _k<l; _k+=16)
l_i:for (_i=0; _i<m; _i+=16)
l_j:  for (j=_j; j<min(n,_j+15); ++j)
l_k:    for (k=_k; k<min(l,_k+15); ++k)
l_i:      for (i=_i; i<min(m,_i+15); ++i)
          {
s:          C[i+j*m] = c[i+j*m] +
                alpha * A[i+k*m]*B[k+j*l];
          }
```

(a) without array copy

```
int _j,_k,_i,j,k,i, _bi,_bk,_v0,_v1;
double alpha, *A, *B, *C;
......
for (_j=0; _j<n; _j+=16)
  for (_k=0; _k<l; _k+=16)
l_i:for (_i=0; _i<m; _i+=16) {
      _bi = min(m-_i,16);   _bk = min(l-_k,16);
      _v0 = 0;
      for (_v1=_k; _v1<_k+_bk; ++_v1)
        for (_v2=_i; _v2<_i+_bi; ++_v2)
          A_buf[_v0++] = A[_v1*m+_v2];
l_j:  for (j=_j; j<min(n,_j + 15); ++j) {
        _v0 = 0;
        for (_v1=_i; _v1<_i+_bi; ++_v1)
          C_buf[_v0++] = C[j*m+_v1];
l_k:    for (k=_k; k<_k+_bk; ++k) {
          B_buf = B[j*l+k];
l_i:      for (i=_i; i<min(m,_i+15); ++i)
s:          C_buf[i-_i] = c_buf[i-_i] +
                  alpha*B_buf*A_buf[(k-_k)*_bi+i-_i];
        }
        _v0 = 0;
        for (_v1=j*m+_i; _v1<min(m,_i+16); ++_v1)
          C[_v1] = C_buf[_v0++];
      }
    }
```

(b) with array copy

Fig. 1. Example: blocked matrix multiplication

code fragment (written in C) that performs matrix multiplication, $C = C + alpha * A * B$, where *alpha* is a scaling factor, and A,B,C are $m * l$, $l * n$ and $m*n$ matrices respectively (each stored in a linearized single-dimensional array). The computation in Figure 1(a) is fully blocked in all loop dimensions, where A,B and C are each partitioned into $16*16$ sub-matrices, and each computation phase (enumerated by the inner loops l_j, l_k and l_i) multiplies a pair of sub-matrices. Because the working set of each computation phase is small enough to fit in the cache of most memory systems, the loop structure in (a) is likely to perform well on modern computers.

The computation in Figure 1(a), however, is not guaranteed to have good memory performance. Because the working set of each computation block is not stored continuously in the memory, each memory access may bring useless elements into cache, resulting in poor spatial locality. Further, when non-continuous array elements are brought into cache, their addresses may conflict with each other, resulting in premature evictions of useful elements. To resolve such problems, compilers could apply data copy transformation, which copies all elements accessed within the computation phase into continuous buffers.

This paper presents a new data copy algorithm for optimizing the performance of general array computations. Figure 1(b) shows the result of transformation after automatically applying our algorithm to the code in (a). Here all elements accessed by the inner loops are copied into separate buffers. Specifically, elements in array B are copied into a scalar variable B_buf, elements in C are copied into a single-dimensional buffer C_buf, and elements in A are copied into a two-dimensional buffer A_buf. The buffer sizes are 1 for B_buf, $\_bi$ for C_buf, and $\_bi * \_bk$ for A_buf respectively, where $\_bi$ and $\_bk$ are the iteration numbers of loops l_i and l_k respectively. The loop body has been accordingly changed to access elements from the local buffers. Since elements in C_buf are modified, these elements are copied back to the original matrix C at the end. As shown in Section 4, the code in (b) can significantly outperform the code in (a) in many cases.

The algorithm in this paper significantly improves previous research [9, 13], which treats array copy as an auxiliary optimization for blocking. Previous formulations would have optimized Figure 1(a) by performing all copy operations at the beginning and end of each computation block, i.e., the location that matrix A is copied in Figure 1(b). Our algorithm is much more flexible in that it treats data copy as a stand-alone optimization. Based on heuristics to reduce both buffer size and the overall copy cost, our algorithm automatically decides where to insert copy operations and which regions of the arrays to copy. The transformed code can use buffers at various levels, corresponding to the different levels of caches in modern computers. Our algorithm can also be specialized to perform scalar replacement optimization, which relocates array elements to scalar variables.

The algorithm in this paper is fully implemented and has been applied to optimize several kernels both combined with blocking and without blocking. Our results show that the algorithm is highly effective and that array copy can significantly improve the performance of scientific computations, both when combined with blocking and when applied separately without blocking.

2 Related Work

Lam, Rothberg and Wolf [9] first proposed applying array copy to reduce cache conflict misses in blocked computations. A few years later, Temam, Granston and Jalby [13] investigated different strategies for applying array copy after blocking and presented an effective strategy that selectively copy arrays based on compile-time cost-benefit analysis. Both Lam et al and Temum et al assumed that the blocked computations access arrays only through regular affine expression subscripts, where data copy can always be safely applied. They both consider data copy as an auxiliary optimization for blocking, where copy operations are inserted only at the beginning and end of blocked computations. Since then, very little work has been published to further investigate applying data copy to optimize array computations in scientific applications.

The data copy algorithm in this paper extends previous work in two aspects. First, our algorithm can optimize computations even if they contain regions of code that access arrays through non-affine expression subscripts. Second, our algorithm includes heuristics both to automatically select arrays to copy and to automatically identify different locations to insert copy operations. Thus our algorithm can be applied to optimize general computations independent of blocking.

Besides array copy, many data layout optimizations have been proposed to improve the memory system performance of regular array operations [11, 12, 3]. These optimizations statically reorganize the layout of arrays to reduce cache conflict misses and to improve spatial locality. They are effective when computations access arrays in a consistent fashion throughout an entire application. However, when computations include different phases, a single memory layout may not be sufficient. This paper does not attempt to globally restructure the layout of data structures. Instead, we dynamically rearrange array elements accessed in each computation phase when beneficial.

Despite being considered expensive, dynamic data layout transformations have been widely applied in optimizing irregular applications [6, 7, 10], where the structures of the input data are unknown until runtime. Because arrays in irregular applications are accessed through indirect pointers (or index arrays), current compiler technology cannot automate the optimization. In contrast, the data copy transformation in this paper is applied automatically to optimize regular array computations.

The algorithm presented in this paper is similar to the scalar-replacement algorithm by Carr and Kennedy [5] in many respects. Their algorithm aggressively promotes array elements into scalar variables so that these elements can later be allocated to registers. Our algorithm can similarly be configured to perform scalar replacement through restrictions on the size of array regions being copied. Our algorithm is more general than the algorithm by Carr and Kennedy in that we apply data copy to dynamically rearrange the layout of arrays in addition to performing scalar replacement.

3 Applying Data Copy

Figure 2 presents our algorithm for applying data copy to arbitrary array computations. This algorithm takes a code fragment C, partitions the array references in C into groups where each group can be safely copied into a single buffer, performs profitability analysis on each group of array references, and finally applies transformations when beneficial. Section 3.1 describes each step of the algorithm. Section 3.2 then describes the profitability analysis in more detail.

3.1 Data Copy Algorithm

As shown in Figure 2, given an input code fragment C, the algorithm includes the following steps.

Apply-data-copy(C)
(1) R = construct-dependence-graph(C)
    $nodes(R)$: memory references in C; $edges(R)$: dependences between references
    $\forall e \in edges(R)$, $dep(e)$: dependence relation; $precise(e)$: whether $dep(e)$ is precise
(2) Construct a DAG R' from R
    $order$ = evaluate-reference-order(C); $nodes(R') = nodes(R)$
    for (each edge $e : r_1 \rightarrow r_2$ in R)
      if ($order(r_1) < order(r_2)$) then add $e : r_1 \rightarrow r_2$ to R'
      else if ($order(r_1) > order(r_2)$) then add $reverse(e) : r_2 \rightarrow r_1$ to R'
(3) $groups$ = apply-typed-fusion(R')
    $BadEdges = \{e \in edges(R') \mid \text{not } precise(e)\}$
    $\forall r \in nodes(R)$, $type(r)$ = array-name(r)
(4) Profitability-analysis(R,$groups$)
  for (each $refs \in groups$), compute
    init-stmt($refs$) and save-stmt($refs$): start and ending points of computation
    cp-region($refs$): elements to be copied to local buffer
    shift-buf($refs$): offset to shift local buffer
    init-region($refs$): elements to be copied before starting
(5) For (each $refs \in groups$), perform array copy transformation
(5.1) buf = create-buffer(cp-region($refs$))
(5.2) if (!cover-modify(init-stmt($refs$), save-stmt($refs$),$refs$))
      $init$ = copy-init(buf,init-region($refs$)); insert-before(init-stmt($refs$), $init$)
    if (is-modified($refs$) or shift-buf($refs$) $\neq 0$)
      $save$ = copy-save(buf,cp-region($refs$),shift-buf($refs$));
      insert-after(save-stmt($refs$), $save$)
(5.3) for (each $r \in refs$)
      r_buf = buffer-access(r,buf,cp-region($refs$)); replace-ref(r, r_buf)

Fig. 2. Algorithm for applying array copy

Step (1) Construct a dependence graph R, where each node of R is a memory reference in the original code C, and each edge from reference r_1 to r_2 indicates that r_1 and r_2 may access the same memory location (i.e., r_2 and r_1 depend on each other), and r_1 is evaluated before r_2. Each edge e from r_1 to r_2 is annotated with two attributes: $dep(e)$, the dependence relation that must be satisfied between iterations of loops surrounding r_1 and r_2; and $precise(e)$, whether the dependence relation $dep(e)$ is precisely determined by the dependence analysis algorithm (i.e., whether both r_1 and r_2 contain only affine expression subscripts). Only when $precise(e)$ is true, r_1 and r_2 are guaranteed to refer to the same memory location when $dep(e)$ is satisfied, and the elements accessed by r_1 and r_2 can be copied into a single buffer.

The dependence graph R can be constructed using well-studied dependence analysis techniques [1, 14, 4]. The only difference here is that nodes in R are memory references rather than statements, and that a pair of references may depend on each other even if neither modifies the memory (that is, input dependences are considered together with the true, output and anti dependences).

Step (2) Prepare for Step (3) by converting the dependence graph R into a DAG (directed acyclic graph) R'. First, define a function *order* which assigns a unique integer number to each memory reference and thus imposes a linear order on all memory references. Specifically, $\forall r_1, r_2 \in nodes(R)$, if $order(r_1) < order(r_2)$, then r_1 appears before r_2 in C in static evaluation order; that is, r_1 is traversed before r_2 when we statically interpret the statements in C, assuming all loop bodies and conditional branches (both true and false branches) are entered exactly once.

Copy all the nodes and edges from R into R'. Ensure R' is acyclic by enforcing that every edge e from r_1 to r_2 in R' satisfies the condition $order(r_1) < order(r_2)$. Specifically, $\forall e : r_1 \rightarrow r_2$ in the original graph R, if $order(r_1) < order(r_2)$, copy e into R'. Otherwise, if $order(r_1) > order(r_2)$, reverse $dep(e)$ and then add the reversed dependence from r_2 to r_1 into R'. Finally, if $r_1 == r_2$, the edge is simply ignored because it does not affect the partitioning of memory references.

Step (3) Partition the memory references in R' into separate groups by applying the typed-fusion algorithm by Kennedy and McKinley [8], originally developed for performing loop fusion optimizations. The input to the original typed-fusion algorithm is a loop dependence graph, where each node of the graph is a loop, and each edge from node x to y indicates that there are dependences from statements inside loop x to statements inside loop y. An edge from x to y is annotated as a *bad edge* if the dependence relations between x and y prevent them from being legally fused. Additionally, each node in the loop dependence graph is assigned a type so that loops of different types are never fused. For each given type of loops (e.g., parallel or serial loops), the typed-fusion algorithm aggressively clusters nodes of the given type that are not connected by fusion-preventing *bad* paths. In order for the algorithm to work correctly, it is required that the input dependence graph must be acyclic (i.e., a DAG).

To adapt the typed-fusion algorithm for partitioning memory references, we use the DAG R' (computed in Step (2)) as input to the algorithm. Here *bad edges* are defined to include each edge $e \in R'$ such that $precise(e)$ is false, so that memory references connected by imprecise dependence paths are never placed into the same group. The names of arrays are used to represent types of memory references, and all non-array memory references are assigned a unique *dummy* type, which is never used as input to the fusion algorithm. Therefore no data copy transformation is applied to non-array memory references.

After applying the typed-fusion algorithm to the dependence DAG R', the result is a collection of clustered groups, where each group $refs$ includes a collection of array references that can be safely relocated to a single buffer. Based on the correctness proof of the original typed-fusion algorithm, it is guaranteed that no references in $refs$ are connected to each other by *imprecise* dependence paths.

Step (4) Use profitability analysis (described in Section 3.2) to further filter and configure the groups of array references to be copied. For each group of memory references $refs$, this step computes the following attributes.

- init-stmt($refs$) The starting point of a computation phase to apply data copy. When applying the transformation, the initialization operations should be inserted before this statement.
- save-stmt($refs$) The ending point of the computation phase. If the local buffer needs to be saved, the necessary operations should be inserted after this statement.
- cp-region($refs$) The region of array elements to be relocated to the local buffer.
- shift-buf($refs$) The offset to shift the local buffer between consecutive iterations of the current computation phase. Since accessing the local buffer is cheaper than operating on the original array, when appropriate, the local buffer can be shifted to reduce the overhead of copying from the original array. For more details, see Section 3.2.
- init-region($refs$) The region of array elements to be copied into the local buffer before init-stmt($refs$). Specifically, *init-region($refs$)* equals to *cp-region($refs$)* if the local buffer cannot be shifted (that is, *shift-buf($refs$)* $= 0$); otherwise, *init-region($refs$)* contains the elements accessed by $refs$ at the first iteration of the computation phase.

The above attributes are used by Step (5) to perform data copy transformations. As example, Figure 3 presents the configuration of these attributes when applying data copy to the matrix multiplication code in Figure 1(a). The evaluation of these attributes is described in more detail in Figure 4 and in Section 3.2.

Step (5) For each group of array references $refs$ to be copied, perform the transformation by allocating a local buffer, inserting operations to copy data between buffer and the original array, and replacing array references in $refs$ with the corresponding buffer accesses.

First, step (5.1) invokes function *create-buffer* to allocate a local buffer from the heap. The allocation is placed at the outermost location where the size of the buffer can be correctly evaluated. Deallocation of the buffer is also automatically inserted if necessary.

Then, step (5.2) inserts operations to copy data between the local buffer and the original array. Unless each iteration of the computation phase modifies all elements accessed by $refs$ before reading them (*cover-modify(init-stmt($refs$),save-stmt($refs$),$refs$)* is true), operations are inserted before *init-stmt($refs$)* to copy elements from the original array to the local buffer. Similarly, if the computation phase modifies elements accessed by $refs$, or if the local buffer needs to be shifted (*shift-buf($refs$)* $\neq 0$) between consecutive iterations of the computation phase, the necessary operations are inserted after *save-stmt($refs$)*.

Finally, step (5.3) replaces each array reference in $refs$ with the corresponding buffer access.

3.2 Profitability Analysis

This section describes Step (4) of the data copy algorithm in Figure 2. As shown in Figure 4, this step uses heuristics to determine whether a data copy trans-

references: $\{A[i + k * m]\}$ init-stmt: l_j save-stmt: l_j cp-region and init-region start: $\_i + \_k * m$ copy: (0,$min(m - \_i, 16)$,1), (0,$min(l - \_k, 16)$,m) shift-buf: 0	references: $\{B[k + j * l]\}$ init-stmt: l_i save-stmt: l_i cp-region and init-region start: $k + j * l$ copy: () shift-buf: 0	references: $\{C[i + j * m]\}$ init-stmt: l_k save-stmt: l_k cp-region and init-region start: $\_i + j * m$ copy: (0,$min(m - \_i$,16),1) shift-buf: 0

Fig. 3. Array copy configurations for Figure 1(a)

formation is beneficial and how to perform the transformation to ensure profitability. For each group of memory references $refs$ to be copied, it includes the following sub-steps.

Step (4.1) To reduce the overhead of performing data copy, make sure that each array element accessed by $refs$ needs to be copied at most twice: initially copied from the original array to the local buffer, and finally copied back from local buffer to original array.

First, invoke function *split-disconnected-refs(refs,R)* to separate array references in $refs$ that are not connected by dependence paths in R. Disconnected memory references are removed from $refs$ and added into the overall collection (*groups*) of array reference groups.

To ensure that each array element is copied at most twice, find *inroot* = common-loop($refs$), the innermost loop that surrounds all array references in $refs$. For each reference $r_2 \notin refs$, if r_2 is connected with references in $refs$ by dependence edges, and if $\exists r_1 \in refs$ such that $l_{r_1 r_2}$ is the innermost loop surrounding both r_1 and r_2, then the required copy operations must be inserted between r_1 and r_2 and inside loop $l_{r_1 r_2}$. If $l_{r_1 r_2}$ is nested at a deeper loop level within *inroot*, the copy operations inside $l_{r_1 r_2}$ will be evaluated multiple times at each iteration of *inroot* (the current computation phase). To avoid such situation, split $refs$ so that r_1 is placed into a separate group. After this step, all copy operations can be safely inserted immediately inside *inroot*.

Using Figure 1(a) as example, when applying steps (1)-(3) of the algorithm in Figure 2, Figure 3 presents the resulting three array reference groups. Since no splitting is necessary, this step merely set *inroot* to loop l_i for all reference groups.

Step (4.2) Decide the outermost loop, *cproot*, where copy operations can be safely inserted; that is, it is safe to relocate all elements accessed by $refs$ at each iteration of *cproot*. A single iteration of *cproot* therefore comprises the computation phase of the current copy transformation.

First, invoke function *copy-loop(inroot,refs,R)* to find the outermost loop, *outroot*, that contains all references in $refs$ but does not contain any reference r such that (i) r is outside *inroot*, and (ii) r and $refs$ may depend on each other within *outroot*. If *outroot* == *inroot*, copy operations must be inserted inside

Profitability-analysis(R,$groups$)
for (each $refs \in groups$)
(4.1) Ensure each element is copied at most twice:
split-disconnected-refs($refs$, R); $inroot$ = common-loop($refs$)
$cut = \{r_1 \in refs \mid \exists r_2 \notin refs$ s.t. $dep(r_2,refs) \neq \emptyset$ and common-loop(r_1,r_2) is inside $inroot$ $\}$
if ($cut \neq \emptyset$) split($refs$,cut); $groups \cup = \{cut\}$; $inroot$ = common-loop($refs$)
(4.2) Compute outermost loop level to perform copy:
$outroot$ = copy-loop($inroot$,$refs$,R)
if ($outroot == inroot$) $cproot = inroot$
else $cproot$ = loop-immediately-outside($outroot$)
(4.3) Impose size limit on the local buffer
split-disconnected-refs($refs$,R($cproot$));
cut=$\{r \in refs \mid$ is-too-big(array-region(r,$cproot$))$\}$
if ($cut == refs$) $cproot$ = loop-immediately-inside($cproot$); repeat step (4.3)
else split($refs$,cut); $groups \cup = \{cut\}$; go back to step (4.1)
(4.4) Ensure profitability of copy transformation
$reuse = \{l \mid l \in$ loops-between($cproot, inroot$) and carry-temporal-reuse($refs$,l)$\}$
if ($reuse \neq \emptyset$) $cproot$ = loop-immediately-outside(outermost-loop($reuse$))
else if ($|refs| \leq 3$) $groups- = \{refs\}$; continue
(4.5) configure copy transformation
cp-region($refs$) = array-region($refs$,$cproot$)
shift-buf($refs$) = array-region-shift($refs$,$cproot$)
if (shift-buf($refs$) $\neq 0$ and $cproot \neq inroot$ and $cproot \neq$loop-immediately-outside($outroot$))
init-stmt($refs$) = $cproot$; init-region($refs$)=init-array-region($refs$,$cproot$)
save-stmt($refs$)=last-stmt($refs$,loop-body($cproot$))
else
init-stmt($refs$)=first-stmt($refs$,loop-body($cproot$)); init-region($refs$)=cp-region($refs$)
shift-buf($refs$)=0; save-stmt($refs$)=last-stmt($refs$,loop-body($cproot$))

Fig. 4. Profitability analysis of array copy

$inroot$ ($cproot = inroot$). Otherwise, since no reference $r \notin refs$ can interfere with the memory accessed by $refs$ throughout the execution of $outroot$, it is safe to insert copy operations outside $outroot$. So $cproot$ should be the loop immediately enclosing $outroot$.

Using Figure 1(a) as example, since no dependence interference exists, we have $outroot = l_j$ for all three array reference groups in Figure 3. Consequently we would have $cproot = l_{\_i}$ for all reference groups.

Step (4.3) Impose a size limit on the local buffer. The size limit is dependent on various features of the computer architecture and is given to the data copy algorithm as a configuration parameter. In our prototype implementation, the size limit is imposed by restricting the dimensionality of local buffers using command-line options (see Section 4).

First, invoke function *split-disconnected-refs(refs,R(cproot))* to separate references that are disconnected from each other in the dependence graph of $cproot$. Next, find each reference r in $refs$ such that at each iteration of $cproot$, the el-

ements accessed by r exceed the buffer size limit. If the collection of references that access too many elements includes everything in $refs$ ($cut == refs$), lower *cproot* to be the loop immediately inside and repeat step (4.3). Otherwise, since only a subset of references in $refs$ are causing the problem, split $refs$ by removing such references, then restart from step (4.1).

Using Figure 1(a) as example, after Step (4.2), we have $cproot = l\_i$ for all reference groups in Figure 3. Since each reference group has a single array reference, and each array reference accesses at most $16 * 16$ elements at each iteration of loop l_i, the local buffer for each reference group has two dimensions. If only single-dimensional buffers are allowed, this step would reset $cproot = l_j$ for all reference groups. Similarly, if only scalar replacement is allowed, we would have $cproot(\{B[k + j * l]\}) = l_k$, and $cproot(\{A[i + k * m]\}) = cproot(\{C[i + j * m]\}) = l_i$.

Step (4.4) Evaluate the benefit of applying data copy and refrain from applying the transformation (by removing $refs$ from *groups*) if the benefit does not outweigh the cost.

First, find all the loops between *cproot* and *inroot* that carry temporal reuses of $refs$; that is, these loops do not increase the overall size of elements accessed by $refs$. Collect these loops into a set $reuse$ in Figure 4.

If $reuse$ is not empty, it is profitable to perform array copy because the local buffer will be reused many times. Find the outermost loop l in $reuse$ such that all the other loops between *cproot* and l merely increase the buffer size without introducing any memory reuse. Reduce buffer size by lowering *cproot* to be the loop immediately enclosing l.

If $reuse$ is empty, the copied elements are reused at most a few times ($\leq$ the number of elements in $refs$). If the number of elements in $refs$ is less than 3, the copy overhead is likely to outweigh the benefit of reuse. In this case, remove $refs$ from the groups of references to be optimized.

Using Figure 1(a) as example, suppose that $cproot = l\_i$ for all reference groups in Figure 3 before entering this step. After this step, we would have $reuse = \{l_j\}$, $\{l_k\}$ and $\{l_i\}$ for reference groups $\{A[i + k * m]\}$, $\{C[i + j * m]\}$ and $\{B[k + j * l]\}$ respectively. Consequently, $cproot(\{C[i + j * m]\})$ and $cproot(\{B[k + j * l]\}$ would be reset to l_j and l_k respectively, resulting in the data copy transformation shown in Figure 1(b).

Step (4.5) Suppose it is beneficial to apply data copy at each iteration of loop *cproot*. Compute necessary configurations to determine where to insert copy operations and what to copy.

First, invoke function *array-region(refs,cproot)* to summarize all the array elements accessed by $refs$ at each iteration of *cproot*. The result includes the starting address of the array to be copied and a sequence of tuples, (i_1, n_1, s_1), (i_2, n_2, s_2), ..., (i_m, n_m, s_m), where in each $(i_j, n_j, s_j)(j = 1, ..., m)$, i_j specifies the current array dimension to be copied, n_j specifies the number of elements to be copied at dimension i_j, and s_j specifies the incremental stride at dimension i_j. This formulation allows multiple copy specifications for each array dimension,

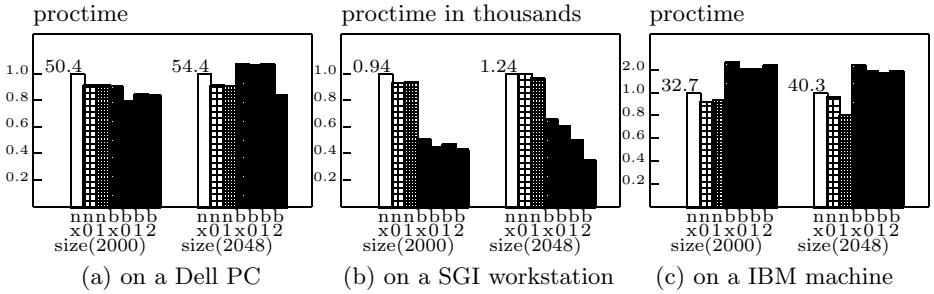

Fig. 5. performance of dgemm (nx:original non-blocked version; n0:optimized with 0-dimensional data copy; n1:optimized with 1-dimensional data copy; bx:optimized with loop blocking; b0:optimized with blocking and 0-dimensional data copy; b1:optimized with blocking and 1-dimensional data copy; b2:optimized with blocking and 2-dimensional data copy)

thus allowing linearized arrays (e.g., the arrays in Figure 1(a)) to be correctly copied. Given the the sequence $(i_1, n_1, s_1)(i_2, n_2, s_2)...(i_m, n_m, s_m)$, the size of the buffer is $n_1 * n_2 * ... * n_m$.

After computing *cp-region(refs)*, invoke function *array-region-shift(refs,cproot)* to compute the intersection of *cp-region* between consecutive iterations of *cproot*. If the overlapping region is not empty (shift-buf($refs$) $\neq 0$), it is more efficient to shift the local buffer rather than re-initiating the entire buffer from the original array. Shifting the local buffer is safe if *cproot* does not contain other references that interfere with $refs$ (*cproot* $\neq$ *inroot* and *cproot* is not the loop enclosing *outroot*).

If shifting the local buffer is necessary, the local buffer should be initialized before entering *cproot*. Thus *init-stmt*($refs$) = *cproot*. The initialization should copy elements accessed by $refs$ at the first iteration of *cproot*, so *init-region*($refs$) = *init-array-region*($refs$, *cproot*). The buffer needs to be shifted and re-initialized at the end of each iteration of *cproot*, so *save-stmt*($refs$) is the last statement in the loop body of *cproot*.

If shifting of local buffer is not necessary, we configure the transformation to always initialize the entire buffer in the loop body of *cproot* before the first statement that contains a reference in $refs$. Similarly, if necessary, the entire buffer should be restored back to the original array after the last statement that contains a reference in $refs$.

The configurations for applying array copy to Figure 1(a) is shown in Figure 3. Based on these configurations, applying Step (5) of Figure 2 to the code in Figure 1(a) would result in the optimized code in Figure 1(b).

4 Experimental Results

We have implemented our data copy algorithm within the loop transformation framework by Yi, Kennedy and Adve [15], which has been integrated as

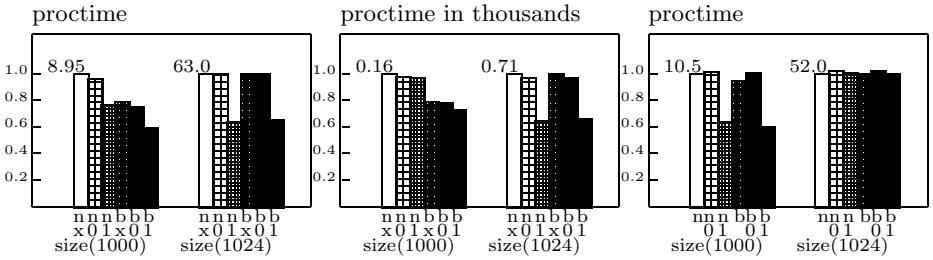

Fig. 6. performance of dgetrf (nx:original non-blocked version; n0:optimized with 0-dimensional data copy; n1:optimized with 1-dimensional data copy; bx:optimized with loop blocking; b0:optimized with blocking and 0-dimensional data copy; b1:optimized with blocking and 1-dimensional data copy)

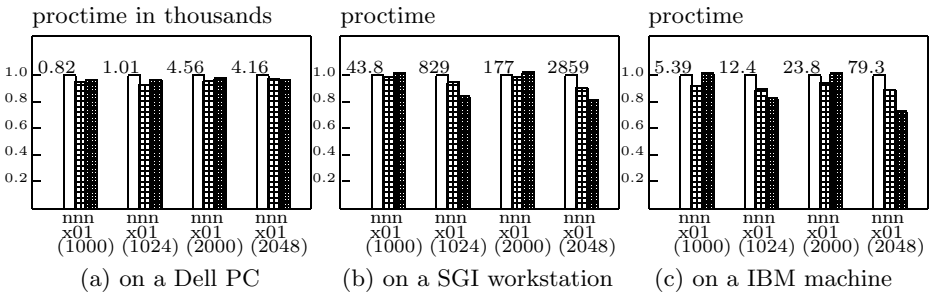

Fig. 7. performance of tomcatv using mesh sizes 1000, 1024, 2000 and 2048 (nx:original version; n0:optimized with 0-dimensional data copy; n1:optimized with 1-dimensional data copy)

a C/C++ source-to-source translator within ROSE, a C/C++ compiler infrastructure at LLNL [17]. This section presents the result of applying our algorithm to optimize three kernels, *dgemm* (matrix multiplication), *dgetrf* (matrix LU factorization with partial pivoting), and *tomcatv* (mesh generation with Thompson solver). All kernels are written in C. Both *dgemm* and *dgetrf* are transcribed from the corresponding non-blocked Fortran kernels in the LAPACK library [2], and *tomcatv* is transcribed from the Fortran kernel in SPEC95. When applying optimizations to these transcribed C codes, the dependence analysis in ROSE assumed that no arrays are aliased.

Data copy is applied to optimize all kernels. In addition, blocking is applied to *dgemm* and *dgetrf* to investigate the combination of blocking and data copy (the result of applying blocking to *tomcatv* is not shown because it was not beneficial). For each blocked version, different block sizes were experimented and the version with the best performance is presented. When performing data copy transformation, the optimizer is configured by command-line options to restrict the dimensionality of required buffers — if the buffer dimension is restricted to be m (denoted as *m-dimensional copy*), the optimizer would only perform data

copy to arrays that require at most m dimensional buffers. When the buffer dimension is restricted to be 0, only scalar replacement is performed.

For each kernel, different problem sizes were experimented. The performance of each version was measured on three different machine architectures: a Dell PC with two 2.2GHz Intel XEON processors (each with a 512KB cache) and 2GB memory; a SGI workstation with a 195 MHz R10000 processor, 32KB 2-way associative first-level cache, 1MB 4-way associative second-level cache, and 256MB memory; and a single 8-way P655+ node (with 16GB memory) on a IBM terascale machine. The kernels were compiled using *gcc* on the Dell PC and vendor-provided compilers on the SGI work station and IBM machine. All versions were compiled using -O3 option, which instructs the compilers to perform aggressive backend optimizations. The processor time (*proctime*) spent executing each version is presented.

Figure 5 presents the performance of *dgemm* using two matrix sizes, 2000^2 and 2048^2. Seven versions are measured for each matrix size, including the original non-blocked version (version nx), versions optimized with data copy optimizations only (versions $n0$ and $n1$), the version optimized with only blocking(version bx, shown in Figure 1(a)) and the versions optimized with both blocking and data copy (versions $b0$,$b1$ and $b2$, version $b2$ is shown in Figure 1(b)[1]).

From Figure 5, we see that 0-dimensional array copy (i.e., scalar replacement) is beneficial for *dgemm* in all cases, and the improvements range from 3%-12%. When using matrix size 2000^2, additional copy transformations do not further improve performance. However, when using matrix size 2048^2, the additional data copy, especially the two dimensional copy of array A, significantly improves the performance (over 40% for the blocked versions on the Dell PC and on the SGI workstation). Here the 2048^2 matrices have incurred much more cache conflict misses, which are subsequently eliminated when the accessed elements are copied into local buffers. The optimizations did not improve the performance as much on the IBM machine due to the heavy integer operation overhead introduced by the optimizations, which will be further investigated.

Figure 6 presents the performance of $dgetrf$ (matrix LU factorization with partial pivoting) using two matrix sizes, 1000^2 and 1024^2. Six versions are measured for each matrix size, including the original non-blocked version (version nx), versions optimized with data copy only (versions $n0$ and $n1$), the version optimized with blocking only (version bx), and versions optimized with both blocking and copy optimizations (versions $b0$ and $b1$). Because $dgetrf$ can be blocked only in the column direction (for details, see Yi et al [16]), at most a single dimension of the matrix needs to be copied. Thus there is no $b2$ version for $dgetrf$.

From Figure 6, we see that 0-dimensional array copy (scalar replacement) is not profitable for $dgetrf$ on the Dell PC and incurs a slight overhead on the IBM machine due to increased register pressure. The 1-dimensional copy transformation, however, significantly improves performance in most cases by

[1] The $b1$ and $b0$ versions are different from version $b2$ in that array A is not copied in $b1$, and only array B is copied in $b0$

20%-40% except when using 1000^2 matrix on the SGI workstation and when using 1024^2 matrix on the IBM machine. Here because the original arrays were accessed with a large stride, applying data copy have provided much better spatial locality. Again, the versions using 1024^2 matrix have performed much worse than using 1000^2 matrix due to conflict misses in memory systems.

Figure 7 presents the performance of *tomcatv* (mesh generation with Thompson solver) using four mesh sizes, 1000^2, 1024^2, 2000^2 and 2048^2. Because blocking is generally not profitable for *tomcatv*, array copy is the only optimization applied. Three versions are measured for each mesh size, denoted using nx (the original version), $n0$ (optimized with 0-dimensional data copy), and $n1$ (optimized with 1-dimensional data copy). In *tomcatv*, as each element is accessed within the inner loop, the four neighboring elements are also accessed. The local buffer therefore serves as a small shifting window through the entire mesh.

From Figure 7, we see that 0-dimensional array copy (scalar replacement) is profitable for *tomcatv* in almost all cases (ranging from 0.5% to 12%). The 1-dimensional copy transformation significantly improves performance by 11%-19% when using 1024^2 and 2048^2 meshes on the SGI workstation and on the IBM machine, but slightly slows down performance by 0.5%-8% in other cases. Here again, when using 1024^2 and 2048^2 meshes, the extra benefit of applying array copy comes from the reduction of conflict misses in the memory system.

In summary, the experimental results indicate that selectively applying data copy to optimize array computations can significantly improve the performance of scientific applications, especially when array elements are accessed in large strides and when conflict misses become a factor in the memory performance. The performance measurements also indicate that data copy does not need to be applied together with blocking to be effective. In fact, data copy optimization was able to significantly improve performance for all three kernels without blocking. Finally, even when data copy is not beneficial, the overhead is not overly significant, and only small slow downs (.5%-8%) in performance are observed for all kernels.

5 Conclusion

This paper presents a general algorithm for applying data copy to optimize arbitrary array computations. The algorithm is fully implemented and has been applied to automatically optimize several scientific computation kernels on different platforms. The results indicate that the algorithm is highly effective and that array copy can significantly improve the performance of scientific computations, both when combined with blocking and when applied alone without blocking.

References

1. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, October 2001.

2. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. The Society for Industrial and Applied Mathematics, 1999.
3. J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformation for multiprocessors. In *ACM Symposium on Principles and Practices of Parallel Programming*, Santa Barbara, July 1995.
4. U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, 1988.
5. S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software — Practice and Experience*, 24(1):51–77, Jan. 1994.
6. C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Gorgia, May 1999.
7. H. Han and C.-W. Tseng. Improving locality for adaptive irregular scientific codes. Technical Report CS-TR-4039, Dept. of Computer Science, University of Maryland, September 1999.
8. K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, Aug. 1993. (also available as CRPC-TR94370).
9. M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, Apr. 1991.
10. J. Mellor-Crummy, D. Whalley, and K. Kennedy. Improving Memory Hierarchy Performance For Irregular Applications. In *Proceedings of the 13th ACM-SIGARCH International Conference on Supercomputing*, Phodes, Greece, 1999.
11. M. O'Boyle and P. Knijnenburg. Integrating loop and data transformations for global optimisation. In *International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, Oct 1998.
12. G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.
13. O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing '93*, Portland, OR, Nov. 1993.
14. M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, 1989.
15. Q. Yi, K. Kennedy, and V. Adve. Transforming complex loop nests for locality. *The Journal Of Supercomputing*, 27:219–264, 2004.
16. Q. Yi, K. kennedy, H. You, K. Seymour, and J. Dongarra. Automatic blocking of qr and lu factorizations for locality. In *The Second ACM SIGPLAN Workshop on Memory System Performance*, Washington, DC, USA, June 2004.
17. Q. Yi and D. Quinlan. Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In *The 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, Sep 2004.