

Localizing Non-affine Array References

Nicholas Mitchell, Larry Carter, Jeanne Ferrante*

University of California, San Diego

E-mail: {mitchell, carter, ferrante}@cs.ucsd.edu

Abstract

Existing techniques can enhance the locality of arrays indexed by affine functions of induction variables. This paper presents a technique to localize non-affine array references, such as the indirect memory references common in sparse-matrix computations. Our optimization combines elements of tiling, data-centric tiling, data remapping and inspector-executor parallelization.

We describe our technique, bucket tiling, which includes the tasks of permutation generation, data remapping, and loop regeneration. We show that profitability cannot generally be determined at compile-time, but requires an extension to run-time. We demonstrate our technique on three codes: integer sort, conjugate gradient, and a kernel used in simulating a beating heart. We observe speedups of 1.91 on integer sort, 1.57 on conjugate gradient, and 2.69 on the heart kernel.

1. Introduction

Researchers have long sought to increase data locality and exploit parallelism in loop nests [34, 32, 16, 5, 33, 18]. These works succeed in optimizing a large class of loop-based stencil computations, such as matrix multiplication. However, each assumes *affine* array index expressions and loop bounds. The affine assumption allows matrix and polyhedral algebra and leads to uniform dependence vectors and memory strides. Furthermore, many codes satisfy this assumption. Nonetheless, as shown in Table 1, many important problems, such as sparse matrix algorithms, unstructured mesh calculations, and some sorting methods, contain non-affine array references and loop bounds.

Furthermore, non-affine references impede performance. On an Alpha 21164a, a loop which computes 100,000 iterations of $A[B[i]]++$ takes 6.3 cycles per iteration when $B[i] = i$, compared to 40 cycles per iteration when $B[i] =$

code	NAR % of total		iteration count
	static	dynamic	
heart loop	13.3%	12.4%	96568512
CG	9.6%	31%	311031
EP	7.1%	16%	26354769
FT	4.5%	1.3%	175104
IS	18%	25%	547085
MG	1.7%	trace	4.6
apsi	5.0%	0.5%	12.1
fpppp	5.2%	4.5%	74734200
mgrid	3.7%	trace	4.6
su2cor	8.9%	85%	18156
go	42%	61%	14.5
Distuf	29%	35%	16223
Zeus2D	12%	5.8%	2228
Zeus3D	1.5%	1.6%	149815
SuperLU	24%	45.1%	1510
bls1	7%	11.5%	15514
las1	3.9%	21.9%	12471
sis1	1.4%	7.2%	11333

Table 1. Using SUIF2, we developed a tool to quantify the occurrences of non-affine references (NARs) in three ways: the fraction of static and dynamic references which are non-affine and the average iteration count of the containing loop nests. Codes are grouped by benchmark.

$i^2 \bmod N$ or $B[i] = random()$. A conjugate gradient code spends 93% of its execution time in a loop nest with an indirect memory reference. Despite the efforts of the vendor compiler to mask latencies, the load nevertheless stalls the processor an average of 24 cycles per iteration. Indirect loads in an integer sort stall the processor 29 cycles per iteration. Yet, existing techniques cannot localize these codes.

What keeps previous solutions from localizing non-affine references? We claim non-affine index expressions add four complications. To localize $A[f]$, where f is a non-affine function of induction variables, we:

*This research supported by NSF grant CCR-9808946 and DARPA grant DABT63-97-C-0028.

strategy	IS permutation	remapping generality	remap base array?	cost analysis
tiling	block-style	n/a	n/a	compile-time
shackling	block-style	n/a	n/a	compile-time
remapping	block-style	linear	n/a	compile-time
inspector-executor for parallelism	only for ism	duplicate removal	yes	limited run-time
inspector-executor for locality	general	non-linear	yes	future work
bucket tiling	general	non-linear	no	future work

Table 2. Summary of recent locality-improving work, compared to bucket tiling.

- **can't tile:** Tiling can only express a restricted class of iteration permutations. Localizing non-affine references requires an arbitrary permutation of the iterations, since the original access pattern may be arbitrary.
- **can't just reorder:** If $f = B[i]$, reordering the i loop increases locality to A , but reduces locality to B . We must also remap the storage for B .
- **can't remap A :** Remapping the base array A requires accessing A through f , which of course results in the nonlocal memory references we are trying to avoid.¹
- **can't assume benefit:** f being non-affine is a necessary, but not sufficient, predictor of poor performance. Yet, a compile-time analysis cannot establish the sufficient predictions. Thus, we must extend profitability analysis to run-time.

Table 2 compares related works on these four issues. *Tiling* partitions a computation into polyhedral regions, called tiles [34, 13, 1]. Data-centric tiling [18] partitions the data into polyhedral regions, and then *shackles* the associated computation to each region. The shackle initially takes the form of conditional guards. If the shackle defines polyhedral computations (as is the case with affine subscripts), then the guard can be pushed into the loop bounds. Both tiling and shackling increase locality by reordering or *permuting* computations. However, these existing techniques rely on loops to express permutations. The advantage of this restriction is that the analysis and transformations can be completed at compile time. The disadvantage is that loops cannot express a general enough permutation to localize non-affine references. Furthermore, these works have not incorporated data remapping or run-time cost analysis.

Other researchers have used linear *storage remapping* to increase the spatial locality of affine memory references [2, 20, 18, 15]. For example, they might remap from row- to column-major order, or to storage aligned with diagonals. Leung's work [20] also addresses issues particular to non-affine index expressions. Other researchers combine iteration and data reordering [6, 7, 14]. All of these works use linear remappings. Just as iteration permutations

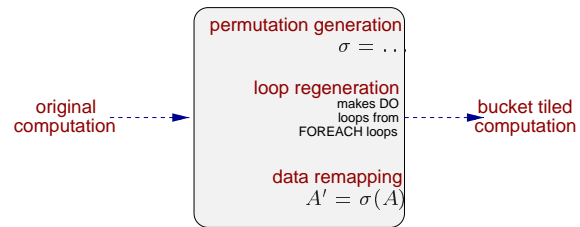


Figure 1. The process of bucket tiling.

via tiling cannot optimize non-affine references, linear data remappings also do not suffice.

Loop nest modifications, such as strip mining and interchange, cannot express these general permutations. Instead, we compute the permutation at run-time, using the *inspector-executor* framework [30]. Saltz *et al.* originally developed this framework to parallelize unstructured codes for distributed memory machines. As such, their solution copies the *base array* (e.g., A in the case of $A[B[i]]$). To parallelize a computation involving $A[B[i]]$, when A is initially distributed amongst the processors, there must be a “processor-local” version of A . Recently, Ding and Kennedy [10] have applied the original inspector-executor technique to increase locality. A direct application of inspector-executor cannot localize non-affine references, as it remaps A . Remapping A would perform as badly as the original computation.

Thus, localizing non-affine references requires a general iteration permutation, as well as a non-linear data remapping. To generate a permutation and remap the data, we must introduce a computation to do so at run-time. However, this new computation must also perform well.

In the remainder of this paper, we describe a technique, bucket tiling, which satisfies these criteria. Figure 1 visualizes the process of bucket tiling. As we will see in Section 2, bucketizing, the first stage of a bucket sort, provides the basis of bucket tiling. A bucket, or counting, sort first splits the set into a number of buckets (bucketizing), then sorts each bucket. Then, in Section 3, we describe the three main tasks of bucket tiling: permutation generation, loop regeneration, and data remapping. Next, Sections 4, 5, and 6 describe the *mechanism* and *cost issues* of these three tasks. Then, in Section 7, we explore bucket tiling's effectiveness

¹In Section 2, we will see why remapping A is undesirable.

on three examples. Finally, in Section 8.1, we discuss future work, including performance prediction.

2. Classifying Reference Patterns

To quantify the performance effects of bucket tiling mechanisms, we introduce a classification of reference patterns. The reference pattern of a computation accessing an array can be one of four types: sequential, k -sequential, $k \times m$ -sparse-sequential, and m -random. A *sequential* pattern references cache lines consecutively and thus has the best performance. A *k -sequential* pattern consecutively references k cache lines in tandem. Intuitively, sequential access traces one finger through memory; k -sequential traces a number of fingers through memory, with the fingers touched in no particular or predictable order. A *$k \times m$ -sparse-sequential* pattern cycles k times through m cache lines; each pass references memory sparsely, but in monotonically increasing order.² Finally, an *m -random* pattern references m cache lines in an unpredictable order. Figure 2 visualizes the four reference patterns.

We now define a metric of performance, *spatial footprint*, which bounds the number of lines a cache must have for the computation to obtain spatial reuse.³

Definition 1 The spatial footprint, \mathcal{F} , of a computation is the sum of contributions from the reference pattern of each array reference in the computation: 1 for sequential, k for k -sequential, m for $k \times m$ -sparse-sequential, and m for m -random.

Thus, a sequential pattern always yields good performance; a k -sequential pattern performs well only when k is small. Both a $k \times m$ -sparse-sequential and an m -random pattern perform poorly only if m is too large.

2.1. Bucketizing reduces spatial footprint

The reference pattern of an array A indexed by a non-affine function is at worst m -random, for some large m . Bucketizing fixes this bad behavior (Obs. 2), without itself being bad (Obs. 1).

Observation 1 Bucketizing a function f into n buckets is n -sequential.

Observation 2 Assume a bucket has at most s cache lines. Processing a bucket, after bucketizing, is at worst s -random.

²For example, an index expression of $i^2 \bmod m$ in a loop running $1 \leq i \leq N$ is $N/\sqrt{m} \times m$ -sparse-sequential.

³A computation has *spatial reuse* if it loads a cache line and eventually uses multiple elements from that line before it is replaced.

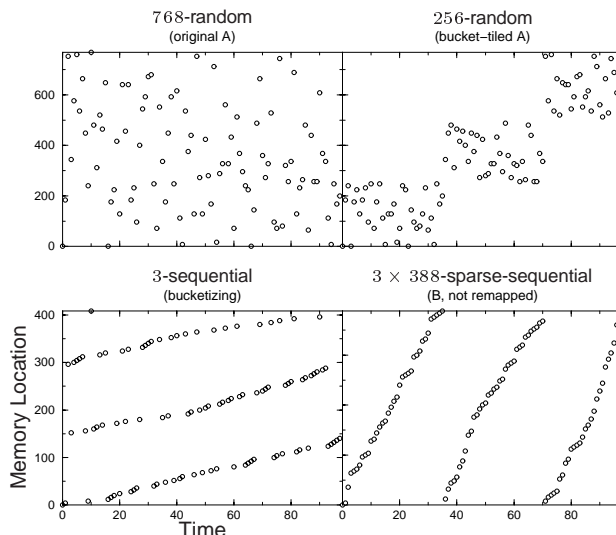


Figure 2. The four reference patterns for $A[B[i]]$, if B is initialized to random values.

These observations follow easily from the properties of bucketizing. Notice two important consequences. First, by bucketizing, we can control the spatial footprint of non-affine references. Second, the benefit of bucketizing depends on both the footprint of the initial reference and our choice of f .

2.2. Net effect of bucket tiling

Directly applying the inspector-executor technique to localizing non-affine references, transforms a m -random pattern to a 1 -sequential pattern; we have effectively linearized storage. However, to accomplish this improvement requires introducing a computation with the same, unfortunate pattern as the original computation.

In contrast, bucket tiling transforms m -random to k -random, for some constant k . To do so, it introduces a *bucketizing* computation which has a n -sequential (for n buckets) reference pattern. We now describe the tasks necessary to support this transformation.

3. The three tasks of bucket tiling

Our solution for localizing non-affine references involves three tasks: permutation generation, data remapping, and loop regeneration. We introduce these concepts with the following loop nest on the left and shackled [18] version on the right:

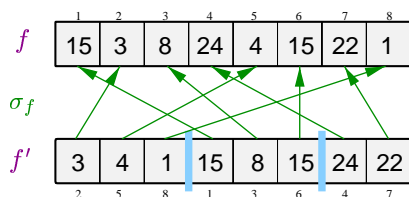
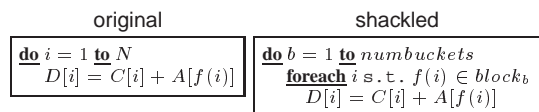


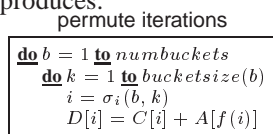
Figure 3. We use bucketizing, the first stage of a bucket sort, to generate a permutation, σ_f , which induces a new index function f' .



If the shackle, $\{i : f(i) \in \text{block}_b\}$, defines a polyhedral region of computation, the *foreach* becomes a *do* loop which scans the polyhedron [1]; this is the case when $f(i)$ is an affine function of i . In this paper, we develop support for the case when the *foreach* loop cannot be implemented with *do* loops. Our strategy is to generate a replacement iteration order. If this new iteration order is legal, localizes the computation, and allows the shackle to be implemented with *do* loops, the transformation has proven successful.

Thus, our first task: **permutation generation**. To generate the new iteration order, we *bucketize* $\{f(i)\}$ at run-time. Bucketizing induces a permutation, which we call σ . For example, bucketizing $\{f(i)\}$ into three buckets might behave as in Figure 3.

Our second task, **loop regeneration**, implements the *foreach* loop as *do* loops. Ancourt and Irigoien [1] developed a strategy for scanning polyhedra with *do* loops. We must provide mechanisms for doing likewise with buckets; we refer to our process for regenerating *do* loops as *bucket scanning*. A straightforward solution implements the *foreach* as a single *do* loop, iterating up to the size of a bucket. As with loop coalescing [35], we maintain legality by extracting the old induction variables; this extraction is precisely defined by a set of permutations, one for each of the original induction variables. In our example, the coalescing strategy produces:



However, loop regeneration alone may exacerbate the problem. Our final task, **data remapping**, ensures that bucket tiling actually increases performance. For example, if we bucketize into k buckets and the original loop accesses A m -randomly, then the permuted loop accesses both C and D in a $k \times m$ -sparse-sequential pattern. Thus, the coalesced implementation may not perform well. On an Alpha 21164a

with $N = 2$ million and f yielding random values from 1 to 2 million, coalescing increases execution time from 160 to 440 cycles per iteration. To remedy this problem we remap the storage of C and D . The permuted, *remapped* computation takes 61 cycles per iteration.

Of course, we have ignored the run-time cost of computing σ , which will affect overall performance. Furthermore, we were only able to increase performance because the original code performed poorly. For example, with integer sort, if the input is initially sorted, performance is degraded by a factor of three using our technique; whereas, if the input is random, there is a 50-75% improvement. In Section 8.1, we briefly issue of *performance prediction*. In most cases, a static analysis cannot predict performance. Consequently, future work will develop low-overhead dynamic techniques to estimate, for example by sampling, the projected benefit.

4. Permutation Generation

Bucket tiling requires a general class of permutations. We use the permutations generated by bucketizing. In this section, we develop the mechanism and cost issues of bucketizing. We do so in three stages: *spotting* the problematic array reference, *choosing* the function from that array's index expression to bucketize, and *generating* the permutation from that function.

Each mechanism described in the remainder of this paper has implications on three aspects of performance: additional time overhead in both the main and setup computations, and increased space requirements. Thus, we describe how each mechanism impacts these performance aspects.

As it reorders iterations, a permutation is legal only if it respects the loop's data dependences. In this paper, we use the conservative restriction that there are no loop-carried dependences.

4.1. Spotting A

First, we must spot the problematic array references.

Mechanism: If a computation contains more than one distinct non-affine reference, bucket tiling cannot apply. However, we can reduce the number of non-affine references in a computation in two ways, both of which divide the computation into two computations, each with fewer non-affine references than the original. Then, we recursively apply bucket tiling to each computation. The two mechanisms are loop splitting and copying. The former only applies when fusion is legal, and the latter only reduces the number of right-hand side non-affine references.

Cost consequences: Copying increases memory usage and adds time overhead. It is beyond the scope of this paper to analyze the cost consequences of loop fusion; see [8] and [16] for in-depth treatments.

4.2. Choosing f

Having spotted the single problematic array reference, we next choose a function, f , to bucketize. Choosing f is only a matter of identification (i.e. pointing it out); hence there are no mechanism issues.

Cost consequences: We examine two cost consequences of a choice of f : the affected loops and bucket tiling's effect on spatial footprint.

Definition 2 Given f , a loop with induction variable i is in the set of affected loops, \mathcal{A}_f , if f is a function of i .

Therefore, a choice of f determines the set of affected loops. As an example consider a problematic array reference $A[I[p] + i, J[p] + j, K[p] + k, d]$.⁴ If we choose f to be the entire index expression, then the set of affected loops is the set of all loops: $\{p, d, i, j, k\}$. If we choose f to be only the non-affine component⁵ then the set of affected loops is $\{p\}$.

In order to compare the set of affected loops with the loops on which an array reference depends, we present two useful notations:

Definition 3 For an array reference a , we denote i_a as the index expression of a , and \mathcal{I}_a as the set of induction variables on which i_a depends.

This notation allows us to express, for example, $\mathcal{I}_a = \mathcal{A}_f$, which means that a 's index expression depends on all of the affected loops of a given choice of f . We have defined \mathcal{A}_f and \mathcal{I}_a as sets of loops or induction variables; we assume there is a one-to-one mapping between loops and induction variables. We denote the iteration space associated with a particular set of loops, such as \mathcal{A}_f , by $space(\mathcal{A}_f)$.

The second cost consequence of a choice of f is on spatial footprint. Suppose $i_A = f + g$. Initially, the spatial footprint of the computation is $\mathcal{F}(i_A) = \mathcal{F}(f) + \mathcal{F}(g)$. After bucketizing f , the new spatial footprint is at most $\mathcal{F}(f') + k$.⁶ Therefore, the change in spatial footprint is $\mathcal{F}(i_A) - (\mathcal{F}(f') + k) = \mathcal{F}(f) - \mathcal{F}(f')$. Recall though that bucketizing bounds $\mathcal{F}(f')$ by s .

Therefore, if we choose f so that $\mathcal{F}(f) \leq s$, then we haven't improved performance. However, we need not always choose f to be the entire index function. We need only maximize $\mathcal{F}(f) - \mathcal{F}(f')$.

4.3. Generating σ

Having spotted A and chosen f , we now bucketize f .

⁴This reference is from the heart loop in Section 7.1.

⁵If A_k is the size of the k th dimension of A , then the non-affine component of the heart-loop's problematic reference is $I[p] + (J[p] + K[p] * A_2) * A_1$.

⁶ f' is the index function induced by bucketizing; see Figure 3.

Mechanism: Bucketizing f generates a permutation which reorders the iterations of the loop nest. We represent the permutation as a collection of permutations σ_i , one for each original induction variable i . We call *permute* the process of bucketizing f , and hence generating the σ_i 's. The following is a template for *permute*(f, n), which bucketizes f into n buckets.

```

permute(f, n)
  foreach  $\vec{i} \in space(\mathcal{A}_f)$ 
    bucketize(bucket(f,  $\vec{i}$ , space( $\mathcal{A}_f$ ), n))++
  allocate necessary memory
  foreach  $\vec{i} \in space(\mathcal{A}_f)$ 
     $b = bucket(f, \vec{i}, n)$ 
     $k = tallies(b)++$ 
     $\forall j \in \mathcal{A}_f \sigma_j(b, k) = j$ 

```

We define $bucket(f, \vec{i}, I, n)$, the bucket into which $f(\vec{i})$ falls, to be $f(\vec{i})/maxsize$. Here, $maxsize = range(f, I)/n$ is the maximum number of problem elements per bucket; $range(f, I)$ is the maximum minus the minimum value of f over the space I .

We cannot provide *permute* as a general-purpose library routine, as we need to loop over the iteration space of the computation being optimized. Instead, we provide a *permute*-generator, *pgen*, which statically generates *permute*, and which in turn dynamically generates the σ_i . *pgen*(f, I), given the indexing function f and an iteration space over which to bucketize f , produces *permute* $_f(n)$. We have developed a *pgen* which produces C code.

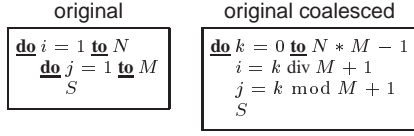
Cost consequences: The *permute* function we generate determines part of the overhead of bucket tiling. Recall from Observation 1 that bucketizing into n buckets is cheap: the spatial footprint of generating one σ_i is n -sequential. Therefore, the overall spatial footprint of generating all the σ 's is $(n|space(\mathcal{A}_f)|)$ -sequential. Notice the interplay between a choice of f and the overhead of bucketizing.

5. Loop Regeneration

Next, we describe loop regeneration. Recall that this task implements a *foreach* loop (for each iteration within the current bucket) with *do* loops. We call the strategies for loop regeneration developed in this section *bucket scanning*. In this section, we provide mechanisms for optimizing performance in this space of possibilities. We start from an initial implementation which always works. Then, we provide two refinements to improve performance.

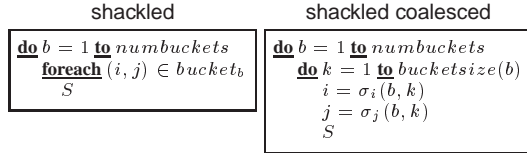
5.1. Initial Implementation: Coalescing

Mechanism: Our initial solution scans each bucket with a single loop, in the manner of loop coalescing [27]. Loop coalescing extracts the original induction variables from the single, new one. For example, coalescing applied to the following loop nest produces the single loop on the right:



In this case, a static transformation can restore the original induction variables. With bucket tiling, the mapping from coalesced to original is not statically determinable.

However, we can extract the original induction variables using the σ 's generated in Section 4. To scan the b th bucket, replace the *foreach* with a *do* loop which iterates over a new induction variable k , such that $1 \leq k \leq \text{bucketsize}(b)$. Inside the new *do* loop, extract each induction variable via $i = \sigma_i(b, k)$. For example:



Cost consequences: Since coalescing is always legal [35], this strategy provides a reasonable starting point. How well does it perform, using the three metrics of time overhead in the setup and main computations, and space requirements? The only setup overhead this strategy adds is to generate the σ 's. Coalescing more adversely affects the main computation: it introduces the outer b loop, adds extra loads to to recover the original induction variables, and increases the spatial footprint by replacing sequential array accesses with sparse-sequential array accesses. For a given choice of f , we must extract $|\text{space}(\mathcal{A}_f)|$ induction variables; notice the interplay between a choice of f and the overhead of bucket scanning. Finally, this strategy increases memory requirements by requiring all σ 's from the permutation generation phase to persist through the main computation.

5.2. Performance Improvements

To improve the performance of this initial implementation, we suggest two operations: permutation hoisting, and loop collapsing.

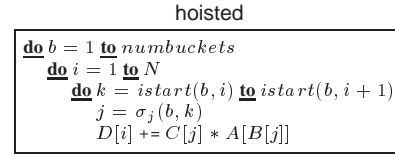
5.2.1 Hoisting

Hoisting provides another mechanism for removing references to original induction variables.

Mechanism: This operation eliminates a reference to induction variable i , if two conditions hold: i must be the induction variable of the outermost loop in \mathcal{A}_f , and each array reference a in the loop nest must have the property that if $i \in \mathcal{I}_a$ then $\mathcal{I}_a = \{i\}$. Thus, if in the above loop nest S is $D[i] = C[j, i] * A[B[j]]$, then hoisting does not apply, because $\mathcal{I}_C = \{i, j\}$. On the other hand, if S is

$D[i] += C[j] * A[B[j]]$, hoisting does apply and eliminates the reference to σ_i .

The initial coalesced version scans each bucket with a single loop in an order determined by the original iteration order. If the above two conditions hold, hoisting restores the outer loop and creates a new coalesced inner loop; the new loop scans iteration points within the (b, i) th bucket. For example, hoisting applied to the above loops produces:



Hoisting, when used in tandem with remapping, replaces an array reference a with $a'[b, i, k]$.

How do we generate the *istart* array? We can integrate this process with permutation generation by inserting a computation of *istart* as follows: before each iteration of the inner loop, and after the last iteration, for all buckets b , $\text{istart}(b, i)$ is set equal to the current bucket tally, $\text{tallies}(b)$.

Cost consequences: Hoisting adds the minimal setup overhead of computing *istart*. It adversely affects the main computation by adding multiple loads of references that used the hoisted σ s, and the additional addressing arithmetic, if used in tandem with remapping. However, hoisting benefits the main computation by removing a reference to one σ . Relative to just remapping, hoisting benefits storage by allowing the removal of a σ without an storage-expanding remapping. It also adds the storage for the *istart* array. The size of this array is nN , if we use n buckets and the iteration count of the i loop is N .

5.2.2 Collapsing

In certain situations, we can collapse the k and b loops into a single loop. *Loop collapsing* [35] is a special case of loop coalescing, defined as follows.

Mechanism: Collapsing works when $|\mathcal{I}_A| = 1$. If this condition holds, we can scan the buckets with a single loop that iterates over $1 \leq k \leq |\mathcal{I}_A|$. This doesn't eliminate the one load of σ_i . However, if a further condition holds, then we can scan the buckets with the original loop, and hence eliminate the load of σ_i . This further condition depends on a concept we call *vicinity*, defined as follows:

Definition 4 *The vicinity of an array a , \mathcal{V}_a , is the set of all references in the same loop nest as a which satisfy the following "chaining" relation. a_1 is chained to a_2 when*

1. $a_1 = a_2 = a$, or
2. $\exists a_3$ such that a_2 is chained to a_3 , and either
 - (a) a_1 is in the same statement as a_2 , or

(b) a_1 dataflow depends [23] on a_2 .⁷

For example, in the following loop nest, $\mathcal{V}_A = \{D, C, A, B, E, F, G\}$.

```

do j = 1 to M
  do i = 1 to N
    D[i] = C[i] + A[B[i]] + E[j]
  do i = 1 to N
    F[i] = G[i] + D[i]
    H[i] = K[i] + L[i]
```

The condition for eliminating σ_i is given as follows. If we chose an array A from Section 4.1, then \mathcal{V}_A must include all references in the loop nest, and we must have remapped all $a \in \mathcal{V}_A$.

Cost consequences: Collapsing does not affect setup overhead. It reduces the main computation's overhead by reducing loop overhead; further, in tandem with remapping, it allows the remapped arrays to have the same dimensionality as their original counterparts. If the original arrays were one-dimensional, normal remapping produces two-dimensional remapped versions. With collapsing, we can use one-dimensional remapped versions, hence reducing addressing arithmetic. Collapsing does not affect memory requirements.

6. Data Remapping

The final aspect of bucket tiling is data remapping.

Mechanism: Remapping replaces a reference to array a with one to a new array $a'[b, k]$, where $1 \leq b \leq \text{numbuckets}$, and $1 \leq k \leq \text{bucketsize}[b]$. The details of remapping and its consequences on performance varies with two factors: whether a is *exposed* and the relation between \mathcal{I}_a and \mathcal{A}_f , for the f chosen in Section 4.

First, where we need to copy depends on *how* a is exposed. An array reference is *upwards-exposed* if a dataflow edge has a source outside and a sink inside this computation; similarly for *downwards-exposed* [26]. If a is not exposed, then textually replace every occurrence of a with a' . If a is upwards-exposed, we must “gather” a into a' using the permutation generated in Section 4. If a is downwards exposed, we must “scatter” a' back to a , according to the inverse of the generated permutation.

Second, whether we need a final reduction operation depends on the relation between \mathcal{I}_a and \mathcal{A}_f . If $\mathcal{I}_a = \mathcal{A}_f$ (i.e. i_a depends on all affected loops' induction variables), then we do not need a reduction. However, if $\mathcal{I}_a \subset \mathcal{A}_f$, then remapping has only generated a set of partial computations. We must use a *scatteradd* operation [31], to reduce the set of partial computations. For example, consider assignments

to $D[i]$, $D[j]$, and $D[i, j]$ The first two require a *scatteradd* reduction, whereas the last one does not.

Cost consequences: How does remapping impact the three performance metrics? First, it has a negative effect on setup overhead, as we now must gather and scatter the remapped arrays.

On the other hand, remapping positively affects the main computation. Data remapping eliminates references to original induction variables and reduces the main computation's spatial footprint from sparse-sequential to sequential. If we remove all uses of an original induction variable i , we need not load σ_i . Also, if remapped arrays are not downwards-exposed, we can further eliminate the corresponding σ . However, if we remap a downwards-exposed reference, we need to maintain the σ 's to perform the scatter operation.

Finally, remapping increases memory requirements. The increase for an reference a is as follows. If $\mathcal{I}_a = \mathcal{A}_f$, then remapping creates a' with the same size as a ($|a'| = |a|$); remapping, in this case, adds a to the memory requirements. If $\mathcal{I}_a \subset \mathcal{A}_f$, then remapping creates an a' with a potentially larger footprint (c.f. *array privatization* [21, 24]). How much do we expand storage? The final size in all cases is the number of iterations in \mathcal{A}_f . The initial size of a reference a is the number of iterations in \mathcal{I}_a . For example, if the above loops contained $D[i] += C[j, i] * A[B[j]]$, then $|C'| = |C|$, $|B'| = N|B|$, and $|D'| = M|D|$.

7. Optimization Examples

In this section, we apply our technique to three codes: a computation used in the simulating of a heart, sparse matrix-vector product, and integer sort.

7.1. Heart loop

First, we study a kernel used by scientists to simulate the beating of a heart. The kernel is a five-deep loop nest:

```

do p = 1 to N
  do d = 1 to 3
    do k, j, i = 1 to 4
      A[d, p] = A[d, p] + Δ[i, j, k] *
        fluid[I[p] + i, J[p] + j, K[p] + k, d]
```

The straightforward code given above performs exceptionally poorly for two reasons. First, notice that the iteration counts of the inner loops are small. Therefore, the original implementation loads in the long TLB lines but does not fully use them before they are replaced. We can remedy this problem by referencing the **fluid** array with d in the fastest (left-most for Fortran) dimension. This fix provides a fairer baseline for our improvements. The second reason for poor performance is the non-affine components of **fluid**'s index expression, those referencing the I , J , and K arrays. The I , J , and K arrays represent the changing location of the muscle fibers as the heart beats. Since this

⁷A true dependence exists from creation to use of variables with the same name. Dataflow, or *value*, dependences restrict this definition to variables of the same value.

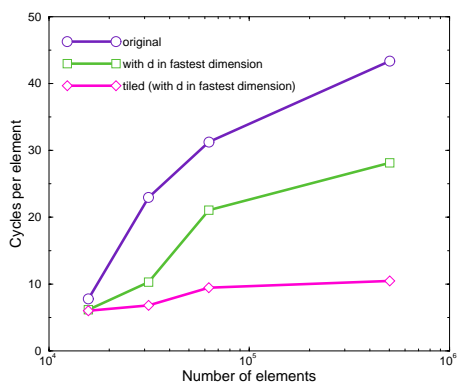


Figure 4. Bucket tiling the heart loop.

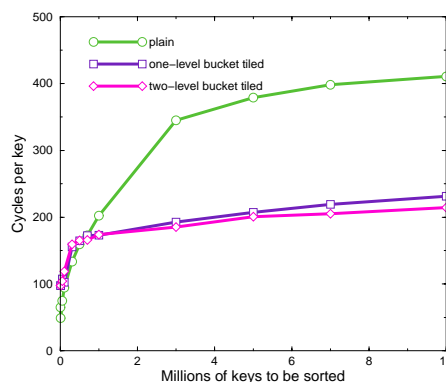


Figure 5. Bucket tiling integer sort.

varies dramatically over time, it will exhibit properties of randomness. For the purposes of our experiments we initialized I , J , and K with random values; if the computation performs well in this setting, we predict it will perform well with the actual values.

To optimize this code, we chose to use 1000 buckets. The following code gives a high-level view of the bucket tiled code. In the real implementation, we used a version of *permute* specialized to also perform the remappings. In the future, we will work on our *permute*-generator, *pgen* (from Section 4.3), to perform this optimization.

```

 $\sigma_i = \text{permute}(1000)$ 
 $I' = \text{remap}(I, \sigma_i)$ 
 $J' = \text{remap}(J, \sigma_i)$ 
 $K' = \text{remap}(K, \sigma_i)$ 
 $A' = \text{remap}(\text{newdata}_{\text{opt}}, \sigma_i)$ 
... original, with  $A \rightarrow A', I \rightarrow I', J \rightarrow J', K \rightarrow K' \dots$ 
 $A = \text{remap}(A', \sigma_i^{-1})$ 
    
```

We experimented with three variants: the original code, a tweaked version with d in the fastest dimension, and a bucket tiled tweaked version. Figure 4 shows that on the largest problem size ($N = 502691$), we achieve a speedup of 2.69 versus the tweaked implementation and 4.16 versus the original implementation.

7.2. Integer sort

Next, we look at integer sort. Our baseline implementation comes from NAS version 1.⁸

```

do i = 1 to numkeys
    keyden[key[i]]++
do i = 2 to maxkey
    keyden[i] += keyden[i - 1]
do i = 1 to numkeys
    rank[i] = keyden[key[i]]--
    
```

Figure 5 compares the performance of untiled and tiled two-million-key integer sort on a Alpha 21164a. We experimented with a range of problem sizes, as shown on the

⁸Version 2.3 of the IS benchmark performs only the first two phases of an integer sort. It does not generate a sorted array.

horizontal axis; we set the number of keys equal to the key range for each problem size. For a small number of keys, the reduction in cache and TLB misses does not outweigh the increased overhead introduced by tiling. When sorting ten million keys, the one-level tiled version is 1.78 times faster than unoptimized code.

We also experimented with a *two-level* bucket-tiled implementation. This is an instance of *multi-level bucket tiling*, which we will explore in future research. We briefly describe multi-level bucket tiling in Section 8.2. The two-level tiling improves performance over the one-level implementation, achieving a speedup of 1.91.

7.3. Sparse matrix-vector

As our last example, we study sparse matrix-vector product. This computation occurs in many scientific computations, including conjugate gradient and eigenvalue solvers.

```

do j = 1 to N
    sum = 0
    do k = rowstart[j] to rowstart[j + 1]
        sum += A[k] * X[colindex[k]]
    Y[j] = sum
    
```

Figure 6 shows the results for conjugate gradient (CG), from the NAS Parallel Benchmarks version 2.3, which includes sparse matrix-vector product as its primary computation. We experimented with a range of matrix sizes. While we tiled only the sparse matrix-vector product portion of the computation, we nonetheless achieve pleasant speedups for even the smallest problem sizes. With a matrix of 5.495 million nonzero elements, the bucket tiled code is 1.57 times faster than the vanilla code. Remember that this speedup is for the entire conjugate gradient calculation, even though we only optimized the sparse matrix-vector product.

Unlike integer sort, conjugate gradient does not suffer from TLB misses. However, integer sort must recompute the inspector phase for every sort. Conjugate gradient, on the other hand, uses a fixed sparse matrix, with varying vector. Therefore, with conjugate gradient, we can extract the

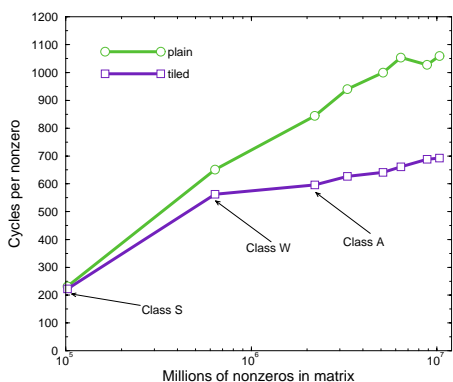


Figure 6. Bucket tiling conjugate gradient.

inspector phase out of all loops, thus amortizing the overhead across the entire computation.

8. Future Work

In this section, we describe three issues of future work.

8.1. Parameters for performance prediction

Under what circumstances is our technique *profitable*? For example, the NAS EP benchmark contains non-affine array references which use a very small segment of memory. Naively applying bucket tiling to EP would add overhead, without an offsetting decrease in cache or TLB misses.

Based on our observations in Section 7, we hypothesize that performance varies with five aspects of the data and computation: randomness, range, non-affine extent, affine extent, and reuse. This paper illustrates the effect of the first four on performance, in Figures 7(a)-(d). We performed these experiments on a 500MHz Alpha 21164a. Figure 7(a) shows that the benefit of bucket tiling $A[B[i]]++$ depends on the *randomness* of B ; (b) shows that the benefit for the same loop depends on the *range* of values contained in B . For (c) and (d) we experimented with a doubly-nested loop computing $A[B[i]][j]$ and $A[j][B[i]]$; (c) shows that the benefit depends on the extent of the non-affine portion (extent of i , in this case); (d) shows that the benefit depends on the extent of the affine portion (extent of j).

8.2. Multi-level bucket tiling

Figure 8 visualizes *multi-level* bucket tiling for a one-dimensional loop nest containing $A[B[i]]$.⁹ The one-level technique we have described in this paper corresponds to

⁹Compare this figure with the plots in Figure 2. In those figures, the vertical axis was “memory location” and the horizontal axis was “time”. Figure 8 generalizes this concept.

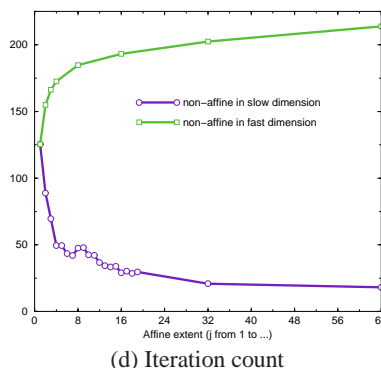
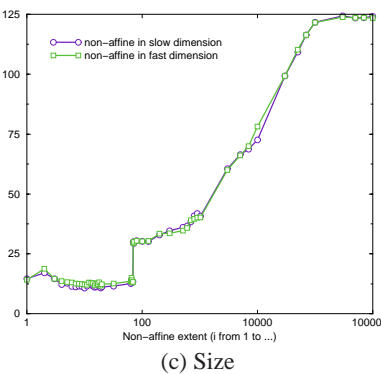
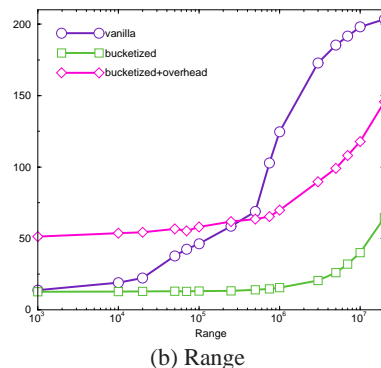
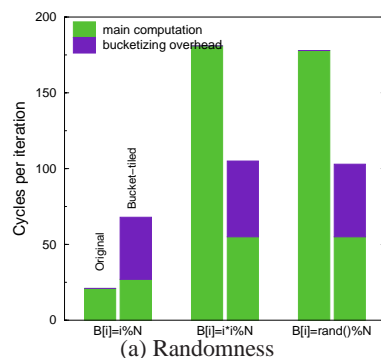


Figure 7. How performance varies with four aspects of the computation and data.

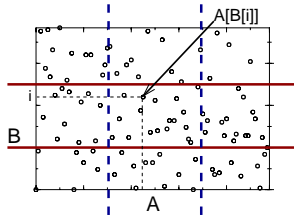


Figure 8. An illustration of two-level bucket tiling, for $A[B[i]]$. Each level is a family of cuts: the horizontal lines, and the vertical lines.

horizontal cuts of the data space. A k -level bucket tiling creates k cuts, either horizontal or vertical. We experimented with using two cuts: one horizontal and one vertical. In [25], we explored this strategy on a simultaneous multithreading processor (SMT). For SMT, we found that this two-level strategy greatly benefits SMT, but does not benefit a standard shared memory multiprocessor. As Figure 5 shows, two-level bucket tiling is of positive, but not substantial benefit for a Alpha 21164a, a conventional superscalar processor.

8.3. Unification Issues

Some of the bucket tiling mechanisms are likely to be independent of the others. For example, spotting A and predicting performance depend only on features of the data and the original computation. Other steps have interdependencies. For example, a choosing f interacts with bucket scanning. Yet the factors represent incomparable quantities. How do we compare increased storage requirements due to remapping with number of cache misses or increased computation due to bucketizing?

In addition, the choices made for one reference might impact the choices made for others. As an example, consider integer sort, where one array is referenced indirectly in two places. If we naively apply the mechanisms presented in this paper to each of these occurrences, we will have introduced twice the overhead necessary. As a more complicated example, imagine a code which accesses an array in two locations with two different non-affine index expressions. The solution to the global problem is an instance of Mace and Wagner’s global shape problem [22].

9. Related Work

Finally, we summarize related work not considered in Section 1.

Unstructured code optimization: Work on optimization in the presence of non-affine array references is sparse. A combined compile- and run-time extension of tiling for

loops with pointer-based data structures [36] was developed in the Illinois Concert System. Their technique focuses on thread-based parallelization of object-oriented programs, unlike our work which focuses on locality for C and Fortran programs.

Sparse-matrix computations: Im and Yelick [12] optimize sparse matrix-vector product using machine models and models of sparse matrix structure. Given these models, their technique packs a sparse matrix in a format amenable to cache or register tiling. The Sparse-BLAS [4] library extends the Basic Linear Algebra Subroutines to sparse matrix computations. Bik [3] and Kotlyar [19] generate a sparse version of a given dense computation. Knijnenburg and Wijshoff [17] match patterns in the sparse matrices against a set of templates, such as block or diagonal. For each template, they have a tailored solution. While these works offer portability, they are specific to sparse computations and limited to the subroutines the developers have provided.

Inspector-executor: Three groups have applied the inspector-executor technique [30, 28] to parallelizing unstructured codes on distributed memory machines. Das *et al.* introduced index array flattening [9]. In that paper, the authors developed a technique for parallelizing codes with indirect memory references. They did not discuss the issue of determining profitability. If the user specifies (perhaps irregular) data objects and tasks, Fu and Yang’s RAPID system [11] partitions the sparse-matrix tasks to processors. Rauchwerger *et al.* use an inspector to discover candidate arrays for privatization and operations for reduction [29].

10. Conclusion

To optimize non-affine array references requires coping with unpredictable access to a potentially large segment of memory. There are two problems with this type of access. First, being to a large segment of memory means that cache performance will suffer. Second, being unpredictable, we cannot use a compile-time solution to improve performance. Our solution, bucket tiling, replaces large-unpredictable access with unpredictable access to a small, controllable segment. Consequently, it does not matter that the accesses are to unpredictable locations.

We accomplish this transformation by *bucketizing*, at run-time, a portion of the index function of the problematic array reference. Bucketizing has the nice property that, not only does it control the size of the unpredictable accesses, but the bucketizing itself performs well.

In this paper, we provided mechanisms necessary for automating our technique. Further, we defined the cost consequences for each mechanism. We briefly defined *performance parameters* necessary for predicting the performance of a non-affine array reference. Finally, we applied bucket tiling to three computations: integer sort, conjugate gra-

dient, and a loop nest used in simulations of heart's beating. We observed sizeable speedups, due to substantially reduced cache and TLB misses.

References

- [1] C. Ancourt and F. Irigoien. Scanning polyhedra with DO loops. In *Principles and Practice of Parallel Programming*, pages 39–50, Apr. 1991.
- [2] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.
- [3] A. J. C. Bik. *Compiler Support for Sparse Matrix Computations*. PhD thesis, Leiden University, The Netherlands, May 1996.
- [4] S. Carney, M. A. Heroux, G. Li, R. Pozo, K. A. Remington, and K. Wu. A revised proposal for a sparse BLAS toolkit. Technical report, National Institute of Standards and Technology, 1996. <http://math.nist.gov/spblas>.
- [5] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1994.
- [6] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Programming Language Design and Implementation*, La Jolla, CA, June 1995.
- [7] M. Cierniak and W. Li. Interprocedural array remapping. In *Parallel Architectures and Compilation Techniques*, San Francisco, CA, Nov. 1997.
- [8] A. Darte. On the complexity of loop fusion. Technical Report RR1998-50, Ecole Normale Supérieure de Lyon, Oct. 1998.
- [9] R. Das, P. Havlak, J. Saltz, and K. Kennedy. Index array flattening through program transformation. In *Supercomputing*, 1995.
- [10] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Programming Language Design and Implementation*, 1999.
- [11] C. Fu and T. Yang. Run-time compilation for parallel sparse matrix computations. In *International Conference on Supercomputing*, pages 237–244, May 1996.
- [12] E.-J. Im and K. Yelick. Model-based memory hierarchy optimizations for sparse matrices. In *Workshop on Profile and Feedback-Directed Compilation*, Oct. 1998.
- [13] F. Irigoien and R. Triolet. Supernode partitioning. In *Principles of Programming Languages*, pages 319–328, Jan. 1988.
- [14] M. T. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matrix-based approach to the global locality optimization problem. In *Parallel Architectures and Compilation Techniques*, Oct. 1998.
- [15] M. T. Kandemir, A. Choudhary, J. Ramanujam, N. Shenoy, and P. Banerjee. Enhancing spatial locality via data layout optimizations. In *Workshop on Automatic Parallelisation*, Southampton, UK, Sept. 1998.
- [16] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Workshop on Languages and Compilers for Parallel Computing*, 1993.
- [17] P. M. W. Knijnenburg and H. A. Wijshoff. On improving data locality in sparse matrix computations. Technical Report 94-15, Leiden University, Netherlands, 1994.
- [18] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Programming Language Design and Implementation*, June 1997.
- [19] V. Kotlyar, K. Pingali, and P. Stodghill. Compiling parallel code for sparse matrix applications. In *Supercomputing*, Nov. 1997.
- [20] S.-T. Leung. *Array Restructuring for Cache Locality*. PhD thesis, University of Washington, Aug. 1996.
- [21] Z. Li. Array privatization for parallel loop execution. In *International Conference on Supercomputing*, Washington, D.C., July 1992.
- [22] M. E. Mace and R. A. Wagner. Globally optimum selection of memory storage patterns. In *International Conference on Parallel Processing*, pages 264–271, 1985.
- [23] D. E. Maydan. *Accurate Analysis of Array References*. PhD thesis, Stanford University, Sept. 1992.
- [24] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array data-flow analysis and its use in array privatization. In *Principles of Programming Languages*, Jan. 1993.
- [25] N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen. ILP versus TLP on SMT. In *Supercomputing*, Nov. 1999.
- [26] S. S. Munchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [27] C. D. Polychronopoulos. Loop coalescing: A compiler transformation for parallel machines. In *International Conference on Parallel Processing*, pages 235–242, St. Charles, IL, Aug. 1987.
- [28] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. In *Supercomputing*, pages 361–370, Portland, OR, 1993. IEEE Computer Society Press.
- [29] L. Rauchwerger, N. M. Amato, , and D. A. Padua. Runtime methods for parallelizing partially parallel loops. In *International Conference on Supercomputing*, pages 137–146, Barcelona, Spain, July 1995.
- [30] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message-passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, Apr. 1990.
- [31] J. Saltz, R. Ponnusamy, S. Sharma, B. Moon, Y.-S. Hwang, M. Uysal, and R. Das. Manual for the chaos runtime library. Technical Report CS-TR-3437, University of Maryland, Department of Computer Science, Mar. 1995.
- [32] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Programming Language Design and Implementation*, 1991.
- [33] M. E. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *International Symposium on Microarchitecture*, Dec. 1996.
- [34] M. J. Wolfe. Iteration space tiling for memory hierarchies. In *Parallel Processing for Scientific Computing*, pages 357–361, 1987.
- [35] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
- [36] X. Zhang and A. A. Chien. Dynamic pointer alignment: Tiling and communication optimizations for parallel pointer-based computations. In *Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.