Array Language Support for Wavefront and Pipelined Computations[*]

Bradford L. Chamberlain E Christopher Lewis Lawrence Snyder

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350

October 29, 1998

Abstract

Array languages such as Fortran 90, High Performance Fortran and ZPL are convenient vehicles for expressing data parallel computation. Unfortunately, array language semantics prohibit the natural expression of wavefront and pipelined computations, characterized by a sequential propagation of computed values across one or more dimensions of the problem space. As a result, programmers scalarize (*i.e.*, use loop nests and scalar indexing instead of array operations) wavefront computations, sacrificing the benefits of the array language. We propose an extension to array languages that provides support for wavefront computation without scalarization and with minimal impact on the language. Our extension is particularly valuable in that it identifies parallelism to both the programmer and compiler just as conventional array operations do. In this paper we motivate the problem, introduce our language extension, describe its implementation in the ZPL data parallel array language compiler, and experimentally evaluate the parallel performance improvement due to its optimization for parallelism.

1 Introduction

Array languages such as Fortran 90 [1], High Performance Fortran (HPF) [6] and ZPL [12] have achieved success in expressing and exploiting data parallelism. They are distinguished from scalar languages by their support of operations on arrays as primitive entities, frequently obviating the need for explicit looping and element-wise indexing. In addition to providing convenient and concise syntax, array operations enable programmer/compiler collaboration, the basis of effective parallel computing. The programmer and compiler collaborate in the sense that the programmer presents a representation of a particular computation and has certain expectations about its eventual performance. If a programmer expresses a computation in terms for which the compiler is unable to meet the performance expectations, collaboration does not exist. Effective programmer/compiler collaboration is essential in the parallel domain, where orders of magnitudes of performance may be lost when the compiler is

[*]This research was supported in part by DARPA Grant E30602-97-1-0152, NSF Grant CCR-9710284 and the Intel Corporation.

```
      DO 100 i = 2 , n-1
         DO 100 j = 2 , n-2
            r=aa(j,i)*d(j-1,i)
            d(j,i)=1.0/(dd(j,i)-aa(j-1,i)*r)
            rx(j,i)=rx(j,i)-rx(j-1,i)*r
            ry(j,i)=ry(j,i)-ry(j-1,i)*r
  100 CONTINUE
```

(a)

```
         DO 100 j = 2 , n-2
            r(2:n-2)=aa(j,2:n-2)*d(j-1,2:n-2)
            d(j,2:n-2)=1.0/(dd(j,2:n-2)-aa(j-1,2:n-2)*r(2:n-2))
            rx(j,2:n-2)=rx(j,2:n-2)-rx(j-1,2:n-2)*r(2:n-2)
            ry(j,2:n-2)=ry(j,2:n-2)-ry(j-1,2:n-2)*r(2:n-2)
     100 CONTINUE
```

(b)

Figure 1: Fortran code fragments from SPECfp92 Tomcatv benchmark demonstrating a limitation of array languages. In order to represent the scalar code in (a) in an array language, a programmer must scalarize the first dimension, as in (b).

unable to produce a parallel implementation of a code. Array languages enable effective collaboration by making parallelism apparent to both the programmer and the compiler in array operations.

Unfortunately, array languages are limited in what computations they can express at the array level. This is because an array language compiler generates a loop nest to implement one or more array statements, and array language semantics prohibit the loop from carrying true data dependences from one statement to its self or earlier statements. If programmers want to express such a computation, they must *scalarize* one or more dimensions of the computation by using loops to explicitly iterate over elements of arrays. Clearly, scalarization is at odds with the goals and benefits of array language programming. Computations characterized by a data dependent flow of values across one or more dimensions of the problem space cannot be expressed only via primitive array operations. We call these *wavefront* computations, because waves of computed values wash across the problem space. They can be thought of as a generalization of scan or prefix computations.

As an example, consider the scalar Fortran 77 code fragment in Figure 1(a); it is from the tridiagonal systems solver component of the SPECfp92 Tomcatv benchmark. The inner loop carries a true data dependence from the second statement to the first due to array `d` and from the third and fourth statements to themselves due to arrays `rx` and `ry`, respectively. An array language can not express these dependences without scalarizing the first dimension of the problem space, called *partial scalarization*, as in the Fortran 90 array statements in Figure 1(b). Clearly, scalarization sacrifices the benefits of array languages, corrupting the collaboration between programmer and compiler. The programmer must decide whether the partially scalarized wavefront computation is phrased in terms that the compiler can recognize and parallelize. In addition, the scalarized code in Figure 1(b) has very poor cache behavior because all the array references are to rows and the arrays are allocated in column-major-order. The compiler might optimize the partially scalarized code for spatial locality and parallelism, but the programmer is left to wonder and is ill-equipped to make implementation decisions. We will demonstrate in Section 5 that the potential performance differential is enormous. The bottom line is that scalarization impedes effective collaboration between programmer and compiler. If a full array representation of wavefront computations could be expressed, it would regain the benefits of array language programming.

We propose an extension to array languages to directly support wavefront computations, eliminating the need for partial scalarization, with minimal impact on the language. Though wavefront computations do not appear in every array language program, they are sufficiently common that they warrant language support. There are two components to our solution. First, we introduce a new array operator, called the *prime* operator, that allows a programmer to refer to values written inside the loop nest that implements the statement containing the prime. Second, we introduce a new compound statement, called the *scan block*, that groups statements in order to establish the scope of the prime operator. These two new array language features permit full array statement representation of codes like that of Figure 1, thereby restoring the benefits of array languages and conveying the high level nature of the computation to the compiler. Most importantly, the programmer need not speculate whether the compiler is able to extract parallelism from the scalarized code. Just as parallelism is manifest in array operations, it is manifest in scan blocks.

This paper is organized as follows. In the next section, we describe our array language extension to support wavefront computations in ZPL. Sections 3 and 4 describe its implementation and optimization, respectively. Performance data is presented in Section 5, and future work and conclusions are given in the final section.

2 Array Language Support for Wavefront Computation

This section describes array language support for wavefront computation in the context of the ZPL parallel array language [12]. We have demonstrated that our ZPL compiler is competitive with hand-coded C with MPI [3], and it generally outperforms HPF [9]. The compiler is publicly available [14] for most modern parallel and sequential platforms, including the Cray T3E, IBM SP2, SGI PowerChallenge, SGI Origin, and networked UNIX workstations using MPI and PVM. The language is in active use by scientists in fields such as astronomy, civil engineering, biological statistics, mathematics, oceanography, and theoretical physics. The first section below gives a very brief summary of the ZPL language, only describing the features of the language immediately relevant to this paper. Detailed coverage of the language may be found elsewhere [12]. After that, we introduce the prime operator and scan blocks as a means of supporting wavefront computations in ZPL.

2.1 Brief ZPL Language Summary

ZPL is a data parallel array programming language. It supports all the usual scalar data types (*e.g.*, `integer`, `float`, `char`), operators (*e.g.*, math, logical, bit-wise), and control structures (*e.g.*, `for`, `while`, function calls). As an array language, it also offers array data types and operators. ZPL is distinguished from other array languages by its use of *regions* [4]. A region represents an index set, and may precede a statement, specifying the extent of the array references within its dynamic scope. By factoring the indices that are to be computed on into the region, the use of regions eliminates the need to index arrays. For example, the following Fortran 90 (slice-based) and ZPL (region-based) array statements are equivalent.

```
a(n/2:n,n/2:n) = b(n/2:n,n/2:n) + c(n/2:n,n/2:n)      [n/2..n,n/2..n] a = b + c;
```

```
for j := 2 to n-2 do
[j,2..n-1] begin
             r = aa * d@north;
             d = 1.0 / (dd - aa@north * r);
             rx = rx - rx@north * r;
             ry = ry - ry@north * r;
           end;
end;
```

(a)

```
[2..n-2,2..n-1] scan
                  r = aa * d'@north;
                  d = 1.0 / (dd - aa@north * r);
                  rx = rx - rx'@north * r;
                  ry = ry - ry'@north * r;
                end;
```

(b)

Figure 2: ZPL representations of the Tomcatv code fragment from Figure 1. The first code fragment (a) uses scalarization to express the wavefront, while the second (b) uses a scan block and the prime operator.

Regions can be named and used symbolically to further improve readability and conciseness. When a computation requires that not all array references refer to exactly the same set of indices, array operators are applied to individual references, selecting some function of the applicable region's indices from the the operators' operands. ZPL provides a number of array operators (*e.g.*, reductions, parallel prefix operations, broadcasts, general permutations), but for this discussion, we will only discuss the shift operator. The shift operator, represented by the `@` symbol, shifts the indices of the covering region by some offset vector, called a *direction*, to determine the indices of its argument array that are involved in the computation. For example, the following Fortran 90 and ZPL statements perform the same stencil computation. Note that `north`, `south`, `west`, and `east` represent the programmer defined vectors $(-1,0)$, $(1,0)$, $(0,-1)$, and $(0,1)$, respectively.

```
a(2:n+1,2:n+1) = (b(1:n,2:n+1)+b(3:n+2,2:n+1)+b(2:n+1,1:n)+b(2:n+1,3:n+2))/4.0
            [2..n+1,2..n+1] a := (b@north+b@south+b@west+b@east)/4.0;
```

Figure 2(a) contains a ZPL code fragment representing the same computation as the Tomcatv Fortran 90 code fragment in Figure 1(b). The use of regions improves code clarity and compactness. Though the scalar variable `r` is promoted to an array in the array codes, we have previously demonstrated compiler techniques by which this overhead may be eliminated via array contraction [8].

2.2 Wavefront Computation in ZPL

Array language semantics dictate that the right-hand side of an array statement is evaluated before the result is assigned to the left-hand side. As a result, the compiler will not generate a loop that carries a true data dependence. For example, the ZPL statement in Figure 3(a) is implemented by the loop nest in 3(b). The compiler determines that the *i*-loop iterates from high to low indices in order to ensure that the loop does not carry a true data dependence. If array `a` contains all 1s before the statement in 3(a) executes, it will have the values in Figure 3(c) afterward.

In wavefront computations, the programmer wants the compiler to generate a loop nest with loop carried true data dependences. We introduce a new operator, called the *prime* operator, that allows a programmer to reference values written in previous iterations of the loops that contain the primed reference. For example, the

```
[2..n,1..n] a := 2 * a@north;
```

(a)

```
for $i \leftarrow n$ downto 2 do
    for $j \leftarrow 1$ to $n$ do
        $a_{i,j} \leftarrow 2 * a_{i-1,j}$
```

(b)

$$a = \begin{array}{ccccc} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{array}$$

(c)

```
[2..n,1..n] a := 2 * a'@north;
```

(d)

```
for $i \leftarrow 2$ to $n$ do
    for $j \leftarrow 1$ to $n$ do
        $a_{i,j} \leftarrow 2 * a_{i-1,j}$
```

(e)

$$a = \begin{array}{ccccc} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 4 & 4 & 4 & 4 & 4 \\ 8 & 8 & 8 & 8 & 8 \\ 16 & 16 & 16 & 16 & 16 \end{array}$$

(f)

Figure 3: ZPL array statements (a and d) and the corresponding loop nests (b and e) that implement them. The arrays in (c and f) illustrate the result of the computations if array `a` initially contains all 1s.

ZPL statement in Figure 3(d) is implemented by the loop nest in 3(e). In this case, the compiler must ensure that a loop carried true data dependence exists due to array `a`, thus the *i*-loop iterates from low to high indices. If array `a` contains all 1s before the statement in 3(d) executes, it will have the values in Figure 3(f) afterward. In general, the direction on the primed array reference defines the orientation of the wavefront.

The prime operator alone cannot represent a complex wavefront such as the Tomcatv code fragment in Figure 2(a), because alone it only permits loop carried true dependences from a statement to itself. We introduce a new compound statement, called a *scan block*, to allow for more complex wavefronts. Primed array references in a scan block refer to values written by any statement in the block, not just the statement that contains it. For example, the ZPL code fragment in Figure 2(b) uses scan blocks and the prime operator to realize the computation in Figure 2(a) without partial scalarization. The array reference `d'@north` refers to values from the previous iteration of the loop that iterates over the first dimension. Thus the primed `@north` references imply a wavefront that travels from north to south. Non-primed references have the usual meaning. In scan blocks, they refer to the values in the array before the scan block was entered.

The scan blocks we have looked at thus far contain only cardinal directions (*i.e.*, directions in which only one dimension is nonzero, such as north, south, east and west). When noncardinal directions appear with primed references, there are nested wavefronts. In this situation, multiple loops carry dependences due to the primed array. This is a logical extension of wavefronts as we have described them. Because nested wavefronts appear far less frequently and they are not a source of significant parallelism, we do not consider them further in this paper.

There are a number of statically checked legality conditions. (i) Primed arrays in a scan block must also be defined in the block; (ii) the directions on primed references may not overconstrain the wavefront (*e.g.*, primed `@north` and `@south` references are not permitted because they imply both north-to-south and south-to-north wavefronts, which are contradictory); (iii) all statements in a scan block must have the same rank (*i.e.*, are implemented by a loop nest of the same depth)—this precludes the inclusion of scalar assignment in a scan block; and (iv) array operands to parallel operators other than the shift operator may not be primed; this is essential because array operators are pulled out of the scan block during compilation (see Section 3).

The prime operator and scan blocks can be added to a language such as Fortran 90 in an analogous way. Again, the wavefront orientation is defined by primed shifted array references. Though it is somewhat less obvious to a programmer how arrays are shifted in slice notation versus a region-based approach, the compilation approach is the same, which we discuss in the next section.

3 Implementation

This section describes our approach to implementing primed array references and scan blocks in the ZPL compiler. First, we show how data dependences determine loop structure in the ZPL compiler. Next, we show how to leverage this infrastructure to determine the loop structure of scan blocks. Finally, we describe the compiler's approach to communication insertion.

3.1 Deriving Loop Structure in ZPL

The ZPL compiler identifies groups of statements that are to be implemented as a single loop nest, essentially performing loop fusion. The data dependences (true, anti and output) that exist between these statements determine the structure of the resulting loop nest. Specifically, they determine what loop iterates over each array dimension and in what direction (*e.g.*, from high to low indices). The ZPL and scalar code fragments in Figure 3(a) and (b) illustrate this. Notice that the j-loop has to iterate from high to low indices in order to preserve the loop carried anti-dependence from the statement to itself.

We have developed *unconstrained distance vectors* to represent these array-level data dependences [8]. Unconstrained distance vectors are analogous to conventional distance vectors [13] except that they are independent of the iteration space. This is essential in an array language compiler because analyses are performed before the array statements have been converted to loop nests, thus before an ordered iteration space exists. The unconstrained representation is enabled by the regularity of the loop nests that are generated for array statements. An unconstrained distance vector associated with a particular dependence is simply the vector difference of the direction at the source and target of the dependence. For example, the unconstrained distance vector due to the anti-dependence from reference `a@north` to `a` in Figure 3(a) is simply $(-1,0)-(0,0)=(-1,0)$. We have previously developed an algorithm that computes a legal loop structure given a list of the unconstrained distance vectors associated with the data dependences between the statements that the loop is to contain [8]. In the event that the unconstrained distance vectors do not uniquely determine a loop structure, cache issues are considered in order to improve spatial locality. In the event that the unconstrained distance vectors over-constrain the loop and no loop structure can be found, all the statements cannot appear in a single loop. Some of them must be placed in separate loops thereby eliminating some dependences and their associated unconstrained distance vectors.

3.2 Deriving Loop Structure of Scan Blocks

In order to simplify analysis, all potentially parallel array operators (except shifted primed arrays) are removed from scan blocks via the insertion of temporary arrays. These temporary arrays will be subsequently eliminated via array contraction if possible [8].

Though all the statements in a scan block are implemented by a single loop nest, we cannot simply calculate unconstrained distance vectors as described above, because primed references have a different meaning. For example, consider the scan block in Figure 2(b). Normal unconstrained distance vector calculation creates an anti-dependence from the first to the second statement due to array d. But because the reference to d is primed, there should actually be a true data dependence from the second statement to the first. We represent this by simply negating the unconstrained distance vector between the two statements. Note that primed array references only appear on the right-hand side of array statements, so only unconstrained distance vectors due to anti-dependences need to be negated when the apparent source of the dependence is primed. Finally, we use our existing algorithm to derive loop structure from the unconstrained distance vectors associated with inter scan block dependences.

3.3 Communication

Next, we consider the issues of generating communication for scan blocks when the arrays they contain are distributed. ZPL aligns interacting arrays so that interprocessor communication is apparent to the programmer and the compiler from the parallel array operators, such as the shift operator [2]. Because only the shift operator on primed arrays may appear in scan blocks, we need only describe its implementation here.

Recall that shifted primed arrays imply that there is a wavefront moving across the arrays in a scan block. In a naive implementation, each scan block is preceded by a message receive and followed by a message send. Thus, a particular processor will not enter the loop nest that computes its portion of a scan block until the data from the previous processors becomes available. When a processor completes this loop nest, it sends the data required by subsequent processors to them. The communication has the effect of serializing all the computation along the direction of the wavefront, which is the intended meaning of the wavefront computation. Figure 4(a) illustrates this interprocessor communication. This simplistic implementation does not exploit any parallelism along the wavefront dimension. The next section discusses optimizations for parallelism.

4 Optimization

This section discusses a number of optimizations that improve the parallel performance of scan blocks.

4.1 Pipelining

In the implementation described above, a processor finishes computing on its entire portion of a scan block before data is sent on to later processors in the wavefront computation. As a result, there is no parallelism along the *wavefront dimension*—the dimension across which the wavefront travels. The computation is sequential and

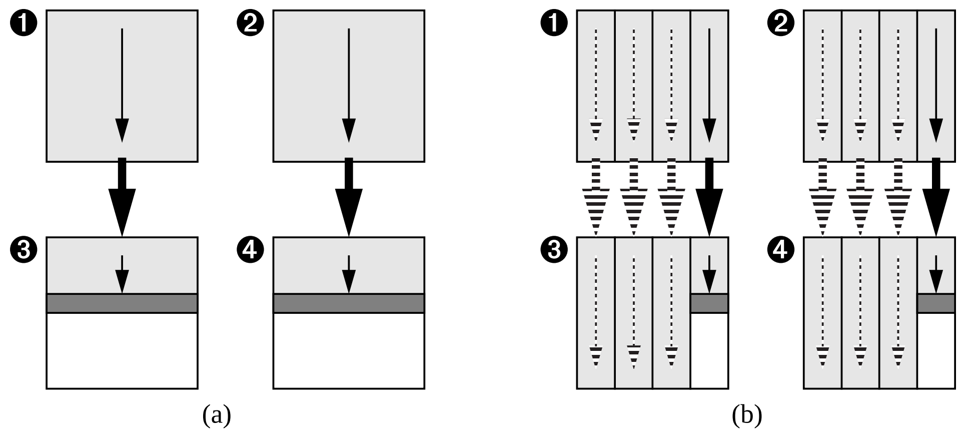

Figure 4: Illustration of the data movement and parallelism characteristics of wavefront computations (a) without and (b) with pipelining.

processors sit idle. Suppose a north-to-south wavefront computation is performed on an $n \times n$ array distributed across a 2×2 processor mesh as in Figure 4(a). Processors 3 and 4 must wait for processors 1 and 2 to compute on $n^2/4$ elements each before they may proceed. Furthermore, processors 1 and 2 will have to wait for the others to complete if they depend on the global result of the wavefront computation.

Alternatively, the wavefront computation may be *pipelined* in order to exploit parallelism. Specifically, a processor may compute a small slice of its portion of a scan block, send on some of the data needed by subsequent processors, then continue to execute its next slice. The benefit of this approach is that it allows multiple processors to become involved in the computation as soon as possible, greatly improving parallelism. Figure 4(b) illustrates this. Processors 3 and 4 only need to wait long enough for processors 1 and 2 to compute a single slice ($n/4 \times 1/4 = n/16$ elements) each. They can then immediately begin computing slices of their portions of the scan block. By the time they are finished with a slice, the next bit of data has been passed on by the preceding processor. Parallel implementation of seemingly sequential wavefront computations is not a new idea [10, 11], but providing direct array language support for it is.

The compiler performs this optimization by generating a loop to iterate over slices of a processor's portion of a scan block. As in the unpipelined case, communication routines and a loop nest to iterate over the slice appear in the loop body. In each iteration of this loop, each processor only computes on a slice of its portion of the scan block. This slice includes the processor's entire range of the wavefront dimension, but only part of the other dimensions. The size of the slice in the non-wavefront dimensions are determined by factors such the relative cost of communication versus computation and cache characteristics.

Clearly, the number of slices impact parallelism. Ignoring communication costs, we expect better parallelism for a larger number of slices (*i.e.*, smaller slices). In reality, at some point the overhead of increased communication overshadows the benefit of increased parallelism. We have developed a simple model to demonstrate this

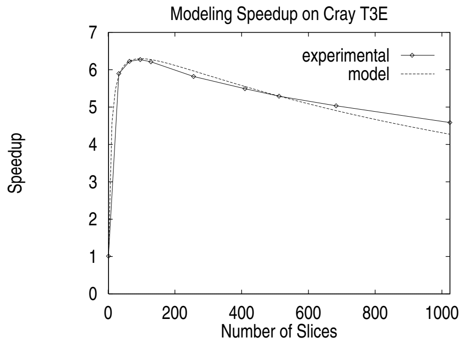

Figure 5: Modeled versus experimental speedup due to pipelining. The data comes from a single 2048×2048 wavefront computation in the Tomcatv benchmark on 8 nodes of the Cray T3E.

tradeoff. This model can be used to determine the optimal number of slices. The following equation models the speedup of pipelining versus not performing pipelining on a $n \times n$ array. Let c be the number of slices by which n is divided and p be the number of processors, arranged in a row or column.

$$\text{Speedup} = \frac{T^{\text{base}}_{\text{comp}} + T^{\text{base}}_{\text{comm}}}{T^{\text{pipe}}_{\text{comp}} + T^{\text{pipe}}_{\text{comm}}}$$

$$T^{\text{base}}_{\text{comp}} = n^2 \qquad\qquad T^{\text{base}}_{\text{comm}} = (\alpha + \beta n)(p-1)$$

$$T^{\text{pipe}}_{\text{comp}} = \frac{n^2}{p} + \frac{n^2}{cp}(p-1) \qquad\qquad T^{\text{pipe}}_{\text{comm}} = (\alpha + \beta\frac{n}{c})(c+p-2)$$

Without pipelining the compute time, $T^{\text{base}}_{\text{comp}}$, is simply a function of the total number of array elements. With pipelining, the compute time, $T^{\text{pipe}}_{\text{comp}}$, is the sum of the time to compute one processor's data (n^2/p) and a slice of the data. Without pipelining, communication is necessary $p-1$ times ($T^{\text{base}}_{\text{comm}}$). The cost of each communication has a constant startup cost, α, and a per transmitted data element cost, β. The pipelined communication cost ($T^{\text{pipe}}_{\text{comm}}$) is similar except that there are more messages, and each message sends fewer data elements. Note that the total number of messages for the pipelined case is $c+p-2$ rather than $c(p-1)$. This is because most of the communication proceeds in parallel. Figure 5 contains a graph that plots modeled and experimental speedup due to pipelining as a function of the number of slices, c, into which the arrays are divided. One slice implies that there is no pipelining, thus the speedup 1. The model closely tracks the experimental data. In future work, the compiler will use this model to find the optimal number of slices for a particular computation and machine.

4.2 Other Optimizations

There are a number of other possible optimizations that may improve performance of certain codes.

Array Contraction. The ZPL compiler fuses statements in an effort to enable the contraction of arrays to scalars [8]. Normally, the compiler does not fuse statements if the fusion introduces a loop carried flow dependence, for this would prohibit parallelism in one or more dimensions. Because scan blocks are inherently sequential in the wavefront dimension, we may extend our implementation of array contraction to allow contraction of arrays resulting in loop carried true data dependences along the wavefront dimension.

Associativity Detection. In the event that scan block computations are associative, more efficient parallel prefix implementations exist [7]. As an optimization, the compiler could recognize this and use the more efficient implementation.

Broadcast Detection. Similarly, scan blocks can be used to copy values across an array. This code is best implemented as a broadcast, rather than the more general serial scheme presented.

5 Performance Evaluation

In this section we demonstrate the potential performance benefits of providing scan blocks in an array language. Though it is possible that a compiler for a language without scan blocks could achieve this same level of performance, by providing explicit language support for wavefront computation, a programmer is ensured that the compiler is aware of the high-level structure of the computation and is thus likely to optimize it. This is an example of programming language design facilitating programmer/compiler collaboration. Alternatively, a compiler could recognize wavefront idioms in partially scalarized code. The problem with this approach is that it is difficult for programmers to know whether they have expressed their programs in terms that the compiler can optimize, and often the success of the compiler is sensitive to small changes in source program.

We conduct experiments on the Cray T3E and the SGI PowerChallenge using the Tomcatv and Simple [5] benchmarks. For each experiment, we consider each program as a whole, and we consider two components of each that contain a single wavefront computation. Our extensions drastically improve the performance of the two wavefront portions of code in each benchmark, which results in significant overall performance enhancement. First, we demonstrate the potential cache performance benefits of scan block versus partially scalarized array code even without the optimizations presented in Section 4. Next, we measure the additional improvement in parallelism of pipelining.

5.1 Cache Benefits

Consider the code fragment without scan blocks in Figure 2(a). A naive implementation of this code iterates over the rows of each array. If the arrays are allocated in column-major-order, performance will be limited by poor cache behavior. The scan block version of this code in Figure 2(b), however, does not specify an iteration order at the source level. This gives the compiler the freedom to choose the most appropriate way by which loops iterate

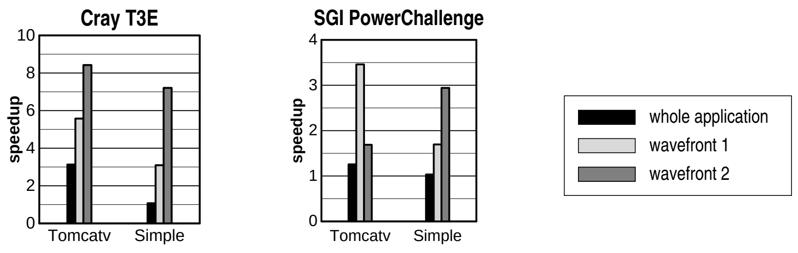

Figure 6: Potential speedup due to scan blocks from improved cache behavior.

over arrays for good cache performance. Though a compiler can optimize the code in Figure 2(a), the better code is much more likely to be generated when it is a by-product of the compilation process, rather than a specific optimization.

With this in mind, we experimentally compare the performance of partially scalarized and scan block exploiting implementations of identical computations. Figure 6 graphs the speedup of the former over the latter. Because these experiments are on a single node of each machine, the speedup is entirely due to caching effects. On the Cray T3E, the wavefront computations alone (the two grey bars) speedup by up to a factor of 8.5, resulting in an overall speedup (black bars) of a factor of 3 for Tomcatv and 7% for Simple. Tomcatv experiences such a large overall speedup, because the wavefront computations represent significant portions of the program's total execution time. The SGI PowerChallenge graph has a similar character except that the speedups are more modest (up to a factor of 4). This is because the PowerChallenge has a much slower processor than the T3E, thus the relative cost of a cache miss is less, so it is less sensitive to cache performance than the T3E.

5.2 Optimization Benefits

We are in the process of implementing the optimizations described in Section 4 in the ZPL compiler. In the meantime, we have performed the pipelining transformations by hand on a number of programs to assess its impact. Figure 7 presents speedup data due to pipelining alone. The basis for calculating the speedup is a fully parallel version of the code without the pipelining transformation. Thus the bars representing whole program speedup (black bars) are speedup beyond an already highly parallel code. The bars representing speedup of the wavefront computations (grey bars) are serial without pipelining, so the baseline in their case does not benefit from parallelism. We would like the grey bars to achieve speedup as close to the number of processors as possible. In all cases the speedup of the wavefront segments approaches the number of processors, and the overall program improvements are very large in several cases (up to a factor of 3). The smallest overall performance improvements are still greater than 5 to 8%. Though the absolute speedup improves as the number of processors increases, the efficiency decreases. This is because we have kept the problem size constant, so the relative cost of communication increases with the number of processors.

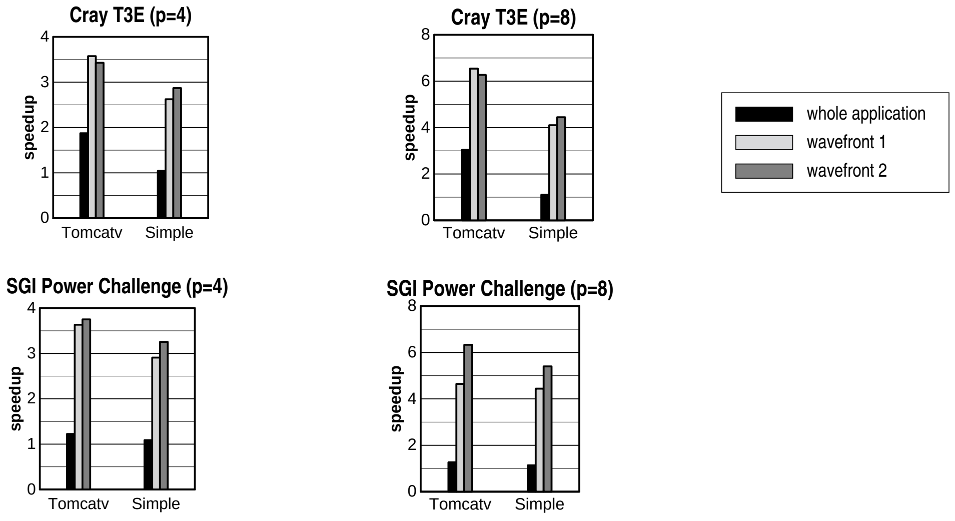

Figure 7: Speedup due to pipelining optimization.

6 Conclusion

We have extended array languages to support wavefront computations without scalarization, thus reclaiming the benefits of representing computations via high level array operations. In addition to its convenient and concise syntax, our extension enhances programmer/compiler collaboration. As a result, programmers' expectations of their codes' performance are likely to be met by the compiler, equipping programmers to make informed implementation decisions. We have described our approach to compiling the new language extension, and we have proposed several optimizations. In addition, we have shown that both full array and optimized implementations result in significant speedup versus naive scalarized and unoptimized implementations, respectively.

The experiments in this paper demonstrating pipelining benefit were optimized by hand. We will next fully mechanize this process, using the speedup model presented in Section 4 in order to automatically select the appropriate number of pipeline slices. We must decide how to parameterize the model given the characteristics of a particular machine and wavefront computation. Furthermore, we must address the issue of the appropriate slice size when the slice is perpendicular to the closely allocated array elements. In this case, the ideal slice size may need to be slightly increased or decreased so that the slice size is a multiple of the cache line size, resulting in improved cache performance.

Acknowledgments. We thank Sung-Eung Choi and Samuel Guyer for their comments on drafts of this paper. This research was supported by a grant of HPC time from the Arctic Region Supercomputing Center.

References

[1] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook*. McGraw-Hill, New York, NY, 1992.

[2] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL's WYSIWYG performance model. In *Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 50–61. IEEE Computer Society Press, March 1998.

[3] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Lawrence Snyder, W. Derrick Weathersby, and Calvin Lin. The case for high-level parallel programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–85, July–September 1998.

[4] Bradford L. Chamberlain, E Christopher Lewis, Calvin Lin, and Lawrence Snyder. Regions: An abstraction for expressing array computation. Technical Report UW-CSE-98-10-02, University of Washington, Department of Computer Science and Engineering, October 1998.

[5] W. Crowley, C. P. Hendrickson, and T. I. Luby. The SIMPLE code. Technical Report UCID-17715, Lawrence Livermore Laboratory, 1978.

[6] High Performance Fortran Forum. *High Performance Fortran Langauge Specification, Version 2.0*. January 1997.

[7] F. Thomas Leighton. *Introduction to Parallel Algorithms and Architectures*, chapter 1.2. Morgan Kaufmann Publishers, San Mateo, California, 1992.

[8] E Christopher Lewis, Calvin Lin, and Lawrence Snyder. The implementation and evaluation of fusion and contraction in array languages. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–59, June 1998.

[9] C. Lin, L. Snyder, R. Anderson, B. Chamberlain, S. Choi, G. Forman, E. Lewis, and W. D. Weathersby. ZPL vs. HPF: A comparison of performance and programming style. Technical Report 95–11–05, Department of Computer Science and Engineering, University of Washington, 1994.

[10] Naomi H. Naik, Vijay K. Naik, and Michel Nicoules. Parallelization of a class of implicit finite difference schemes in computational fluid dynamics. *International Journal of High Speed Computing*, 5(1):1–50, 1993.

[11] Ton A. Ngo. *The Role of Performance Models in Parallel Programming and Languages*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 1997.

[12] Lawrence Snyder. *The ZPL Programmer's Guide*. MIT Press (in press—available at ftp://ftp.cs.washington.edu/pub/orca/docs/zpl_guide.ps), 1998.

[13] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.

[14] ZPL Project. ZPL project homepage. http:/www.cs.washington.edu/research/zpl.