Explicit Data Placement (XDP): A Methodology for Explicit Compile-Time Representation and Optimization of Data Movement

Vasanth Bala Jeanne Ferrante Larry Carter

IBM T.J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598
{vas, ferrant, carterl}@watson.ibm.com

Abstract

The ability to represent, manipulate and optimize data movement between devices such as processors in a distributed memory machine, or between global memory and processors in a shared memory machine, is crucial in generating efficient code for such machines. In this paper we describe a methodology for representing and manipulating data movement explicitly in a compiler. Our methodology, called Explicit Data Placement (XDP), consists of extensions to the compiler's intermediate program language, as well as run-time structures that allow certain operations to be performed efficiently. We also illustrate one of the unique features of the XDP methodology: the ability to manipulate the run-time transfer of data ownership between processors.

1 Introduction

Many program representations used in compilers represent data movement and partitioning only partially, and in an implicit manner. Explicit Data Placement (XDP) is a methodology for the explicit representation and treatment of data movement and placement in a compiler. The key ideas behind the XDP methodology are:

1. Separation of data transfer from local computation, enabling the compiler to control their overlap.

2. Language- and machine-independent representation of data transfer operations, allowing their incorporation into existing optimizations such as code motion and redundant code elimination.

3. Unified and explicit treatment of data and ownership transfer enabling optimizations specific to these operations.

4. Generalized compute rules which allow the compiler freedom to go beyond the "owner-computes" rule.

5. Delayed binding of communication primitives to the transfer operations.

The XDP methodology can be incorporated into compilers that use a high-level compiler intermediate language in the SPMD (Single Program Multiple Data) execution model; the program will be loaded onto every processor of the target machine that is assigned to the program. While SPMD programs are commonly used in a distributed memory setting, the XDP methodology can also be used for compiling shared memory (sequential or parallel) programs to a distributed memory SPMD node program. The original shared memory program can be considered to be an SPMD node program that is replicated along with all its data, on every node. The compiler can then use data partitioning to transform the intermediate representation into the eventual distributed memory SPMD node program desired. At present, the XDP methodology does not apply to languages with pointer variables; the addition of pointers is the subject of future work.

The rules governing execution allow non-determinism and do not guarantee coherence or freedom from deadlock. The XDP approach is to expose the power of manipulating data ownership and transfer to a compiler allowing the loosest possible semantics for a variety of implementations. While XDP could be used as a programming language, it has been designed for use by the compiler, which can use XDP's unsafe operations with care.

Our thesis is that if a compiler is to optimize data movement, it needs a methodology with the key ideas 1-5 noted above. The XDP operations and structures provide a convenient platform for this optimization. In this paper, we give a syntax and an operational semantics for the XDP language constructs, outline an implementation of the data structures and routines to support the constructs at run-time, briefly discuss optimization and code generation, and give examples that illustrate one of XDP's unique features: the ability to specify dynamic transfer of data ownership.

2 XDP Language Constructs

The Explicit Data Placement (XDP) methodology can be used to extend an existing compiler Intermediate Language (IL) to obtain an SPMD program representation. Henceforth, we will use "IL+XDP" to denote a compiler intermediate language that has been extended with the XDP constructs and support structures. Before giving the formal definition of the XDP constructs, we first give some underlying assumptions and illustrate some of its features with an example.

2.1 Preliminaries

In this paper, we assume every variable is either a scalar or an array[1]. Each variable consists of *elements*; a scalar has only a single element.

XDP assumes the distribution of elements of all variables among processors: every element of a variable is either *exclusively owned* by a single processor or *universally owned* by all processors. It is possible to transfer the ownership of exclusively owned elements between processors. If an element is universally owned, each processor has a copy, and the values at each processor can be different.

A *section* of a variable is either a scalar variable or some subset of an array's elements. The form of possible sections is determined by IL; in this paper, we assume that sections are defined by Fortran 90 triplet notation. We say that a section of a variable is *exclusive* if every element of the section is exclusively owned; a section is *universal* if every element is universally owned. It is possible for one section of an array to be universal and another section of the same array to be exclusive.

We say that a section of a variable is *owned* by a processor if the processor exclusively or universally owns every element of the section. We distinguish between references to the *value* and the *name* of a section of a variable. A value cannot be used unless it is owned by the processor, names in XDP statements can be any section of any variable.

In XDP an exclusive section can be in one of three states with respect to a given processor p: unowned by p; accessible, meaning owned by p, and p has not initiated a receive that hasn't completed for that section; and transitional, meaning owned by p and p has initiated an uncompleted receive for that section. The value of a transitional section is unpredictable, and yet XDP does not automatically check the state of a variable at run-time (except by use of the accessible( ) and await( ) predicates described later). This allows optimizations to remove run-time checks when it can be determined they are unnecessary.

[1]Adding structures is an easy extension, pointers would be harder.

2.2 A Simple Example

Consider the program fragment:

```
do i = 1, n
  A[i] = A[i] + B[i]
enddo
```

It can be straightforwardly translated into the IL+XDP SPMD program:

```
do i = 1, n
  iown(B[i]) : { B[i] -> }
  iown(A[i]) : {
                 T[mypid] <- B[i]
                 await(T[mypid])
                 A[i] = A[i] + T[mypid]
               }
enddo
```

This translation follows the "owner-computes" rule. The variable mypid is an intrinsic which evaluates on each processor to a unique integer. Here, we assume that the elements of arrays A, B and T are all exclusively owned and processor mypid owns the mypid-th element of T. The variable i is universally owned, so each processor has its own copy of i.

In the example, each iteration of the loop is executed on every processor. On a given iteration of the loop, the execution of the first statement of the loop will be executed only by the exclusive owner of B[i]; this is insured by guarding the statement with the intrinsic predicate iown(B[i]). The use of iown is an example of a *compute rule*, which can be used to guard any XDP statement. Similary, only the exclusive owner of A[i] will execute the second statement on any iteration of the loop.

Following "iown(B[i]):" is a data transfer statement, where the exclusive owner of B[i] sends its value to another, unspecified processor. The notation "->" denotes the initiation of a data transfer operation in which the executing processor sends both the name and the value of a section of a variable to an unspecified processor. The statement "T[mypid] <- B[i]" is a data receive statement, where the executing processor receives the message with name B[i], putting the value into T[mypid]. It is the responsibility of the compiler to only generate programs in which all sends have matching receives. The await( ) intrinsic ensures the sum is not computed until the received value is available.

Optimization can be applied by the compiler to this straightforward translation, based on its knowledge of ownership. For instance, if the same processor that exclusively owns A[i] also owns B[i], then the data transfer statments can be eliminated. Even if they cannot be eliminated, the compiler may be able to move them out of the computation loop and combine or *vectorize* [8] the messages. In either case, if the loop bounds can be adjusted so that each reference to A[i] is local, then the ownership test on the remaining body of the loop can also be eliminated, yielding a much more efficient SPMD program.

An important feature of XDP is that other strategies than "owner-compute" can be expressed. For instance, the compiler might determine that it would save future communication if ownership of each element of the A array were moved to the same processor as the corresponding element of the B array. The following IL+XDP program fragment shows the result of this optimization:

```
do i = 1, n
  iown(A[i]) : { A[i] -=> }
  iown(B[i]) : { A[i] <=- }
  await(A[i]): { A[i] = A[i] + B[i] }
enddo
```

Here, the "-=>" and "<=-" notation indicates that both the ownership and value of A[i] is moved to the processor that owns B[i]. Only the processor that is the new owner of A[i] will perform the addition.

We next discuss the XDP language constructs and their semantics, which are also summarized in Figure 1.

2.3 Intrinsics

The first argument of an intrinsic is a *name* of an exclusive section, but it need not be owned by the executing processor. Thus, intrinsics can be evaluated on any processor.

XDP assumes each processor has a unique *processor id* denoted **mypid**.

The routine **mylb(X,d)** returns the smallest index in the dth dimension of any element of the exclusive section X owned by the invoking processor. If no element is owned, **MAXINT**, the largest representable integer, is returned. A similar routine myub(X,d) can be used to get the upper bound.

The iown( ) predicate returns true iff the processor executing it is the owner of all elements of the named section.

The accessible( ) predicate returns true iff the section is accessible on the calling processor. It can be used to allow a processor to perform a background computation while awaiting data from another processor.

The await( ) intrinsic returns false if the section is unowned, otherwise it blocks until the section becomes

NOTATION	
X	Any exclusive Section.
E	Exclusive section owned by p.
U	Exclusive section, all elements unowned by p.
INTRINSICS	
mypid	Returns the unique identifier of p.
mylb(X,d)	If any element of X is owned by p, returns the smallest index in dimension d, MAXINT otherwise.
myub(X,d)	If any element of X is owned by p, returns the largest index in dimension d, MININT otherwise.
iown(X)	Returns true if X is owned by p, false otherwise.
accessible(X)	Returns true if X is owned by p and its data is accessible, false otherwise.
await(X)	Returns false if X is unowned by p, otherwise blocks until X is accessible, then returns true.
SEND STATEMENTS	
E ->	Initiate send of the name and value of E.
E -> S	Initiate sends of the name and value of E to processors specified by set S.
E =>	Blocks until E is accessible, then initiate send of the ownership of E.
E -=>	Blocks until E is accessible, then initiate send of ownership and value of E.
RECEIVE STATEMENTS	
E <- X	Blocks until E is accessible, then initiate receive of the value named X into E.
U <=	Initiate receive of the ownership of U.
U <=-	Initiate receive of ownership and value of U.
STATES OF A SECTION	
accessible	Entire section is owned by p and p has no uncompleted receives involving any element of the section.
transitional	Entire section is owned by p and an uncompleted receive involving any element of the section has been initiated by p.
unowned	Some element of section is not owned by p. If a section is not unowned, we say it is owned.

Figure 1: Rules governing execution on processor p

accessible, at which time it returns `true`. Thus, `await` is a synchronization primitive.

All of the intrinsics can be implemented as a lookup into the processor's local run-time symbol table, discussed in section 3.1.

2.4 Compute Rules

A compute rule is any expression, including uses of intrinsics, that evaluates to `true` or `false`. However, compute rules may not have side effects, so in particular they may not include send or receive statements. Compute rules are used to govern execution of XDP statements. Only if the compute rule evaluates to `true` will the statement it guards be executed [3].

In a compute rule, any reference to a section (other than as the first argument of an intrinsic) which is not owned by the processor causes the entire compute rule to evaluate to `false`. Thus, a compute rule can always be executed on any processor without error.

Compute rules are syntactically distinct from the other IL+XDP statements so they can be treated separately, allowing the compiler to optimize them more easily. A typical optimization is compute rule elimination — the removal of a compute rule that always evaluates to true. Compute rule elimination can often be performed after the loop bounds are adjusted so that the computation within the loop only references owned sections [21, 4, 17].

XDP generalizes the notion of compute rule used in previous work by allowing general Boolean valued expressions to be used by the compiler.

2.5 Statements

Statements are executed only if the compute rule guarding them evaluates to `true`; in the absence of a compute rule, statements are executed by each processor that reaches the statement.

XDP augments IL with data and ownership transfer statements. These are either *send* or *receive* statements, and have an *initiation* and a subsequent *completion.*

XDP does not check whether a section used by a statement is transitional. Thus, the compiler must guard statements with appropriate synchronizing compute rules to ensure the program's correctness. This choice has been made to allow the compiler to remove run-time checks when it determines they are unnecessary.

We now discuss the send and receive statements in turn. Since these operations are distinct from the other operations in XDP+IL, they can be separately optimized.

2.6 Send Statements

Here, E always denotes an exclusively owned section of a variable by the executing processor.

Send statments come in several flavors. The statement "E ->" denotes the initiation of a data transfer operation in which the executing processor sends the name[2] and the value of E it exclusively owns to another unspecified processor. The restriction of data sends to exclusively owned sections of variables can always be overcome by copying the value of a universally owned section to an exclusive section. We impose the restriction here to simplify the semantics of our data transfer operations.

We also allow statements of the form E -> S, where S is some set of processor id's. This statement denotes the initiation of a set of data transfer operations in which the executing processor sends the value and name of E it exclusively owns to the specified locations. This statement can be used with S containing only one processor id as a way for the compiler to annotate which processor will be the recipient of the section. It can also be used for a broadcast or multicast operation.

A novel feature of XDP is its treatment of data ownership. Ownership in XDP is a transferable object, just as a data value can be transferred from one processor to another through communication. The statement "E -=>" denotes the initiation of an ownership send in which the executing processor relinquishes the exclusive ownership of E as well as its value to an unspecified processor. The statement "E =>" indicates the transfer of only the ownership of E, and not the value. The compiler may be able to determine that only the ownership, and not the value, needs to be transferred, and use the latter operation. Owner send operations block until the section is accessible.

There are various uses that can be made of XDP's ability to transfer ownership. First, when ownership of a section is transferred out of a processor, the storage it had occupied can be reused for a newly acquired section. This conserves address space and reduces paging. Second, it provides a wealth of possibilities for redistributing computation among the processors. Normally, one implements load balancing by migrating processes between processors. However, in XDP, load balancing can be implemented by migrating ownership of data while still running the same SPMD program on each processor. Since ownership dictates which SPMD program statements are executed by each processor, the ability to transfer data ownership allows the computation done on each processor to be altered dynamically

[2]The name is used as a tag to associate a send with a corresponding receive. It will be unnecessary to actually send the name if either the association between sender and receiver can be made at compile time, or if the hardware can make the association as on a shared address space machine.

without migrating any code. Thirdly, it opens up the possibility of new uses. For instance, a debugger could allow the user to input an ownership transfer command that moves exclusive ownership of a variable (and hence the permission to execute certain SPMD code segments, such as a print command that outputs the value of local data structures to the user's screen) from one processor to another. Thus, processors can be selectively monitored by simply transferring ownership of this variable.

2.7 Receive Statements

Here, X always denotes an exclusive section (but not necessarily one owned by the executing processor p), E always denotes a section exclusively owned by p, and U denotes an exclusive section, no element of which is owned by p.

The statement "E <- X" denotes the initiation of a data receive operation, in which the executing processor assigns to the variable E the received value of X. If the section is transitional, the statement blocks until it becomes accessible. "U <=-" denotes the initiation of an ownership and value receive from an unspecified processor, in which the executing processor accepts the exclusive ownership and value of U. The statement "U <=" indicates the initiation of only the transfer of ownership of U, and not the value of U. Ownership of a section can only be received if the section was unowned.

Upon initiation of a receive of a section on a processor, the section must be put in state `transitional`; upon completion of the receive, the section is returned to state `accessible`. This can be implemented by an update to the processor's run-time symbol table, as discussed in section 3.1.

It is incorrect usage of XDP if the sections transferred in send and receive operations do not match. The results of such a communication are unpredictable. XDP restricts the left hand side of a receive statement to an exclusive section so that the run-time symbol table need not contain entries for universally owned variables.

It is legal to have several processors initiate receive statements for the same section concurrently. For simplicity, a particular compiler may choose not to use this construct. However, it can be used to advantage, for instance to facilitate load balancing. This could be accomplished by having the owner of a particular variable initiate a sequence of sends of values of the variable, each value representing a certain job to be performed. Meanwhile, any processor that was otherwise idle could initate a receive of that variable, and then perform the indicated job. Depending on the load at run-time, there might be multiple outstanding sends or outstanding receives.

3 Implementation

While XDP language constructs are designed to be used by a compiler, it is entirely possible that the compiler will not be able to remove *all* ownership or accessibility tests, and so `iown( )`, `await( )` and `accessible( )` predicates may need to be evaluated at run-time [17]. In addition, ownership transfers result in run-time changes in ownership and so may need to be tracked at run-time. To support this, the XDP methodology supplies both a compile-time symbol table, and a run-time, per-processor symbol table for exclusive sections, discussed in detail in the next section.

The XDP language constructs allow ownership transfers to occur at the granularity of a single element. However, for efficiency's sake, a compiler may use a *coarser granularity* of ownership transfer. The use of segments in the implementation given here is an example.

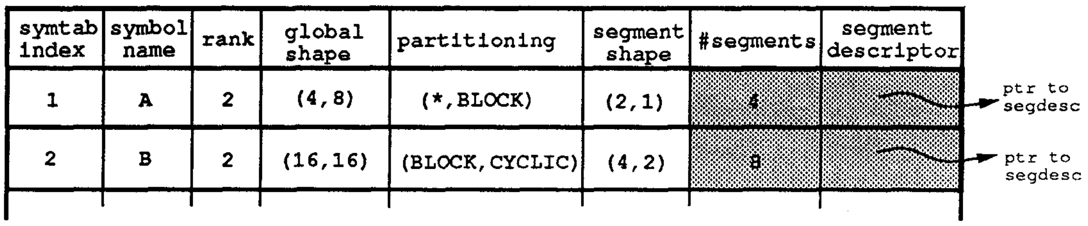
Whether the symbol table is simple or complex depends on such choices as whether the number of processors is fixed and known at compile-time, and what partitioning of arrays into sections are allowed. These choices also affect what optimizations can be performed. In our example implementation, we assume a fixed, known processor grid and and partitioning as allowed in HPF [5].

3.1 Symbol table

An important structure required for incorporating the XDP methodology is the symbol table. The XDP symbol table structure is used at compile-time by the compiler, as well as at run-time by all the processors that execute the output SPMD code. Each processor must maintain and update its own local copy of the XDP symbol table structure at run-time, unless all uses of the table have been optimized away. In contrast to a regular symbol table, the run-time XDP symbol table only contains information about exclusive sections.

Figure 2 illustrates the XDP symbol table structure for two array variables `A[1:4,1:8]` and `B[1:16,1:16]`, partitioned over 4 processors, which are assumed to be indexed as a 2x2 processor grid. The `symtab index`, `symbol name`, `rank`, and `global` shape fields are self-explanatory. The `partitioning` field indicates the partitioning scheme of the array. The partitioning scheme, together with the shape of the processor grid, are used by the compiler and the XDP run-time system to determine ownership of array sections.

For efficiency's sake, the compiler can logically divide each processor's local partition of an array into *segments* of a size and shape chosen by the compiler. A processor can transfer the ownership of each segment individually. The last three fields of the symbol table describe the partitioning. They specify how many segments comprise the processor's partition, the shape of each seg-



symtab index	symbol name	rank	global shape	partitioning	segment shape	#segments	segment descriptor
1	A	2	(4,8)	(*,BLOCK)	(2,1)	4	ptr to segdesc
2	B	2	(16,16)	(BLOCK,CYCLIC)	(4,2)	8	ptr to segdesc

Figure 2: XDP symbol table structure, showing the entries for two arrays A and B, partitioned over a 2x2 processor grid. The shaded entries are filled in at run-time by each processor.

ment (which must have the same rank as the array variable), and finally an array of segment descriptors, which record, for each segment, the array elements contained in the segment and the current state (unowned, transitional, or accessible). In our implementation, the segment descriptor data structure was declared as:

```
struct SegmentDesc {
  int  status;      /* accessibility status   */
  int  lbound[rank]; /* lower bound indices */
  int  ubound[rank]; /* upper bound indices */
  int  stride[rank]; /* strides             */
  ...               /* other relevant info   */
  long segptr;      /* pointer to segment    */
} segdesc [#segments];
```

The last two fields of the XDP symbol table are shaded dark in Figure 2, to indicate that these entries are filled in only at run-time. When ownership is transferred or receives are initiated or completed, the symbol table must be updated.

Either at the start of program execution or dynamically, each processor allocates local storage for its segments in contiguous chunks whose sizes are determined by the segment shape field. The number of such segments allocated depends on the number of array elements the processor owns. Figure 3 illustrates two different partitioning schemes for a 4x8 array, and for each partitioning scheme, two possible logical segmentations are shown.

The use of segments allows the pipelining of a transfer of a section, either ownership or data. A processor can transfer each segment individually, requiring only enough synchronization to ensure that the transfer is legal in XDP. In many cases, this can effectively reduce the total time by allowing a processor to overlap one segment's transfer with computation on another segment. This will be illustrated the 3-D FFT example.

If the code running on a processor executes an iown( ) intrinsic at run-time, the section described in the query is intersected with all the segment bounds corresponding to the named array variable. If the union of all the results is equal to the queried section, and no segment that has a non-null intersection is unowned, then the iown( ) query returns true. Otherwise it returns false. For example, consider an array C[1:4,1:8], distributed as (BLOCK,BLOCK) over a 2x2 processor grid, and 2x1 segmented (as shown in Figure 3 (a)). Suppose processor P3 executes the operation iown(C[1,5:7]). Intersecting the bounds of the section (1,5:7) with the bounds of the four 1x2 segments owned by P3 yields: {(1,5), (1,6), (1,7), null}. The union of these is (1,5:7), which is equal to the section specified in the iown( ) query. Now, if none of the non-null intersecting segments are unowned, the operation returns true, and it returns false otherwise. The other intrinsics are handled similarly. Although the algorithm we described for evaluating iown( ) involves examining the entire segment descriptor array, more efficient algorithms could be developed.

When any receive is initiated or completed on a segment, the status field of the segment needs to be updated as well. When any ownership transfer is initiated, the processor must update the segment descriptor fields of its symbol tables to reflect the data that is currently owned. The partitioning field may need to be updated as well.

We have chosen *not* to supply in the XDP methodology a mechanism for testing which processor owns an arbitrary section at run-time. A compiler using the XDP methodology could itself provide such a mechanism. If such information is unavailable at compile-time and needs to be repeatedly computed at run-time, the techniques such as [20] can be used to improve efficiency. Note, however, that it may be unsafe to compute owner information on an array that is undergoing incremental ownership transfer, until the transfer of all segments has been finished.

3.2 Optimization and Code Generation

Compiler optimizations that affect data movement and storage issues can be represented as transformations to the IL+XDP code. After the optimization phase is complete, the IL+XDP program is translated to executable code by the compiler's back end.

The translation needs to map XDP constructs to operations provided by the target computer's hardware and operating system. For instance, on a shared-address

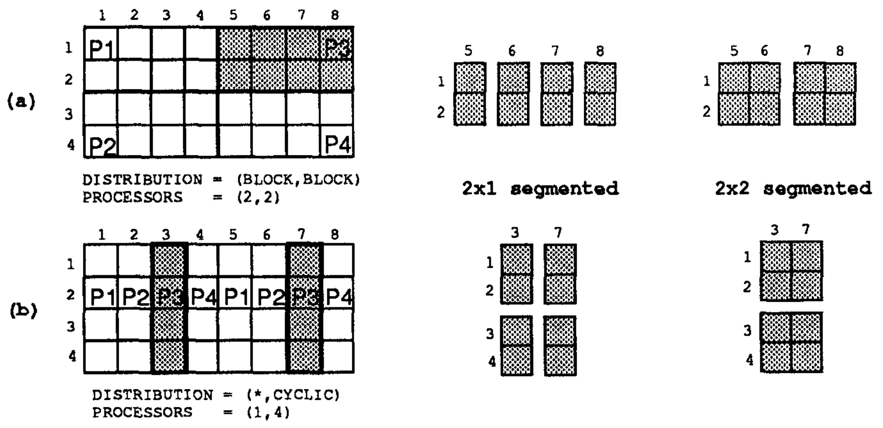


Figure 3: Example distributions and local segmentations of a 4x8 array, shown for processor P3.

computer such as the KSR1 [16], receives and sends might be translated as prefetch and poststore instructions. On a message-passing machine, they would become calls to the communication primitives. The XDP data and ownership transfer operations allow the compiler to delay until code generation time, the actual binding of communication primitives to the data transfer operations. The advantage of this delayed binding for optimization is noted in [7].

Some optimizations, such as compute rule elimination, are independent of the target computer. Other optimizations need to be modified depending on various architectural and system considerations. For instance, if the communication primitives generated by the compiler are non-blocking, then it is generally desirable to move the XDP receive statements as early in the program as possible (consistent with the data dependence constraints) to give the maximum opportunity of overlapping communication with computation. However, if the communication primitives are blocking, then the optimizations must be careful not to introduce deadlock.

To perform optimizing transformations and code generation, the compiler may need information about the IL+XDP program in addition to the usual data structures such as a control flow graph and data flow information. For instance, it may be useful for optimizations (and essential for code generation) to annotate an XDP send statement with the id of the receiving processor. Other aspects of XDP code can be handled by traditional techniques. For instance, if no use-def chains from a use of `X` in an `accessible(X)` intrinsic lead back to a receive statement, then it may be possible to eliminate the `accessible(X)` call.

The set of optimizations on XDP code, as well as details of XDP itself, are the subject of current research. For instance, aggregating a set of separate data transfers into a single message can reduce overhead on some systems. It might be desirable to allow this aggregation to be expressed in XDP, for instance by allowing the left-hand side of XDP send and receive statements to be a set of sections, rather than a single section.

4 An Example: 3-D FFT

We now illustrate a use of XDP, using a 3-dimensional Fast Fourier Transform (3-D FFT) application as an example. The 3-D FFT code considered here operates on an array `A[1:4,1:4,1:4]` which is assumed to be initially distributed as `(*,*,BLOCK)` over a linear array of 4 processors `P1-P4`. Thus, processor `i` owns the section `A[1:4,1:4,i]`. We assume that the compiler has chosen to divide each processor's local storage into segments containing 4 consecutive array elements each. The 3-D algorithm employs a 1-D FFT routine, `fft1D( )`, that is successively applied along each line of the second dimension of the array, then the first and finally the third dimensions to compute the 3-D FFT. The initial `(*,*,BLOCK)` distribution of the array allows the first two dimensions to be handled with no interprocessor communication. The array is then redistributed to a `(*,BLOCK,*)` scheme in order that the 1-D FFT along the third dimension can be done independently on each processor without communication. The two partitioning schemes, and the actual data layout in each processor's local storage is shown in Figure 4.

The following programs illustrate the steps involved in the optimization of the redistribution operation. In order to keep the illustration compact, we start with the IL+XDP code after some compiler optimizations have been finished. These initial optimizations include the insertion of `iown( )` guards based on the data distribution of the array `A`, and generating the appropriate XDP ownership transfer operations to do the redistribute operation to change the partitioning scheme of `A` from `(*,*,BLOCK)` to `(*,BLOCK,*)`.

```
// A is distributed as (*,*,BLOCK)
// Loop1: 1-D FFT in the j direction
  do k = 1, 4
    iown(A[*,*,k]): {
      do i = 1, 4
        fft1D (A[i,*,k])
      enddo
    }
  enddo
// Loop2: 1-D FFT in the i direction
  do k = 1, 4
    iown(A[*,*,k]): {
      do j = 1, 4
        fft1D (A[*,j,k])
      enddo
    }
  enddo
// Loop3: Redistribute A as (*,BLOCK,*)
    do p = 1,4
      iown(A[*,*,p]): {
        do n = 1,4
          A[*,n,p] -=>
        enddo
        do n = 1,4
          A[*,p,n] <=-
        enddo
      }
    enddo
// Loop4: 1-D FFT in the k direction
  do j = 1, 4
    await(A[*,j,*]): {
      do i = 1, 4
          fft1D (A[i,j,*])
      enddo
    }
  enddo
```

Loop 3 in the above code is one possible way of performing the desired array redistribution using the XDP ownership transfer operations. Although not shown here, an auxillary data structure is created by the compiler that links the -=> and <=- statements. This is used for communication binding at code generation time and to generate matching message types for these communications. Another data structure is used to bind the local segment that will hold the received ownership (and associated values) upon termination of an ownership transfer.

A typical optimization step is compute rule elimination. This is achieved by adjusting the outer loop bounds so that each processor only does those iterations for which it owns the data. In our example, the result is each processor has to execute only one outer loop iteration for each of the loops shown above. By replacing all references to the loop's induction variable in the body of the loop by mypid, these single iteration outer loops can also be removed as a further optimization. The resulting code is:

```
// A is distributed as (*,*,BLOCK)
// Loop1: 1-D FFT in the j direction
    do i = 1, 4
      fft1D (A[i,*,mypid])
    enddo
// Loop2: 1-D FFT in the i direction
    do j = 1, 4
      fft1D (A[*,j,mypid])
    enddo
// Loop3a,3b: Redistribute A as (*,BLOCK,*)
    do n = 1,4
      A[*,n,mypid] -=>
    enddo
    do n = 1,4
      A[*,mypid,n] <=-
    enddo
// Loop4: 1-D FFT in the k direction
    await(A[*,mypid,*]): {
      do i = 1, 4
        fft1D (A[i,mypid,*])
      enddo
    }
```

Dependence analysis of Loops 2 and 3a indicates that they can be fused together. Note that the analysis for validity of fusion must also check to make sure that between any -=> and its corresponding <=- operation, no ownership queries are performed on the associated data, and that these data are not accessed by computation in the interim. The potential benefit of the loop fusion is that it allows the ownership transfer to be "pipelined" so that the redistribute latency can be partially covered by the computation.

A second transformation is also illustrated: moving the await statement *into* Loop 4. Although this might incur a greater run-time overhead, it can allow the FFT operations to proceed while other data is still being transferred. The resulting program is:

```
// A is distributed as (*,*,BLOCK)
// 1-D FFT in the j direction
    do i = 1, 4
      fft1D (A[i,*,mypid])
    enddo
// 1-D FFT in the i direction
    do j = 1, 4
      fft1D (A[*,j,mypid])
      A[*,j,mypid] -=>
    enddo
// Loop3b
    do n = 1,4
      A[*,mypid,n] <=-
    enddo
// 1-D FFT in the k direction
    do i = 1, 4
      await(A[i,mypid,*]): {
        fft1D (A[i,mypid,*])
      }
    enddo
```

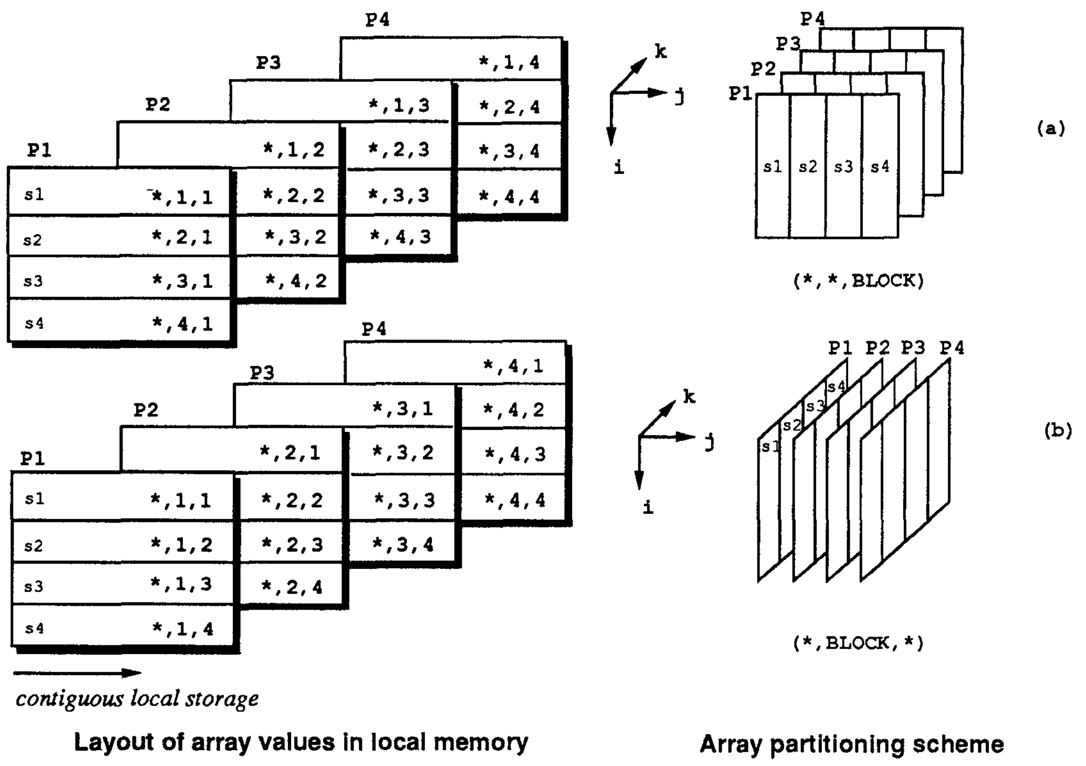


Figure 4: 3-D FFT example. The column on the left shows how the array data are assigned to logical segments on each processor. The column on the right illustrates the conceptual partitioning scheme of the 3-D array geometry.

The actual performance improvements of these optimizations depend largely on the capabilities of the runtime communication library of the target machine.

5 Related Work

Traditional optimizing compilers [1] use relatively language- and machine-independent intermediate program representations, but do not represent data movement and placement in an explicit manner, as done in XDP. Compilers being developed for distributed-memory multiprocessors [2, 4, 6, 8, 9, 19, 12, 13, 14, 15, 17, 18, 21] represent data movement in terms of communication primitives available on the target machine. Lake [11] has cited the importance of annotating programs with data placement, and suggested its insertion into imperative languages. Ownership transfer at the operating system level is considered by systems such as [10].

6 Conclusion

The XDP methodology has been designed to expose the power of manipulating data transfer and ownership to the compiler. We have given rules governing the use of its constructs; the compiler must supply adequate synchronization to satisfy these rules. Coherence and freedom from deadlock must also be ensured by the compiler.

The key ideas behind the XDP methodology are its separation of data transfer from local computation, its non-blocking semantics to allow overlapping of communication with computation, and its unified treatment of data and ownership transfer. In addition, XDP offers the compiler a convenient platform for doing optimizations involving data movement by providing mechanisms for delayed communication binding and generating generalized compute rules. The run-time symbol table given here to support XDP is implementable as an extension to most high-level compiler intermediate languages. The applicability of XDP is quite general, and is not restricted to the optimization of communication for distributed memory machines. For instance, it can be used to optimize data transfers across different levels of a memory hierarchy.

Acknowledgments

We would like to thank Manish Gupta for participating in several discussions with us and also helping us with a preliminary implementation. We are also grateful to Fran Allen, Guang Gao, Dave Gelernter, Francois Irigoin, Kathy Knobe, Sam Midkiff, Anne Rogers, Randy Scarborough, Edith Schonberg, Harini Srinivasan, Guy Steele and the PPoPP program committee for their input.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] S. Ahuja, N.J. Carriero, and D.H. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[3] V. Balasundaram. Translating control parallelism to data parallelism. *Fifth SIAM Conference on Parallel Processing for Scientific Computing*, Houston, TX, March 1991.

[4] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[5] HPF Forum. High Performance Fortran language specification. *Version 1, available from Rice University, Houston TX*, January 1993.

[6] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. *Proceedings of the Sixth Distributed Memory Computer Conference (DMCC6), Portland, Oregon*, April 1991.

[7] Mary Hall, Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. *Proceedings of Supercomputing '92*, pages 522–534, November 1992.

[8] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[9] K. Ikudome, G. Fox, A. Kolawa, and J.W. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. *Proceedings of the Fifth Distributed Memory Computing Conference, Charleston, S. Carolina*, April 1990.

[10] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[11] T. Lake. Distributing computations. In R.H. Perrott, editor, *Software for Parallel Computers.* Chapman and Hall, London, 1992.

[12] J. Li and M. Chen. Generating explicit communication from shared memory program references. *Supercomputing 90, New York*, pages 865–877, Nov 1990.

[13] P. Mehrotra and J. Van Rosendale. The BLAZE language: a parallel language for scientific programming. *Parallel Computing*, pages 339–361, 1987.

[14] P. Mehrotra and J. Van Rosendale. Compiling high level constructs to distributed memory architectures. *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, March 1989.

[15] A. Reeves. The Paragon programming paradigm and distributed memory compilers. Technical Report EE-CEG-90-7, Cornell University, Computer Engineering Group, June 1990.

[16] Kendall Square Research. Technical summary. Technical report, Kendall Square Research, 1992.

[17] A. Rogers and K. Pingali. Process decomposition through locality of reference. *ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, pages 69–80, June 1989.

[18] M. Rosing, R. Schnabel, and R. Weaver. The DINO parallel programming language. *Journal of Parallel and Distributed Computing*, 13(9):30–42, September 1991.

[19] R. Rühl and M. Annaratone. Parallelization of Fortran code on distributed memory parallel processors. *Proceedings of the ACM International Conference on Supercomputing*, 1990.

[20] J. Wu, J. H. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. *Proceedings of the 1991 International Conference on Parallel Processing, St. Charles, IL*, August 1991.

[21] H.P. Zima, H.J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic SIMD/MIMD parallelization. *Parallel Computing*, 6:1–18, 1988.