International Journal of Parallel Programming, Vol. 22, No. 5, 1994

XDP: A Compiler Intermediate Language Extension for the Representation and Optimization of Data Movement

Larry Carter,[1] Jeanne Ferrante,[1] and Vasanth Bala[2]

Received August 18, 1993

The ability to represent, manipulate, and optimize data placement and movement between processors in a distributed address space machine is crucial in allowing compilers to generate efficient code. Data placement is embodied in the concept of data *ownership*. Data movement can include not just the transfer of data values but the transfer of ownership as well. However, most existing compilers for distributed address space machines either represent these notions in a language- or machine-dependent manner, or represent data or ownership transfer implicitly. In this paper we describe XDP, a set of intermediate language extensions for representing and manipulating data and ownership transfers explicitly in a compiler. XDP is supported by a set of per-processor structures that can be used to implement ownership testing and manipulation at run-time. XDP provides a uniform framework for translating and optimizing sequential, data parallel, and message-passing programs to a distributed address space machine. We describe analysis and optimization techniques for this explicit representation. Finally, we compare the intermediate languages of some current distributed address space compilers with XDP.

KEY WORDS: Optimizing compiler; intermediate language; data ownership; data placement; parallel computers.

[1] IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598. E-mail: {carterl,ferrant}@watson.ibm.com. Current address: University of San Diego, 9500 Gilman Drive, La Jolla, California 92093-0114.

[2] Kendall Square Research, 170 Tracer Lane, Waltham, Massachusetts 02154. E-mail: vas@ksr.com.

0885-7458/94/1000-0485$07.00/0

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. Explicit data and ownership transfer operations using nonblocking semantics. A unified framework facilitates optimization of these operations; in particular, the compiler can control the overlap of communication with computation.
2. Language- and machine-independent representation of data transfer operations, allowing their incorporation into existing optimizations.
3. Generalized compute rules which allow the compiler freedom to go beyond the "owner-computes" rule.
4. A run-time system to support the XDP primitives. This allows unoptimized code to be executed, while optimizations can eliminate the use of calls to run-time XDP primitives or reduce their overhead.
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 same program will be loaded onto every processor of the target machine that is assigned to the program. One common way of writing SPMD programs is with barrier synchronization; such synchronization can be translated in a straightforward manner to XDP primitives. However, optimizations on the resulting intermediate code might change the synchronization, yielding a less structured (but hopefully more efficient) program.

While SPMD programs are commonly used in a distributed address space setting, the XDP methodology can also be used for compiling a shared address space program (either sequential or data parallel) to a distributed address space SPMD node program. The original shared address space 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 this into the eventual distributed address space SPMD node program desired. At present, the XDP methodology does not apply to languages with pointer variables.

The rules governing execution of XDP programs allow nondeterminism and do not guarantee coherence or freedom from deadlock. 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. Although not discussed here, a deadlock detection mechanism in the XDP run-time environment might be desirable.

Our thesis is that if a compiler is to optimize data movement, it needs a methodology with the key ideas 1–5 noted earlier. The XDP operations and structures provide a convenient platform for this optimization. In subsequent sections, 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, give an example of initial translation and optimization, discuss analysis and optimization, and compare XDP with the intermediate languages of other distributed address space compilers.

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 then 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. (Adding structures is an easy extension, pointers would be harder.) Each variable consists of *elements*; a scalar has only a single element.

XDP assumes the elements of all variables are distributed among processors: every element of a variable is either *exclusively owned* by a single processor or *universally owned* by all processors. (Techniques such as in Refs. 1–3 address the question of how the data should be distributed initially and redistributed during the program's execution.) 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 p if p exclusively or universally owns every element of the section. A section is *unowned* by p if it includes some element that is not owned by p. We distinguish between references to the *value* and to the *name* of a section of a variable. A value cannot be referenced unless it is owned by the processor; names in XDP statements can be any section of any variable.

XDP language constructs include several forms of send and receive operations. The communication constructs all have an initiation and a completion to be described later. Initiation and completion are kept track of by a run-time mechanism that records a *state* for each exclusive section. Sends and receives, both of values and ownership, are nonblocking operations, so communication can take place concurrently with subsequent operations. Thus, whenever a transfer occurs, the variable's state must be tested to ensure the transfer has completed. (One exception to this rule is that a value may be sent and then subsequently used without a state check.)

In XDP an exclusive section can be in one of four states with respect to a given processor p: *unowned* by p; *R/W accessible*, meaning owned by

STATES OF A SECTION

Unowned (U)	Some element of section is not owned by p.
Transitional (T)	Entire section is owned by p and an uncompleted receive involving some element of the section has been initiated by p.
Read accessible (R)	Entire section is owned by p and an uncompleted send involving some element of the section has been initiated by p.
R/W accessible (R/W)	Entire section is owned by p and p has no uncompleted receives or sends involving any element of the section.

Fig. 1. States of a section.

p, and *p* has no uncompleted receives or sends involving any element of the section; and two additional states for owned sections that have uncompleted communication. The section is *transitional* if *p* has initiated a receive for that section that has not yet completed; it is *read accessible* if *p* has initiated an uncompleted send. States of a section are summarized in Fig. 1.

2.2. A Simple Example

Consider the program fragment:

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

Assuming that the elements of arrays A and B are exclusively owned, the fragment is straightforwardly translated into the IL + XDP SPMD program:

```
do i = 1, n
  iown(B[i]) : { B[i] -> }
  iown(A[i]) : {
                 T[mypid] <- B[i]
                 readawait(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. XDP requires that values be received into exclusively owned sections, so the array `T` is used by each processor to receive an element of `B`. We assume 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 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. Similarly, only the 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 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 generate only programs in which all sends have matching receives. The readawait( ) 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 statements can be eliminated. Even if they cannot be eliminated, the compiler may be able to move them out of the computation loop and *vectorize*[4] the messages, that is, combine many small messages into one large message. 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 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] <=- }
  writeawait(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 Fig. 2.

NOTATION	
X	Any exclusive Section.
E	Exclusive section owned by p.
U	Exclusive section, no element owned 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 dth dimension, `MAXINT` otherwise.
`myub(X,d)`	If any element of X is owned by p, returns the largest index in dth dimension, `MININT` otherwise.
`iown(X)`	Returns `true` if X is owned by p, `false` otherwise.
`readable(X)`	Returns `true` if X is owned by p and its data is read or R/W accessible, `false` otherwise.
`writeable(X)`	Returns `true` if X is owned by p and its data is R/W accessible, `false` otherwise.
`readawait(X)`	Returns `false` if X is unowned by p, otherwise waits until X is read or R/W accessible, then returns `true`.
`writeawait(X)`	Returns `false` if X is unowned by p, otherwise waits until X is R/W accessible, then returns `true`.
SEND STATEMENTS	
`E ->`	Waits until E is read or R/W accessible, then initiates send of the name and value of E.
`E -> S`	Waits until E is read or R/W accessible, then initiates sends of the name and value of E to processors specified by set S.
`E =>`	Waits until E is read or R/W accessible, then initiates send of the ownership of E.
`E -=>`	Waits until E is read or R/W accessible, then initiate send of ownership and value of E.
RECEIVE STATEMENTS	
`E <- X`	Waits until E is R/W accessible, then initiates receive of the value named X into E.
`U <=`	Initiates receive of the ownership of U.
`U <=-`	Initiates receive of ownership and value of U.

Fig. 2. Rules governing execution on processor *p*.

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 `d`th 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. (More elaborate intrinsics would undoubtedly be helpful, particularly if IL allows complicated array distributions such as cyclic or block-cyclic.)

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

The `writable( ) (readable( ))` predicate returns `true iff` the section is R/W (either read or R/W) 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 `writeawait( ) (readawait( ))` intrinsic returns `false` if the section is unowned, otherwise it waits until the section becomes R/W (either read or R/W) accessible, at which time it returns `true`. Thus, these `await` intrinsics are for synchronization.

All of the intrinsics can be implemented as a lookup into the processor's local run-time symbol table, discussed in Section 3.1. In addition, the synchronization intrinsics may require waiting for a state to change.

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 other than waiting for a state to change. 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.[5]

Compute rules can be handled exactly like `if` statements. In XDP, compute rules are shown syntactically distinct from the other IL + XDP statements, and side effects are prohibited, 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 is not required for synchronization and 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.[6–8]

XDP generalizes the notion of compute rule used in previous work[7,8] by allowing general Boolean valued expressions to be used by the compiler.

2.5. Statements

Statements are executed only if the compute rules guarding them evaluate 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*. One fundamental idea behind the XDP style of communication is that communication statements are nonblocking in the sense that once initiated, execution can proceed without waiting for the completion of the communication. A second key idea is that argument section(s) of a communication statement can be used as buffers, instead of requiring separate buffer storage. However, in order to preserve the correct order of receive statements, prior to initiating any communication, any outstanding receive must be completed. Prior to initiating any receives, all outstanding sends must have completed. Multiple sends of the value of a section can be initiated on a processor, since the section's value will not be changed. However, there can be only one outstanding receive of the same section initiated on any processor. When an outstanding send or receive completes, any synchronization intrinsic that was in the wait state can proceed. The initiation and completion of XDP communication statements are kept track of using the states of the argument section, further described in Section 2.6.

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.

2.5.1. Send Statements

Here, E always denotes a section that is exclusively owned by the executing processor.

Send statements come in several flavors. The statement "`E->`" denotes the initiation of a data transfer operation in which the executing processor sends *the name* and the value of `E` it exclusively owns to another unspecified processor. (*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.) 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 to simplify semantics; specifically, universal variables do not require state-checking.

XDP also has 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 processors. This statement can be used with S containing only one processor id as a way for the compiler to indicate 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. (Any ownership transfer always includes the boundaries of the section.) 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. In this case, the value is lost.

Before it initiates, any data or ownership send statement waits for all previously initiated receives (but not sends) of the section on the executing processor to complete, at which time the section will be in state read accessible or R/W accessible. Upon initiation of a data send, the section is put in state read accessible; upon initiation of an ownership send the section is put in state unowned. Sends are nonblocking in the sense the current send does not have to complete in order for execution to continue. Upon completion of the data send, the section is put in state R/W accessible. Figure 3 summarizes the state transition rules for XDP statements.

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. (In the case of a "$-=>$" operation, the storage is not reclaimed until the data transfer is completed.) 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

STATEMENTS	STATE OF X U	T	R	R/W
X ->	Error	Wait	R	R
X =>	Error	Wait	U	U
X -=>	Error	Wait	U	U
X <- Y	Error	Wait	Wait	T
X <=	T	Error	Error	Error
X <=-	T	Error	Error	Error
= X	Error	Error	R	R/W
X =	Error	Error	Error	R/W
INTRINSICS				
iown	false	true	true	true
readable	false	false	true	true
writeable	false	false	false	true
readawait	false	Wait	true	true
writeawait	false	Wait	Wait	true

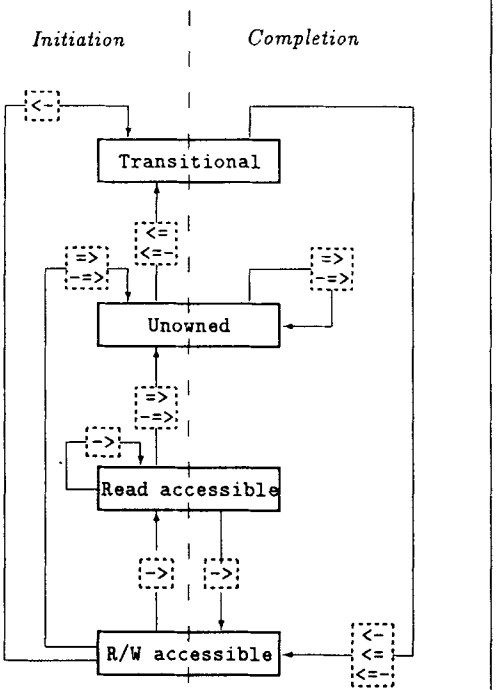

Fig. 3. State transition rules.

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 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.

2.5.2. 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. 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. "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 receipt of ownership of U, and not the value of U.

Any statement of the form "E <– X" waits for all outstanding sends and receives of E to complete before it initiates; upon initiation, the state of E becomes R/W accessible. For an ownership receive, there shouldn't be any outstanding sends or receives of the section, since as previously stated, the section cannot be received unless all elements are previously unowned. Thus at initiation of an ownership receive, it is an error if the section is not in state unowned. Upon initiation of a receive, the section is put in state transitional. Receives are also nonblocking. Upon completion of any receive, the section is put in state R/W accessible.

It is legal to have several procesors initiate receive statements for the same section concurrently. To simplify the run-time procedures needed to support XDP communication, 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 initiate 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.

2.6. States of a Section

XDP enforces the restrictions mentioned above by assigning a *state* to each section.

Figure 4 shows the effects of the XDP statements on the states of the referenced section. The table shows which states are legal for the different language constructs. Execution of a transfer statement is the *initiation* of the communication; the table shows which state is entered upon initiation. *Completion* of communication occurs asynchronously sometime later. The state transition indicates which state is entered upon both initiation and completion of each legal statement.

The table entries labeled "wait" mean that the statement is not initiated until the section enters the R/W accessible state, and then begins execution. For instance, if a "X < – Y" statement is encountered while X is read accessible, the statement waits until the outstanding send is completed, then the receive is initiated (which leaves X in the transitional state.)

	STATEMENTS
E ->	Use of E, Use and Def of E.state
E -> S	Use of E, Use and Def of E.state
E =>	Use of E, Use and Def of E.state
E <- X	Def of E, Use and Def of E.state
U <=	Use and Def of U.state
E =	Def of E, Use of E.state
= E	Use of E, Use of E.state
	INTRINSICS
mypid	Use of mypid
mylb(X,d)	Use of X.state and d
myub(X,d)	Use of X.state and d
iown(X)	Use of X.state
readable(X)	Use and Def of X.state
writable(X)	Use and Def of X.state
readawait(X)	Def of X.state, Use of all .state's
writeawait(X)	Def of X.state, Use of all .state's

Fig. 4. Data flow effects of XDP constructs.

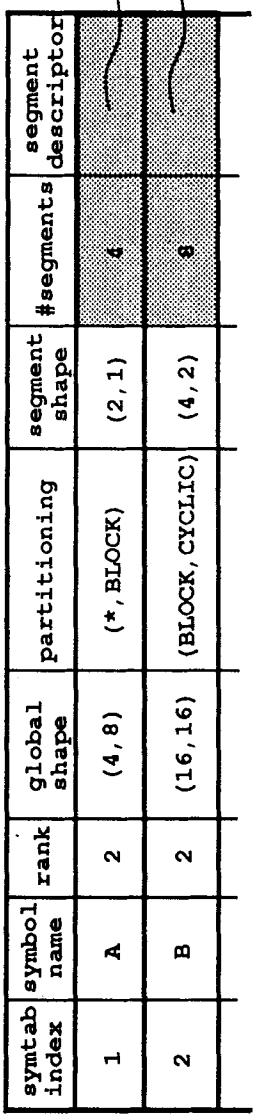

Fig. 5. XDP symbol table structure, showing the entries for two arrays A and B, partitioned over a 2×2 processor grid. The shaded entries are filled in at run-time by each processor.

The value of a transitional section is undefined, and an error could occur if the value of a read accessible variable were to be changed. Nevertheless, XDP does not automatically check the state of a variable at run-time; instead, the state must be explicitly checked by calls to the appropriate intrinsics. For most types of statements, there is an intrinsic which determines whether the statement is legal, and there are also intrinsics that can be used as compute rules to ensure an error will not occur. For instance, an assignment into a variable `X` is safe if and only if `writeable(X)` returns true. Similarly, the statement `writeawait(X):{X=3}` cannot cause an error.

One way to ensure that a section is not referenced illegally is to precede each statement with the appropriate intrinsics. As we will see later, optimizations can remove unnecessary run-time checks. Assuming a proper translation of the source program into IL + XDP and valid optimizations (ones that never produce code that could evoke the `Error` condition) assignments and uses to exclusive variables can proceed without the run-time overhead of checking the variables' states. (However, it would be a good idea to have run-time checks in place while developing the compiler.)

The rationale for some of the details of XDP is to allow the compiler to manage send and receive buffers explicitly. Another point is that ownership sends can proceed even if there are outstanding value sends. However, we decided to disallow ownership sends while there are outstanding receives. This decision was made to avoid the run-time complication of having to forward received values of sections that are no longer owned.

3. RUN-TIME STRUCTURES

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`, `readawait`, `writeawait`, `readable`, and `writeable` predicates may need to be evaluated at run-time. 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 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. We illustrate this with the use of *segments*.

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 examples, we assume a fixed, known processor grid and partitioning as allowed in HPF.[9]

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 compiler's symbol table, the run-time XDP symbol table only contains information about exclusive sections.

Figure 5 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 2×2 processor grid. Each exclusive variable has a symbol table entry. 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. The last two fields of the XDP symbol table are shaded dark in Fig. 5, to indicate that these entries are filled in only at run-time.

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 segment (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`, `read accessible` or `R/W accessible`). In our implementation, the segment descriptor data structure is declared as:

```
struct SegmentDesc {
  int  state;     /* accessibility state   */
  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];
```

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 6 illustrates two different partitioning schemes for a 4×8 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 in the 3-D FFT example.

If the code running on a processor executes an `iown( )` intrinsic at run-time, the symbol table entry for the array variable named in the query is used as follows. 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

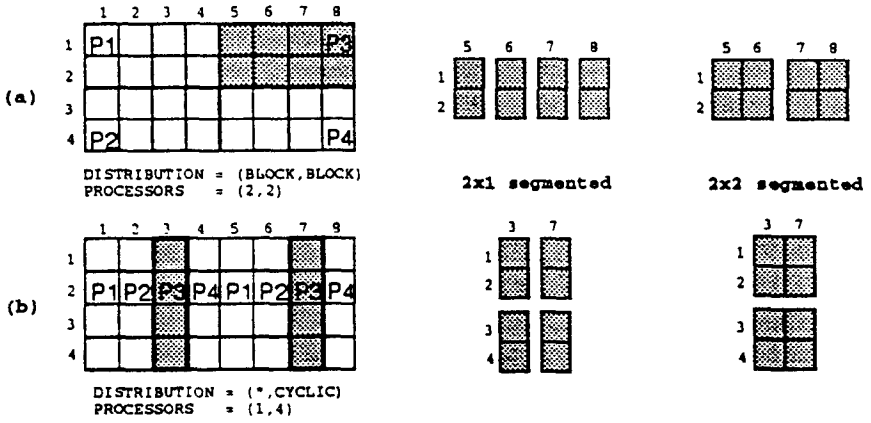

Fig. 6. Example distributions and local segmentations of a 4×8 array C, shown for processor $P3$.

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 2×2 processor grid, and 2×1 segmented (as shown in Figure 6(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 1×2 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. (Although the algorithm we described for evaluating `iown( )` involves examining the entire segment descriptor array, more efficient algorithms could be developed.) The intrinsics `readable` and `writeable` are handled similarly by consulting the appropriate symbol table entry. `readawait` and `R/W await` might involve waiting for the `state` field of the symbol table entry to be changed due to the completion of an outstanding communication.

When any send or receive is initiated or completed on a segment, the `state` 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 are 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 determining the id of the processor that 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, techniques[10] 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.

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:N, 1:N, 1:N]` which is initially distributed as `(*, *, BLOCK)` over a linear array of processors. 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 following programs illustrate the steps involved in the translation and optimization of the FFT program. The program in Fig. 7 is an SPMD Program with HPF directives for the distribution and redistribution. We remark that such a program could also have resulted from a FORTRAN program without HPF directives. The compiler could use known techniques[1-3] to automatically determine a good distribution and redistribution.

This program could execute in SPMD mode on each processor by copying all the data and ignoring the directives. However, having all the processors do all the work is not an efficient way to execute a program. The IL + XDP program in Fig. 8 shows a straightforward translation of the initial program using the owner-computes rule.

```
C Distribute A as (*,*,BLOCK)
  !HPF$ DYNAMIC A
  !HPF$ DISTRIBUTE A(*,*,BLOCK)

C Phase1: 1-D FFT in the j direction
  do k = 1, N
     do i = 1, N
        fft1D(A[i,*,k])
     enddo
  enddo

C  Phase2: 1-D FFT in the i direction
   do k = 1, N
      do j = 1, N
         fft1D(A[*,j,k])
      enddo
   enddo

C  Phase3: Redistribute A as (*,BLOCK,*)
  !HPF$ REDISTRIBUTE A(*,BLOCK,*)

C  Phase4: 1-D FFT in the k direction
   do j = 1, N
      do i = 1, N
         fft1D(A[i,j,*])
      enddo
   enddo
```

Fig. 7. SPMD program with HPF directives.

```
C Distribute A as (*,*,BLOCK)
  do k = 1, N
     do j = 1, N
        do i = 1, N
           iown(A[i,j,k]): {A[i,j,k] -=>}
        enddo
     enddo
  enddo
  BS = ceiling(N/numprocs)
  mylo = mypid*BS+1
  myhi = min(N,mypid *(BS+1))
  A[*,*,mylo:myhi] <=-

C Phase1: 1-D FFT in the j direction
   do k = 1, N
      do i = 1, N
         writeawait(A[i,*,k]): {fft1D(A[i,*,k])}
      enddo
   enddo

C Phase2: 1-D FFT in the i direction
  do k = 1, N
     do j = 1, N
        writeawait(A[*,j,k]):{fft1D(A[*,j,k])}
     enddo
  enddo

C Phase3: Redistribute A as (*,BLOCK,*)
  A[*,*,mylo:myhi] -=>
  A[*,mylo:myhi,*] <=-

C Phase4: 1-D FFT in the k direction
  do j = 1, N
     do i = 1, N
        writeawait(A[i,j,*]:{fft1D(A[i,j,*])}
     enddo
  enddo
```

Fig. 8. Initial IL + XDP program.

Translation of the initial distribution of A is general: the elements of A could reside on any processor, and thus no initial knowledge of the ownership of A is assumed. However, if there is analysis that determines the prior distribution as in Ref. 11, simpler code for ownership sends can be generated.

A typical optimization step 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, in Phase1, the compiler can determine that when k is not between mylo and myhi, the writeawait will evaluate to false, allowing the inner loop to be eliminated for those values of k. The improved code segment is shown here.

```
C Phase1: 1-D FFT in the j direction
     do k = mylo, myhi
        do i = 1, N
           writeawait(A[i,*,k]): {fft1D(A[i,*,k])}
        enddo
     enddo
```

Phase2 and Phase4 benefit from similar optimization. Phase2 allows the added optimization of eliminating the writeawait compute rule altogether. This is a consequence of two considerations: when writeawait returns, it always has the value true since the loop bounds have been adjusted appropriately, and no synchronization is needed since there is no intervening communication after Phase1 made sure the data are R/W accesible. The resulting code for Phase2 is:

```
C Phase2: 1-D FFT in the i direction
  do k = mylo, myhi
     do j = 1, N
       fft1D(A[*,j,k])
     enddo
  enddo
```

The Redistribute in Phase3 relies on the run-time system to figure out which sections of A get moved to which processors. However, some of this work can be done at compile time. First, the data can be partitioned into sections so that each processor sends only one section to each other processor. The variables jlo and jhi are introduced to bound the columns that need to be sent, and similarly klo and khi delimit the data being received. Second, some of the ownership transfers are redundant, namely those sent and received from a processor to itself. These can be

eliminated by inserting tests into both the send and receive loops. The results are as shown:

```
C Phase3: Redistribute A as (*,BLOCK,*)
  do idto = 0,numprocs-1
     jlo = idto*BS+1
     jhi = min(N,idto*(BS+1))
    if (idto.ne.mypid) A[*,jlo:jhi,mylo:myhi] -=>
enddo
do idfrom = 0,numprocs-1
   klo = idfrom*BS+1
   khi = min(N,idfrom*(BS+1))
   if (idfrom.ne.mypid) A[*,mylo:myhi,klo:khi] <=-
enddo
```

Even with the improvements mentioned earlier, the program must complete Phase2 before beginning the redistribute operation. An extremely aggressive optimizing compiler might arrange for the ownership transfer to be "pipelined" so that as sections are computed in Phase2, the appropriate sends and receives are initiated. This would allow overlapping communication with computation. A substantial amount of reorganization is required to realize the benefit of such pipelining. Standard optimizations like loop fusion are needed, as well as more advanced transformations. In particular, it would be undesirable for all the processors to flood processor 0 with results, then inundate processor 1, and so on. More balanced communication results if processor `i` first computes and transfers the data that goes to processor `i+1`, then `i+2`, and so on. The entire program shown in Fig. 9 illustrates pipelined communication. (We remind the reader that the purpose of this paper is to present a language in which various code improvements can be expressed. We do not wish to imply that we know how to find all such improvements automatically!)

Finally, we mention an interesting question that arises concerning the optimization of `await` instructions. It is possible for the compiler to move the `writeawait` in Phase1 (or similarly for Phase4) out of the inner loop, as shown here:

```
C Phase1: 1-D FFT in the j direction
     do k = mylo, myhi
        writeawait(A[*,*,k]): {
           do i = 1, N
              fft1D(A[i,*,k])
           enddo
        }
     enddo
```

```
C A is distributed as (*,*,BLOCK)
  do k = 1, N
     do j = 1, N
        do i = 1, N
           iown(A[i,j,k]): {A[i,j,k] -=>}
        enddo
     enddo
  enddo
  BS = ceiling(N/numprocs)
  mylo = mypid*BS+1
  myhi = min(N,mypid *(BS+1))
  A[*,*,mylo:myhi] <=-

C Phase1: 1-D FFT in the j direction
  do k = mylo, myhi
     do i = 1, N
        writeawait(A[i,*,k]): {fft1D(A[i,*,k])}
     enddo
  enddo

C Phase2&3: FFT in the i direction and redistribute
  do distance = 0, numprocs-1
     idto = mod(mypid+distance,numprocs)
     jlo = idto*BS+1
     jhi = min(N,idto*(BS+1))
     do k = mylo, myhi
        do j = jlo, jhi
          fft1D(A[*,j,k])
        enddo
     enddo
     if (idto.ne.mypid) A[*,jlo:jhi,mylo:myhi] -=>
     idfrom = mod(mypid-distance,numprocs)
     klo = idfrom*BS+1
     khi = min(N,idfrom*(BS+1))
     if (idfrom.ne.mypid) A[*,mylo:myhi,klo:khi] <=-
 enddo

C Phase4: 1-D FFT in the k direction
     do j = mylo, myhi
        do i = 1, N
           writeawait(A[i,j,*]): {fft1D(A[i,j,*])}
        enddo
     enddo
```

Fig. 9. Pipelined IL + XDP program.

In fact, the `writeawait` could be moved outside of the outer loop also. There is a trade-off involved in such motion. By reducing `N` calls to `writeawait` on small sections of data by one call on a larger section, there may be less overhead. On the other hand, leaving `writeawait` *inside* a loop may allow operations to proceed that might otherwise have to wait for the arrival of larger sections of data. Thus, whether such code motion is profitable depends on implementation considerations.

5. ANALYSIS AND OPTIMIZATION

Compiler optimizations that affect data movement and ownership issues can be represented as transformations to the IL + XDP code. Some of these optimizations fit naturally into the framework of traditional optimizing compilers for sequential languages. We will illustrate this by first describing how data flow analysis of IL + XDP can be performed, and then showing how some optimizations can proceed using this information.

Other optimizations, particularly those that redistribute ownership among the processors, require more advanced techniques. An example would be, in the FFT section, the change of communication order when Phases 2 and 3 were combined. Such optimizations need more information than is provided by traditional data flow analysis, both to identify what transformations are *legal*, and also which are *desirable*. While these transformations are perhaps the most interesting, they are also beyond the scope of this paper. The point we wish to make here is that XDP is designed to facilitate the incorporation of optimizations that have been developed for both data-parallel and message-passing languages.

5.1. Analysis

To support the more traditional optimizations, our approach is to analyze the program that is run on one of the processors without instantiating the value of `mypid`. This analysis will therefore be valid for all processors. Data and ownership transfers, as well as calls to the intrinsics, are treated as calls to a run-time library. The data flow effects of these calls are determined by the semantics of the library routines. Given this information, data flow analysis can proceed using standard techniques.[12] This analysis determines the *def*, *use*, and *ref* information for each section at each statement of the program, where *def* means the statement assigns a new value to the section, *use* means the statement uses the section's value, and a *ref* is either type of reference. Finally, optimizations can be performed using the data flow information.

The data flow effects of the XDP constructs are determined by details of the run-time library. XDP allows some flexibility in the implementation of these routines. In particular, correct IL + XDP programs should not use (def) a variable after a receive (send) has been initiated but before the state has been checked with a call to an appropriate intrinsic. Thus, a correct program will behave identically independent of when, within this range, the transfer occurs or when the variable's state changes from transitional (read accessible) to R/W accessible occurs. Because of this, we can choose the operational semantics (which are used to derive to data flow effects of the run-time library) to be different from the actual implementation. We do so to enhance the optimization opportunities.

To this end, the operational semantics model execution by a single machine M that simulates all the processors. M runs the code of processor p until it encounters a *synchronization intrinsic* (`readawait` or `writeawait`), at which point it switches execution to other processors' code before returning to further simulation of p's code. During the course of executing p's code, M uses and updates p's run-time symbol table, in particular the state field of each exclusive section. Notice that, except at synchronization points, the *only* references to the symbol table (or any other variable) are the explicit actions of the simulated operations. Thus, the operational semantics (unlike the actual implementation) has no asynchronous communication or state changes.

The data flow effects for the various IL + XDP constructs are summarized in Fig. 4 and described later in more detail.

For each exclusive section `E` of the program, the compiler introduces a new variable, `E.state`, to represent the state field of the per-processor symbol table information for `E`. (Since sections of exclusive variables can overlap, uses and defs of `E.state` also affect other symbol table entries. Data access descriptors[13] or regular section descriptors[14] are appropriate techniques for summarizing these effects.) Whenever `E` is referenced in the program, it is a use of `E.state`, since M must consult `E.state` to determine if the section's state is valid for the operation. Additionally, any reference to `E` that has the potential of changing `E`'s state is a def of `E.state`. This includes all sends and receives, as well as the some of the intrinsics.

When a synchronization intrinsic is encountered, M may query what transfers have been initiated in order to determine which other processors can proceed. Thus, the two synchronization intrinsics must be considered to be uses of all symbol table entries. (These uses can prevent an optimization from changing the set of pending transfers that reach a synchronization intrinsic. This prevents optimizations from introducing deadlock.) The `readable(X)` and `writeable(X)` intrinsics must also be treated as

possible defs of `X.state`, giving M an opportunity to change `X.state` to R/W accessible.

The statement `E - >` is a use of `E` and `E < - X` is a def of `E`. That is, we consider the data transfer to occur at the transfer statement, even though the `E` remains in state read accessible or transitional until the corresponding intrinsic is executed.

Figure 4 omits the entries for "`- = >`" and "`< = -`" since these are obtained by combining the effects of the data and ownership transfer operations listed. Also, a guarded IL + XDP statement "`g:{S}`" is treated as "`if g then S`".

There are a number of enhancements to this basic data flow information that might be helpful for some optimizations. First, data flow analysis techniques for computing def-use chains and similar information make a distinction between *preserving defs*, which retain currently-live defs, and *killing defs*, which kill all reaching defs. When `E`'s ownership is transferred, it can be considered to be a killing def of `E.state`, since `E.state` is always set to unowned. The defs in the intrinsics are preserving defs; for instance, `readawait(E)` will leave `E.state` unchanged if it was unowned or read accessible, but change state transitional to R/W accessible.

Second, it is useful for the compiler to derive other information as well. Ideally, the compiler attempts to determine the distribution of ownership of all sections at each point in the program. Additionally, it can keep track of the source or target of a receive or send statement. This information may be present in the original program (for instance, the HPF redistribute statements in our FFT example), or derivable by suitable analysis techniques (such as reaching decompositions analysis[11]). Such information can be used by optimizations that redistribute the data and computation, and to bind the XDP communication primitives to efficient object code.

Finally, analysis of the parallel execution order[15-19] can yield more precise information about which communications statements can be reordered without introducing deadlock or other errors. This information can be represented by removing some of the uses of state variables at certain occurrences of synchronization intrinsics.

5.2. Optimization

In this section, we illustrate how traditional optimization techniques can be applied to the XDP constructs, resulting in some of the code improvements illustrated in the earlier examples.

5.2.1. Loop Fusion

From Ref. 20, loop fusion for two adjacent `do` loops L_1 and L_2 with the same loop bounds is valid iff there are not statements $S_1 \in L_1$ and $S_2 \in L_2$ such that there is a ref-ref dependence from S_1 on iteration i_1 to S_2 on iteration i_2 with $i_1 > i_2$. This criterion must be applied to both the variables of the original program and to the compiler-created state entries. This criterion is strong enough to fuse simple cases, such as the following:

```
do k = mylo, myhi
   A[*,*,k]= foo(k)
enddo
do k = mylo, myhi
   A[*,*,k]-=>
enddo
```

Applying loop fusion to these adjacent loops would yield:

```
do k = mylo, myhi
   A[*,*,k]= foo(k)
   A[*,*,k]-=>
enddo
```

This transformation allows a simple pipelining effect, where the value can be sent as soon as it is produced.

5.2.2. State Analysis and Optimization

It is possible for some unnecessary calls to XDP intrinsics to be eliminated or simplified by optimizations, particularly if the compiler performs constant propagation on the state variables. For instance, a straightforward translation of source code to IL + XDP might produce code fragments such as:

```
iown(A) : {
           A <- Y
           readawait(A) : {B = A+1}
           writeawait(A) : {A = 5}
          }
```

Within the scope of the `iown(A)` compute rule, `A.state` cannot be unowned. Thus, the two await statements are certain to evaluate to `true`, allowing conditional branches to be eliminated. This change is reflected in the IL + XDP code as:

```
iown(A) : {
           A <- Y
           readawait(A)
           B = A+1
           writeawait(A)
           A = 5
          }
```

Summary information derived from Fig. 3 shows that the A < − statement will leave `A.state` transitional and the `readawait(A)` will change it to R/W accessible. This should enable the compiler to eliminate the `writeawait` intrinsic entirely, leaving:

```
iown(A) : {
           A <- Y
           readawait(A)
           B = A+1
           A = 5
          }
```

Similarly the occurrence of `iown(A)` might be eliminated if for all defs of `A.state` that reach the `iown(A)`, `A.state` cannot be set to state unowned.

Other methods for identifying redundant synchronization appear in Refs. 21–24.

5.2.3. Compute Rule Elimination

Compute rule elimination has been used by distributed address space compilers such as in Refs. 4, 7, and 8, to speed up SPMD programs. Given knowledge of data placement, the compiler can sometimes adjust the bounds of a loop so that a processor will execute only those iterations whose data are local to the processor. For example, consider the simple owner-computes example in Section 2.2. Assume A and B are distributed as blocks of size BS, where `n=numprocs*BS`. The compiler can split the original loop (`do i=1, n`) into loops with three disjoint ranges (`1` to `mylo − 1`, `mylo` to `myhi`, and `myhi+1` to `n`) using index set splitting.[25] Then the compiler can eliminate the first and third loops by dead code elimination, resulting in the following code:

```
mylo = BS*mypid
myhi = BS*mypid+BS-1
```

```
do i = mylo, myhi
  iown(B[i]) : { B[i] -> }
  iown(A[i]) : {
                 T[mypid] <- B[i]
                 readawait(T[mypid])
                 A[i] = A[i] + T[mypid]
               }
enddo
```

Further analysis and optimizations might now eliminate the two `iown` compute rules since they always evaluate to `true`, replace the now-adjacent send and receive of `B[i]` by the assignment "`T[mypid]=B[i]`", and remove the `readewait` since the section `T[mypid]` is certain to be in state R/W accessible. The resulting code would be:

```
do i = mylo, myhi
   T[mypid] = B[i]
   A[i] = A[i] + T[mypid]
enddo
```

Finally, value numbering or variable propagation could eliminate the unnecessary `T[mypid]`.

5.2.4. Optimization of Communication

Knowledge of current ownership information at each point in the program can help to eliminate redundant communication. In the example of Section 5.2.3, we saw that if the same processor that exclusively owns `A[i]` also owns `B[i]`, then the data transfer statements can be eliminated.

Even if they cannot be eliminated, the compiler may be able to *vectorize*[11] the messages. In that paper, following an algorithm of Refs. 26 and 27, for a given statement, all true data dependences to the statement are examined to determine the loop level at which messages can be combined. In Ref. 4, message tags are inserted to mark the level of the necessary communication for message vectorization. In XDP, the dependences from actual transfer statements to the statement can be examined, and the statements can actually be moved to the appropriate loop level and combined. Suppose in our example, `A` and `B` are distributed in blocks of the same size but not aligned. In this case, the data transfer statements can be moved out of the loop as shown later. Note that analysis can also determine that the `writeawait` can be moved as well.

```
iown(B[mylo:myhi]) : {B[mylo:myhi] -> }
iown(A[mylo:myhi]) :
        {T[mylo:myhi] <- B[[mylo:myhi]}
writeawait(T[mylo;myhi])
do i = mylo,myhi
   A[i] = A[i] + T[i]
enddo
```

5.2.5. Other Optimizations

Many other traditional compiler optimizations (e.g., dead code elimination, value numbering, and so on) can proceed as usual based on the data flow information described earlier. Optimizations that affect the order of statements (such as invariant code motion and statement reordering) will be prohibited from moving synchronization intrinsics across send and receive statements since the transfer statement is a def of some section's state, and the synchronization is a use of all sections' states. This prevents optimizations from introducing deadlock. The restriction could be relaxed to allow optimizations that increase, but not decrease, the set of sections for which sends and receives have been initiated. More sophisticated analysis can allow even more aggressive code motion.

Some optimizations can only be done with knowledge of the underlying architecture. For instance, if the communication primitives of the underlying machine are nonblocking, 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.

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.

6. COMPARISON TO OTHER INTERMEDIATE LANGUAGES

Traditional optimizing compilers[12] use relatively language- and machine-independent intermediate program representations. Some IL's for these compilers[12] first generate load and store operations assuming an

infinite number of symbolic registers, and then assign these to real machine registers. However, these compilers do not otherwise represent data movement and placement between devices in an explicit manner, as done in XDP. Lake[28] 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 in Ref. 29. The KSR1[30] implements ownership transfers automatically in hardware. A preliminary version of this paper appeared in Ref. 31. Here, the semantics of XDP send statements has been changed so buffering of messages is not required, states of a section have been adjusted accordingly and are discussed in more detail, and a method for including XDP primitives in data flow analysis and further details about optimization are given.

In the remainder of this section we describe several existing IL's for compilers developed for distributed address space multiprocessors and compare them with XDP. These range from IL's with language- or machine-specific representation of data transfers, to no explicit representation of data transfer, except as auxiliary data structures to the IL, to IL's with explicit machine-independent representation of data but not ownership transfer.

The compiler developed at Rice[32] uses a high-level language as IL: Fortran77 with data decompositions, FORALL, and FORTRAN90 intrinsics such as CSHIFT, PACK, and UNPACK, and reductions such as SUM and DOTPRODUCT. Thus the data transfer operations here are language-dependent. The back-end compiler uses Fortran77 plus machine-specific message-passing calls as IL. Hence at no level does this compiler represent data and ownership transfer in a language- or machine-independent manner.

The FortranD compiler[4] developed at Rice uses message-tags[4] to indicate where communication must be inserted. Although these tags can be moved, it seems clear they are not operations with semantics like XDP operations, and cannot be manipulated by the compiler in the same manner as an operation like addition.

In fact, the FortranD compiler has embraced the philosophy that program analysis should come first and drive code generation, rather than inserting guards and element-wise messages and then performing optimization to obtain more efficient code.[4] For example, by manipulating only Fortran77 code, there is no possibility of introducing deadlock. This style of compilation is compatible with XDP in that compute rules and transfer operations need not be initially generated. While we agree that initial program analysis is both necessary and beneficial, we also believe that unless there is a direct representation of data and ownership transfer at the

right level, the compiler will have a more difficult job of manipulating these operations. Allowing their representation as in XDP might afford this compiler greater opportunities for optimization, particularly in getting beyond the owner-computes rule.

The Kali system[33] calculates send and receive sets like FortranD but they are not represented as operations.

Several existing distributed address space compilers represent data transfer in a language- and machine-independent manner. The SUPERB system[6] generates general EXchange Send and Receive (EXSR) statements to communicate overlaps. Similarly, Rogers and Pingali[8] insert send and receive operations from an abstract message-passing machine model into their IL. Callahan and Kennedy[7] insert general load and store operations before actual communication generation. Crystal[34] generates communication actions prior to analyzing specific communication patterns. None of these represent ownership transfer operations in the manner suggested by XDP.

7. CONCLUSIONS AND FUTURE WORK

The XDP methodology has been designed to give the compiler the power to manipulate data and ownership transfer. 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 separations of data transfer from local computation, its nonblocking 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. A run-time symbol table allows XDP to be implemented as an extension to a compiler's intermediate language.

We have given here some optimizations which use the explicit representation of data and ownership transfer in XDP. Future work lies in the development of further optimizations and their evaluation.

While XDP has been designed with distributed address space machines in mind, future work will include the application of its key ideas in more general contexts. In particular, we would like to use it 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, David Culler, Guang Gao, Dave Gelernter, Dirk Grunwald, François Irigoin, Kathy Knobe, Piyush Mehrotra, Sam Midkiff, Anne Rogers, Randy Scarborough, Edith Schonberg, Harini Srinivasan, Guy Steele, Hans Zima, the PPoPP93 program committee, and the referees for their input. A preliminary version of this paper appeared in PPOPP93.

REFERENCES

1. M. Gupta and P. Banerjee, Automatic data partitioning on distributed memory multiprocessors, *Proc. of the Sixth Distributed Memory Computer Conf.* (DMCC6), Portland, Oregon (April 1991).
2. M. Gupta and P. Banerjee, Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers, *IEEE Trans. on Parallel and Distributed Systems* (April 1992).
3. K. Knobe, J. D. Lukas, and G. L. Steele, Jr., Data optimization: allocation of arrays to reduce communication on SIMD machines, *J. of Parallel and Distrib. Comput.* **(8)**:102–118 (1990).
4. Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng, Compiling Fortran D for mimd distributed-memory machines, *Comm. of the ACM*, **35**(8):66–80 (August 1992).
5. V. Balasundaram, Translating control parallelism to data parallelism, *Fifth SIAM Conf. on Parallel Processing for Sci. Comput.*, Houston, Texas (March 1991).
6. H. P. Zima, H. J. Bast, and M. Gerndt, SUPERB: A tool for semi-automatic SIMD/MIMD parallelization, *Parallel Computing* **6**:1–18 (1988).
7. D. Callahan and K. Kennedy, Compiling programs for distributed-memory multiprocessors, *J. of Supercomput.* **2**:151–169 (October 1988).
8. A. Rogers and K. Pingali, Process decomposition through locality of reference, *ACM SIGPLAN 89 Conf. on Programming Language Design and Implementation*, pp. 69–80 (June 1989).
9. HPF Forum, *High Performance Fortran Language Specification*, Version 1, available from Rice University, Houston, Texas (January 1993).
10. J. Wu, J. H. Saltz, S. Hiranandani, and H. Berryman, Runtime compilation methods for multicomputers, *Proc. of the 1991 Int. Conf. on Parallel Processing*, St. Charles, Illinois (August 1991).
11. Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng, Compiler optimizations for Fortran D on mind distributed-memory machines, *Proc. of Supercomputing '91* (November 1991).
12. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley (1988).
13. V. Balasundaram, A mechanism for keeping useful internal information in parallel programming tools: the Data Access Descriptor, *J. of Parallel and Distrib. Comput.* **(9)**:154–170 (1990).
14. D. Callahan and K. Kennedy, Analysis of interprocedural side effects in a parallel programming environment, *J. of Parallel and Distrib. Comput.* **5**(5):517–550 (October 1988).

15. D. Callahan and J. Subhlok, Static analysis of low-level synchronization, *Workshop on Parallel and Distributed Debugging*, pp. 100–111 (May 1988).
16. D. Callahan, K. Kennedy, and J. Subhlok, Analysis of event synchronization in a parallel programming tool, *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 21–30 (March 1990).
17. D. Grunwald and H. Srinivasan, An efficient construction of parallel static single assignment form for structured parallel programs, Technical Report CU-CS-564-91, University of Colorado, Boulder (1991).
18. D. Grunwald and H. Srinivasan, Data flow equations for explicitly parallel programs, *Proc. of SIGPLAN Conf. on Principles and Practice of Parallel Programming* (May 1993).
19. D. Grunwald and H. Srinivasan, Efficiently computing preserved sets, Technical Report in preparation, University of Colorado, Boulder (1993).
20. Hans Zima and Barbara Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press, Addison-Wesley (1991).
21. D. N. Jayasimha, Communication and synchronization in parallel computation, PhD. Thesis, CSRD Report No. 819, University of Illinois, Urbana-Champaign (1988).
22. S. P. Midkiff and D. A. Padua, Compiler generated synchronization for DO loops. In K. Hwang, S. Jacobs, and E. Swartzlander (eds.), *Proc. of the 1986 Int. Conf. on Parallel Processing*, pp. 544–551 (August 1986).
23. P. L. Shaffer, Minimization of interprocessor synchronization in multiprocessors with shared and private memory. In F. Ris and P. M. Kogge (eds.), *Proc. of the 1989 Int. Conf. on Parallel Processing*, Vol. 3, pp. 138–141 (August 1989).
24. S. P. Midkiff, The dependence analysis and synchronization of parallel programs, PhD. Thesis, CSRD Report No. 1165, University of Illinois, Urbana-Champaign (January 1992).
25. S. Carr and K. Kennedy, Compiler blockability of numerical algorithms, *Supercomputing 92*, Minneapolis, Minnesota (November 1992).
26. V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, An interactive environment for data partitioning and distribution, *Proc. of the Fifth Distrib. Memory Comput. Conf.*, Charleston, S. Carolina (April 1990).
27. M. Gerndt, Updating distributed variables in local computations, *Concurrency: Practice and Experience* (1990).
28. T. Lake, Distributing computations. In R. H. Perrott (ed.), *Software for Parallel Computers*, Chapman and Hall, London (1992).
29. E. Jul, H. Levy, N. Hutchinson, and A. Black, Fine-grained mobility in the Emerald system, *ACM Trans. on Computer Systems* **6**(1):109–133 (February 1988).
30. Kendall Square Research, Technical summary, Technical report, Kendall Square Research (1992).
31. Vasanth Bala, Jeanne Ferrante, and Larry Carter, Explicit data placement (xdp): A methodology for explicit compile-time representation and optimization of data movement, *Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 139–149 (May 1993).
32. Chau-Wen Tseng, An optimizing Fortran D compiler for MIMD distributed-memory machines, PhD. Thesis, Rice University (1993).
33. P. Mehrota and J. Van Rosendale, Compiling high level constructs to distributed memory architectures, *Proc. of the Fourth Conf. on Hypercube Concurrent Computers and Applications* (March 1989).
34. J. Li and M. Chen, Generating explicit communication from shared memory program references, New York, *Supercomputing 90*, pp. 865–877 (November 1990).