

Data Flow Analysis Across Tuplespace Process Boundaries

James B. Fenwick Jr.

Lori L. Pollock

Department of Computer and Information Sciences

University of Delaware

Newark, DE 19716

{fenwick,pollock}@cis.udel.edu

Abstract

A current limitation of compilers for shared memory parallel languages is their restricted use of traditional code-improving transformations, such as constant propagation and dead code elimination. A major problem lies in the lack of data flow analysis techniques for programs with user-specified parallelism. In this paper, we demonstrate how data flow analysis remains quite viable in a compiler for shared memory parallel programs in a structured distributed shared memory environment, in which a shared space of tuples is accessed by properly synchronized methods. We demonstrate standard intraprocess data flow analysis performed in the midst of tuplespace communication statements, and present improvements to the precision of the analysis in the presence of these statements. We present a data flow system to compute reaching definitions across process boundaries, and a technique to improve the precision of this interprocess analysis. Lastly, some transformations enabled by this analysis are presented.

1 Introduction

Distributed memory multiprocessors and networks of workstations offer scalability for large parallel applications. The programmer has several options in programming these systems to exploit the available parallelism: message passing, or the use of a parallelizing compiler targeting distributed shared memory, or user-specified shared memory parallelism via a parallel language. Because writing efficient message passing programs is difficult, error-prone, and tedious, the distributed shared memory paradigm is receiving increased attention. The shared memory abstraction provides an easier transition from a sequential to a correct parallel program. While a shared memory, parallelizing compiler provides the applications programmer with an easy avenue to parallelization, programming languages that allow the user to explicitly

specify parallel constructs are becoming more prevalent. Sometimes, it is impossible to express a parallel algorithm using a sequential language, even with parallelization directives.

Regardless of whether the global shared address space is implemented in hardware or software, compilers for shared memory parallel languages targeting a physically distributed memory architecture perform sophisticated analysis and transformation to effectively exploit parallelism and deal with communication between the physically distributed memories. However, current compilers for parallel programs with user-specified parallelism either do not perform traditional data flow analysis to enable classic scalar optimizations, or restrict the scope to specific sequential segments, enabling little to no scalar optimization across parallel constructs. Meanwhile, data flow analysis and scalar optimization have played a key role in ambitious optimizing and parallelizing compilers. A global or interprocedural analysis of the program to collect information about variable uses and definitions, and expressions, enables subsequent determination of the safety of code-improving transformations such as constant propagation, dead code elimination, common subexpression elimination, and loop invariant code motion. While data flow analysis frameworks have been developed for behavior analysis of parallel programs with the goal of detecting synchronization errors and data usage errors, little work has focused on data flow analysis to enable traditional scalar optimizations.

Previous investigations into data flow analysis of parallel programs[17, 15, 20], transformation of a parallel program into Static Single Assignment (SSA) form[31], and data dependence tests for guaranteeing the legality of parallelizing transformations[23, 24] all focused on shared memory parallel programs with lexically-specified parallel constructs, such as

`cobegin/coend` or `parallel_sections`. These studies demonstrated how the memory consistency model of the parallel programming language has a large effect on the complexity of the data flow equations and dependence testing. They also indicated that with any of the memory consistency models, the analysis for optimization of this style of shared memory parallel programs involves considerable rethinking and modification of the sequential analysis techniques.

In this paper, we describe our work in developing data flow analysis systems within a structured distributed shared memory paradigm that uses generative communication[13]. The generative communication paradigm of parallel programming offers the simplicity of shared memory parallel programming and only a small number of primitives for coordination, while also providing flexibility, power, and the potential to scale like message passing. Rather than sharing variables, processes share a data space. Messages are not sent from one process to another, but rather are placed in the shared data space for other processes to access. The data placed into the shared space are termed *tuples* and thus the shared space is called *tuplespace*. Tuplespace is an associative memory meaning that tuples do not have addresses but rather are referenced by their content. The actual implementation of the parallel program on the target architecture is hidden from the programmer, and the architecture can be any number of platforms ranging from shared or distributed memory to networks of workstations.

Compile-time analysis has been developed to structure tuplespace to significantly reduce the associative search cost [4], and run-time strategies have been developed to counteract observed communication inefficiencies[1]. Compile-time methods targeting tuplespace communication improvements have also been developed [3, 12, 19]. Indeed, it has been demonstrated that distributed tuplespace implementations can be efficient [8, 22], and these performance studies have analyzed a wide variety of real applications encompassing a large scope of parallel algorithm classifications.

Our implementation of the generative communication model is an implementation of Linda¹ tuplespace, the best known implementation of the generative communication model. We have built a Linda optimizing compiler based on the SUIF compiler infrastructure[32], and a distributed tuplespace runtime system[11]. The runtime system executes on a network of SUN workstations. We have been devel-

oping and implementing analysis and code-improving transformations directed toward communication improvements of tuplespace on a network of workstations. Our experiences in developing this optimizing compiler have uncovered situations where classic code-improving transformations such as constant propagation and folding, common subexpression elimination, and dead code removal would not only increase the efficiency of individual computational processes, but also increase the opportunities for communication optimization.

This paper describes the results of our efforts to incorporate data flow analysis and classic code-improving transformations into our Linda optimizing compiler. We specifically address the following questions: (1) How do parallel constructs in this paradigm affect the standard data flow analyses? (2) How do the issues of applying data flow analysis in this paradigm differ from the shared memory parallel programming model studied by others? (3) Is a new framework required, or can the traditional data flow analysis framework be used, with or without modification, and still guarantee legality of optimizing transformations? (4) Can the precision of the analysis be improved? (5) Can data flow between processes be analyzed at compile time in a practical manner?

After providing pertinent background on the generative communication model, we discuss our findings for data flow analysis within separate processes in isolation. We present a new data flow framework that enables characterization of the data flow between parallel processes, and describe how to improve the precision of this information. We describe the framework in the context of reaching definitions. The paper concludes with some example transformations that utilize the gathered data flow information and a discussion of related work.

2 Tuplespace

```

main() {                                player1() {                             player2() {
    EVAL(player1());                    int play;                                int play;
    EVAL(player2());                    RD("play",?play); RD("play",?play);
    OUT("play",100);                    while(play--){                            while(play--){
    OUT("ping");                        IN("ping");                               IN("pong");
    IN("done");                         OUT("pong");                              OUT("ping");
    IN("done");                         }                                           }
}                                        OUT("done");                              OUT("done");
}                                        }                                           }

```

Figure 1: Example tuplespace ping pong program

Tuplespace is the shared data space central to

¹Linda is a registered trademark of Scientific Computing Associates, New Haven, Conn.

the generative communication parallel programming model, most notably embodied by Linda. This model makes no assumptions regarding the underlying architecture, and has been implemented on shared and distributed memory machines. Tuplespace is distinct from processor local memory (if any) and equally accessible by any processor. A tuple is an ordered collection of typed fields which are either data objects or place holders. The field types are dependent on the underlying sequential language. The tuplespace operations are atomic and provide process creation, inter-process communication, and process synchronization. The tuples in tuplespace are manipulated by the operations: OUT, EVAL, IN, RD, INP, RDP.

Tuplespace contains data tuples inserted by OUT operations and process tuples inserted by EVAL operations. When a process tuple completes its processing, it quiesces to a data tuple. These generation operations are non-blocking. Data tuples are removed from tuplespace by the IN operation. Values of fields in the data tuple are copied into the address space of the process issuing the IN operation, wherever the IN indicates a *formal* field with a '?' syntax. A field that is not formal is termed *actual*. The IN primitive is a blocking operation. The RD operation is another synchronous receive which acts like the IN, only it does not remove the tuple from tuplespace. If a matching tuple is not present in tuplespace, the process issuing an IN or RD operation blocks until a match is inserted into tuplespace. The INP and RDP operations are predicate versions of their counterparts. They do not block when a matching tuple is not present in tuplespace but rather return a false value. New processes are created to evaluate the fields of an EVAL operation. In response to the high cost of process creation, many Linda implementations only create processes for the function-valued fields of an EVAL. This is how the programmer explicitly creates parallelism.

Figure 1 shows an example tuplespace program. The *main* process creates two *player* processes, and deposits two tuples into tuplespace. The *player* processes both use a formal field in the RD operation to find out how many times to play.

3 Program Representation

The ultimate goal of our research is compile-time analysis to identify opportunities for tuplespace communication optimizations [12, 10]. For these analyses, a high level representation of the parallel program is the most appropriate. Specifically, we need all the information available in the tuplespace operations. Unfortunately, most tuplespace programming systems lose this high level information in much the same way

as high level array access information is lost in low level intermediate representations. The tuplespace operation information is lost because typically these operations are mapped to calls to a message passing library. To retain the necessary high level information, we perform our analyses on a high level intermediate program representation. All the standard data flow analyses remain feasible, and now tuplespace communication analysis is also effectively supported.

In particular, we have extended the high level representation developed for the SUIF shared memory parallelizing compiler[32] to serve as our high level representation. The SUIF system provides parsing modules for C and Fortran, which we have modified to support the additional syntax for tuplespace operations. Each tuplespace operation outwardly appears like a procedure call, but is further annotated with high-level information, including its tuplespace partition,² an indication of whether each field is formal or actual, and other useful information. Processes are identified by examining the EVAL tuplespace operations, which contain the name of the procedure that will be executed in parallel. The definitions of the named procedures are annotated to indicate that they are process entry points.

Tuplespace communications are modeled by constructing directed edges from tuplespace generation operations (OUT, EVAL) to extraction operations (IN, RD, INP, RDP). Naive communication edge construction would result in a complete bipartite graph with edges from each generation operation to each extraction operation. However, implementing Carriero's tuplespace partitioning improvement [4] allows the elimination of many of these naive edges by restricting communication to occur only within each disjoint tuplespace partition.

The entire program can be viewed as a forest of process intermediate representations, where each process is a collection of procedures. The communication edges connecting these process representations form our Linda intermediate representation. Currently, procedures that are used in more than one process are cloned to simplify interprocedural analysis within processes, but only one representation of each procedure is maintained for each process regardless of the number of invocations of the procedure by that process. Similarly, only one process representation is maintained regardless of how many instantiations of the process are made at runtime.

²Carriero [4] made the associative memory of tuplespace efficient by partitioning a program's tuplespace operations into disjoint sets thus reducing the associative search to only a single partition, rather than all of tuplespace.

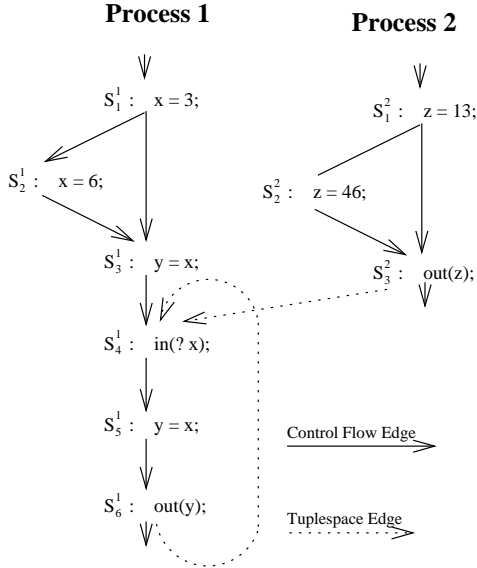


Figure 2: Example program representation

There are a number of interprocedural representations described in the literature [2, 9, 18, 27]. The current version of our compiler builds an interprocedural flow graph similar to [9]. Each procedure is represented in the form of the SUIF IR, and it is not necessary to have all procedure representations in memory at once. Interprocedural execution paths are not explicitly represented by edges in the IR, and similarly there are no edges from process invocation sites to process entry points.

Figure 2 depicts an example program representation of a program with two processes that are connected by tuplespace edges. In this example there is only one tuplespace partition, so each OUT is connected to every IN operation.

4 Intraprocess Data Flow Analysis

The basic abstraction of global shared memory is a programming environment in which each process can access any data item, without the programmer having to worry about where the data is located, or how to obtain its value. All processes have access to the same address space, and different processes can access, and more importantly write to, the same global address. This is the heart of the problem in statically characterizing the data flow in shared memory parallel programs[23]. Intuitively, a *read* should always return the last value written to a given variable. In order to determine the data flow through a single process, you need to know something about the possible *write*'s (in both this process and other processes) to the shared

variables used in this process.

In contrast, tuplespace processes share a logically global data space, but address spaces of the tuplespace processes are distinct. While the tuplespace model requires an extra level of copying (involving network access for distributed memory tuplespace implementations), it does simplify analysis. Intuitively, a memory location in one process can not be accessed by another process. Therefore, only the actions of process i itself need to be examined in order to analyze how definitions and uses of variables accessible to process i flow within process i .

Consider the following situation. In order to *increment* the value of a “global” variable, represented by the tuple (“x”, 3) in tuplespace, an IN(“x”, ?x) operation takes the old tuple out of tuplespace, and a succeeding OUT(“x”, x+1) inserts the new tuple (“x”, 4) into tuplespace, leaving only one tuple with the new incremented value in tuplespace as desired. The important observation is that variable x is a local variable of this process. If another process performs the same actions, it will get the current tuple that matches (“x”, ?x), but it will store its value in *its* local variable x. Now, both processes have some value for their *local* variable x.

Thus, while the concept of “last value written” is not well defined in other shared memory systems, it can be conservatively determined for the local variables of tuplespace processes by analyzing only the process in which the variable is declared. As such, it is not difficult to understand that standard data flow analysis *within a process* remains feasible in the context of tuplespace parallel programming.

We now examine whether the precision of the collected data flow information is compromised in any way by the tuplespace operations, and if so, whether it is possible to improve the precision. The answer, it turns out, is *Yes* to both queries.

As mentioned earlier, most Linda compilers map the tuplespace operation into a procedure call of the native language. The fields of the tuplespace operation are similarly handled as parameters to a procedure. For example, the C-Linda program statement IN(“num”, ?num) would be mapped into something resembling the following native C program statement `__l0_279sXb(&(num))`, which is a call to a runtime library function that implements this IN operation. Code is not generated for the string constant because the partitioning phase of the Linda compiler removes fields that are constant in all operations of a partition.

Thus, the procedure call `__l0_279sXb(&(num))` is passed to a standard global data flow analysis. More-

over, the compiler does not consistently select the same procedure name as its mapping target. Indeed, a second compilation of the *same* C-Linda IN operation yielded a call to `_1o_GwjMCb`. Conservative data flow analysis of this C procedure call treats `num` as being *ambiguously* defined by this statement, meaning analysis cannot be certain of whether the variable is defined within the called procedure. Interprocedural analysis will not help because the body of the function `_1o_279sXb` is not available for analysis.

However, the semantics of the original tuplespace operation clearly indicate that this definition is in fact unambiguous. We know that this IN operation will not return until it finds a matching tuple, and there will be some value of the same type as `num` in the second field, which will be assigned to the local variable `num`. In languages that use a pass-by-reference parameter mechanism (like Fortran), the same type of imprecision is present. Additionally, in pass-by-reference parameter passing, actual (non-formal) fields of tuplespace operations, which semantically are not definitions at all, are conservatively handled as ambiguous definitions. Ambiguous definitions affect the size of the GEN and KILL sets in such heavily-used data flow problems as reaching definitions and available expressions.

The precision of the standard data flow analyses for tuplespace programs can be improved by maintaining and using the high level information of tuplespace operations. In particular, a straightforward modification of the computation of GEN and KILL examines the statement to determine whether it is a tuplespace operation. If so, then formal fields become *unambiguous* definitions of the associated variable, and actual fields are not considered to be definitions at all. Both the intraprocedural and interprocedural analysis remain unchanged, except the changes in the GEN and KILL sets for tuplespace operations.

In terms of the reaching definitions data flow problem, for example, using this high-level semantic information can result in a reduction in the size of the set of propagated definitions. Propagated definitions can now be killed by the definitions of formal fields of tuplespace operations, whereas these formal fields would not be included in the KILL set when treated as ambiguous definitions. Recognizing that the actual fields of tuplespace operations are not definitions, but in fact uses of variables, aids in more precise available expressions sets as available expressions are not killed by these actual fields. This can lead to more opportunities for common subexpression elimination.

5 Interprocess Data Flow Analysis

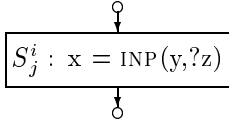
This section leaves the confines of performing data flow analysis on a single process. We present data flow equations to compute reaching definitions information for all the processes of a Linda parallel program. We show how to improve the precision of the interprocess analysis by identifying *unrealizable* tuplespace edges; that is, communications that can not occur at runtime.

In a sequential program, a definition of a variable is said to reach a program point p if there is a definition-free path in the control flow graph from that definition to p . Tuplespace programs typically consist of multiple processes with each process being a single procedure or encompassing multiple procedures. Since our analysis focuses on the handling of tuplespace operations, the analysis can be performed within an interprocedural setting, using well-known interprocedural analysis methods as the basis. For ease in presentation, we assume here that each process consists of a single procedure. Since variable names can be re-used in different processes as distinct variables, we make an assumption often utilized by interprocedural analysis that re-used names are changed to be distinct and that the process in which the name is used is recoverable from the name. Let \mathcal{V} be the set of all variable names in the program. A definition is then two pieces of information, the name (and hence the process) and the statement in that process defining the name. We subscript the statement number to the name as our notation (e.g., v_2). The set of all definitions in the parallel program is denoted as D^* .

The data flow information sets GEN , $KILL$, IN , and OUT for a given basic block are generally viewed as single sets containing individual definitions of various variables in the program. For data flow through tuplespace, it becomes convenient to have each of these sets be segmented such that there is a separate set of definitions for each variable. For example, for a statement S in a process with two variables x and y , $IN[S] = IN_x[S] + IN_y[S]$. The union of two sets is defined to be the union of each of the corresponding variable segments (i.e., $OUT[S] \cup OUT[T] = \forall v \in \mathcal{V}, OUT_v[S] \cup OUT_v[T]$). A statement S must also indicate its process and statement number. To denote statements, we use a three-tuple notation, S_j^i , where i indicates the process and j the statement within that process.

5.1 Reaching Definitions

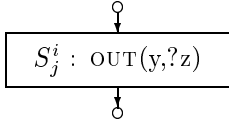
To characterize reaching definitions through tuplespace, we must consider three aspects: the direction of propagation of information, the transfer functions which characterize how information propagates



$$\begin{aligned}
 GEN_{v \neq x}[S_j^i] &= \emptyset & KILL_z[S_j^i] &= D^* & BIND_z[S_j^i] &= b_e(S_j^i, fieldnum(z)) \\
 GEN_x[S_j^i] &= x_j^i & KILL_x[S_j^i] &= D^* - x_j^i & BIND_{v \neq z}[S_j^i] &= \emptyset \\
 & & KILL_{v \neq z, v \neq x}[S_j^i] &= \emptyset & &
 \end{aligned}$$

$$IN_v[S_j^i] = \bigcup_{P \text{ a control predecessor of } S_j^i} OUT_v[P]$$

$$OUT_v[S_j^i] = GEN_v[S_j^i] \cup (IN_v[S_j^i] - KILL_v[S_j^i]) \cup BIND_v[S_j^i]$$



$$GEN_v[S_j^i] = KILL_v[S_j^i] = BIND_v[S_j^i] = \emptyset$$

$$IN_v[S_j^i] = \bigcup_{P \text{ a control predecessor of } S_j^i} OUT_v[P]$$

$$OUT_v[S_j^i] = GEN_v[S_j^i] \cup (IN_v[S_j^i] - KILL_v[S_j^i]) \cup BIND_v[S_j^i]$$

Figure 3: Equations for reaching definitions

through a node (i.e., statement), and the confluence operator which characterizes how information is handled at points in the program where paths are joining together. The reaching definitions problem is known to be a forward problem with union as the confluence operator; therefore, we define $TS_RDEF = \langle \mathcal{L}, \cup, \mathcal{F} \rangle$ as a data flow system for computing reaching definitions across tuplespace process boundaries. The lattice \mathcal{L} is defined to be the power set of the set of definitions, D^* . The confluence, or meet, operator is segmented set union as described above and is applied to the control flow edges in the Linda intermediate representation. The relation \leq defined over elements of \mathcal{L} is reverse set inclusion (i.e., $x \leq y \Rightarrow y \subseteq x$).

The function space \mathcal{F} consists of functions that model the transfer equations described here. For statements that are not tuplespace operations, the data flow equations behave identically to those of standard reaching definitions as there are no incoming tuplespace communication edges. The transfer equations for the INP and OUT tuplespace operations are shown in Figure 3. The equations of IN are the same as for INP without the effect for the assignment to x . The equations for RD and RDP follow the IN and INP, respectively. For compactness, figure 3 shows tuplespace operations with one actual and one formal field, but the equations hold for operations with multiple actual and/or formal fields (e.g., $IN(a, b, ?x, ?y)$).

We define a new variable-segmented set $BIND$ to account for definitions mapping from one process context to another. Computation of $BIND$ utilizes a mapping function similar to the b_e and $back-bind_{C_p}$

functions utilized by the interprocedural data flow in [7] and [25] respectively. Our b_e function takes a node representing a tuplespace communication and the position of the formal field and returns a set of definitions that is the union of the variable-segments for the associated field of all predecessor communication nodes (i.e., $b_e(n, f) = \bigcup OUT_v[m]$ for each node m such that $m \rightarrow n$ is a tuplespace communication edge, and $fieldnum(v) = f$). If there are no tuplespace communication edges entering a node n , then each variable-segment of the $BIND$ for n is the empty set (i.e., $\forall v \in \mathcal{V}, BIND(n, v) = \emptyset$).

The GEN data flow set contains definitions generated in the usual reaching definitions sense. In particular, definitions “generated” by formal fields in tuplespace operations are *not* included in the GEN . This is due to the fact that the GEN set is precomputed and remains constant throughout the data flow computation. In contrast, definitions from other processes “reaching” a tuplespace extraction operation may have to be propagated within their own process to reach the corresponding tuplespace generation operation. The $KILL$ set reflects the semantics of formal fields in the tuplespace operations. Specifically, formals in IN, RD, INP and RDP operations define variables; moreover, these definitions are unambiguous. Therefore, *all* definitions of that variable in the process are killed at the point of the tuplespace operation. This is why the formal field variable-segment of $KILL$ includes all the program definitions. The correct definitions coming from the tuplespace communication edge will be included by the effect of the $BIND$ set.

The *IN* set is the same as in traditional reaching definitions. Intuitively, definitions still reach statements in the usual way, that is, they are said to reach a statement if they reach any of its control predecessor statements, represented by the union of definitions leaving all control predecessor statements. *BIND* isolates definitions from the communication edges. The *OUT* set is defined to be $OUT[n] = GEN[n] \cup (IN[n] - KILL[n]) \cup BIND[n]$. For those variables that are not in a formal field, the *OUT* set is computed in the usual way because the variable-segment of *BIND* for these variables is the empty set. However, a formal field variable of an extraction operation (IN, INP, RD, RDP) will have a non-empty variable-segment in its *BIND* set (and its *KILL* set will be D^*). This allows *only* those definitions “being passed by” the formal’s corresponding field in a matching tuplespace generation operation (OUT, EVAL) to propagate. Because of the way tuplespace operations are connected by communication edges, this will result in a safe overestimation of the set of reaching definitions as definitions are propagated even if the communication edge is not realized during program execution, similar to definitions propagating along control flow edges that may not execute.

Modeling the resultant tuple of an EVAL operation proves more difficult as the communication edge relates the EVAL to a tuplespace extraction operation, but the definition of a formal field may be the return value of an invoked process which is not connected to the extraction operation. However, the formal field is *implicitly* connected to the spawned process since this process is named in the EVAL operation. The SUIF system makes obtaining the data flow information from this implicit connection straightforward.

To compute the reaching definitions using these equations, the standard $\mathcal{O}(N^2)$ algorithm is run to iterate over the program nodes, N , until a fixed point is reached. In [10], *TS_RDEF* is shown to be a monotone data flow system.

5.2 Tuplespace Edge Elimination

More precise interprocess data flow information is possible if the number of tuplespace edges connecting tuplespace generation operations (OUT, EVAL) to extraction operations (IN, INP, RD, RDP) can be reduced. This reduction can be achieved by propagating control flow information. In the example in Figure 2, consider the edge connecting the OUT operation in statement S_6^1 of process 1 to the IN operation in statement S_4^1 also in process 1. Notice that it is not possible to execute statement S_4^1 again if statement S_6^1 is being executed. That is, there is no control flow path from

S_6^1 to S_4^1 . Since it is not possible for a tuple generated by the OUT operation in S_6^1 to be consumed by the IN operation in S_4^1 , the tuplespace edge connecting these two statements can be removed. Removing the tuplespace edge reduces the set of definitions that reach S_4^1 making the data flow analysis more precise.

This notion of eliminating tuplespace edges can be generalized to include eliminating edges in which the source and sink are not both in the same process. In this more general case, we need to characterize the set of IN, INP, RD and RDP operations that are *not* reachable from a given point in a process. We define $NREACH(n)$ as the set of tuplespace extraction operations that node n cannot reach, and $n \notin NREACH(n)$. An edge can be eliminated without hurting the conservativeness of the interprocess data flow analysis if the sink of a tuplespace edge (e.g., the IN operation) is in the $NREACH$ set of the source (e.g., the OUT).

Figure 4 presents an algorithm for edge elimination by computing and using $NREACH$ information. The algorithm is based on a reachability problem described in [30]. Initialization consists of setting the

algorithm edgeElimination(G)

```

/* INPUT:  $G$  - a forest of control flow graphs,
representing the processes and augmented with TS
edges ( $O \rightarrow I$ ) connecting generation operations to
corresponding extraction operations as determined
by partitioning
*/

```

```

Initialize  $NREACH$  sets of tuplespace operations

```

```

Worklist initialized to  $\{O \rightarrow I : O \rightarrow I \text{ is a tuplespace edge}\}$ 

```

```

while (Worklist  $\neq \emptyset$ ) {
  remove  $O \rightarrow I$  from Worklist
  if ( $I \in NREACH(O)$ )
    remove  $O \rightarrow I$  from  $G$ 

```

$$NREACH(I) = \bigcap_{O' \rightarrow I} NREACH(O')$$

```

for each  $O' \in REACH(I)$  {
   $NEW = NREACH(O') \cup NREACH(I)$ 
  if ( $NEW \neq NREACH(O')$ ) {
     $NREACH(O') = NEW$ 
    for all edges  $O' \rightarrow I'$ 
      Worklist = Worklist  $\cup O' \rightarrow I'$ 
  }
}

```

```

}
end algorithm

```

Figure 4: Tuplespace Edge Elimination Algorithm

$NREACH$ set to \emptyset for all tuplespace extraction opera-

tions. Tuplespace generation operations are initialized by the equation $NREACH(g) = U - REACH(g)$, where g is a tuplespace generation operation (i.e., OUT, EVAL), U is the set of all tuplespace extraction operations in the *same* process as g , and $REACH(g)$ is the standard reachability data flow information for this process. The worklist is initialized to contain all the tuplespace edges, which are subsequently taken off the worklist and processed. A tuplespace edge is removed from the graph if its sink is contained in the $NREACH$ set of its source; that is, the IN operation can not obtain a tuple generated by the OUT operation. The $NREACH$ set for the sink is then recomputed to be the intersection of all the $NREACH$ sets of OUT operations remaining connected to this IN by tuplespace edges. Then this information is propagated within the sink’s process to all the reachable tuplespace generation operations. If this propagation changes the $NREACH$ information for one of these OUT operations, all edges for which this OUT is a source are added to the worklist in order to propagate the new $NREACH$ information. The edge elimination algorithm executes in $\mathcal{O}(E_{TS})$ time (E_{TS} is the number of tuplespace communication edges).

Processes are initiated by the EVAL tuplespace operation, which is frequently contained in a loop to create distinct “slave” processes that utilize the same program text. This introduces a problem in the edge elimination algorithm. Consider again the example graph in Figure 2. Suppose that process 1 is created by an EVAL tuplespace operation that occurs in a loop. Therefore, the control flow graph shown for process 1 would also be representing the control flow graph for another process, say process 3. Now the tuplespace edge $S_6^1 \rightarrow S_4^1$ in the figure cannot be eliminated because it is possible for the OUT of the new process 3 to satisfy the IN of process 1. There are a couple of ways to construct the intermediate representation so that the algorithm still works. If the EVAL loop, which creates multiple processes from a single process definition, has statically known bounds that are not unreasonably high, then the process definition can be replicated that number of times. This idea of procedure cloning has proven beneficial in interprocedural compiler analysis [6]. If the bounds of the EVAL loop are not known at compile time or are unreasonably high for cloning, then we can apply a technique used in representing recursive data structures [26]. The idea is to clone a “summary” process definition that is treated differently because it is representing more than one process. The edge elimination algorithm requires a slight modification to ensure that an edge in which

the source or sink resides in a “summary CFG” is not removed.

6 Some Enabled Transformations

In a standard optimizing compiler, reaching definitions information is utilized by many common transformations. Here, we show how constant folding lends itself nicely to communicating processes. Moreover, the constant folding transformation need not be modified, since the parallel aspect of the transformation is already characterized by our interprocess reaching definitions information. In addition, a more precise solution for propagating constants can be computed through a straightforward modification of the interprocess reaching definitions data flow analysis presented in the previous section. The following example demonstrates how propagating constants through tuplespace processes can enable further sequential transformations.

<pre> PROCESS 1 OUT("num workers", 16); OUT("data size", 1024); </pre>	<pre> PROCESS 2 RD("num workers", ? W); RD("data size", ? N); for(i=0; i < N; i++) { /* body of loop 1 */ } for(i=0; i < N; i+=W) { /* body of loop 2 */ } </pre>
--	---

In the first loop, loop unrolling and other loop transformations may be beneficial if interprocess constant propagation can make the value of N known to standard analyses of process 2. Assuming the second loop were performing a type of cyclic array access, sequential loop transformations for improving the use of cache may be enabled for such array accesses if the stride is known.

However, an equally interesting possibility lies in how the tuplespace operations themselves may be improved by constant folding. Because the tuplespace partitioning makes use of constant values, it is possible for the propagation of constants to subdivide a partition, possibly reducing the runtime associative search cost. The partitioning information also allows the tuplespace partitions to be implemented with efficient data structures. The increase in available information due to constant propagation may be able to change the classification of a partition allowing the use of a more efficient data structure. For example, consider the following operations.

<pre> PROCESS 1 OUT("descrip", 0, descrip[0]); OUT("descrip", 1, descrip[1]); </pre>	<pre> PROCESS 2 RD("descrip", i, ? my_descrip); </pre>
<pre> PROCESS 3 RD("descrip", j, ? my_descrip); </pre>	

The initial partitioning would place all four of these Linda operations into the same tuplespace partition. The partition would be implemented with a hash table in which buckets would be distributed and the hash key would be the second field of the operations. If constant propagation discovered that the variable i in the operation of process 2 could be replaced by 0, and that j in process 3 could be replaced by 1, then the tuplespace partition itself could be improved. The first operation of process 1 and the process 2 operation could be subdivided from the second operation of process 1 and the process 3 operation. Additionally, each of the two smaller (sub)partitions could be implemented using a queue rather than a hash table, which reduces runtime processing of the operations.

7 Related Work

On the surface, interprocedural data flow analysis of sequential programs appears to be related to interprocess data flow analysis of parallel programs, in addition to its need to analyze data flow among procedures of a single process. There have been significant advances in exhaustive interprocedural analysis frameworks [28, 29, 27], non-exhaustive, demand-driven analyses [9], and interprocedural data flow analysis for specific problems [2, 14, 18]. We have seen that this body of interprocedural data flow techniques can be utilized for analyses *within* each tuplespace process.

Unfortunately, the differences between data flow between procedures and data flow across tuplespace processes appear to be too great to allow straightforward extension/modification of interprocedural techniques for inter-*process* analysis. Foremost of the differences is the notion of *valid* execution paths, or *realizable* paths. To obtain precise interprocedural data flow information, an analysis must only consider paths from procedure returns to the call site of the most recent call. However, tuplespace processes have no flow-of-control return to their creator; the processes simply cease to exist. A tuplespace process receives information from its creator only through formal parameters. Data flow information can propagate along interprocess tuplespace communication edges in addition to intraprocess control flow edges.

There has also been work on analyzing user-specified, shared memory parallelism expressed lexically, such as with `cobegin/coend` statements. Midkiff and Padua illustrated the problems of applying classic automatic parallelizing analysis to parallel shared memory programs [23]. A number of conventional parallelism and data flow techniques have been augmented to accommodate this style of parallelism [24, 5, 15]. More recently, Knoop, et.al.[17]

show an implementable method of solving unidirectional bitvector problems that are as efficient as the sequential methods. These problems cover a broad range of common and useful analyses including reaching definitions, availability, and liveness, as well as powerful optimizations like code motion and partial dead code elimination [16]. Moreover, their technique is shown to be optimal. However, the shared dataspace model of tuplespace is sufficiently different from the shared memory model because there are no shared “locations.”

In their non-concurrency analysis of parallel programs, Masticola and Ryder use a *sync graph* consisting of control flow and synchronization edges, and realize that elimination of synchronization edges improves analysis [21]. Because tuplespace is an intermediary between process communication statements (i.e., every OUT and IN causes communication through tuplespace), it is not necessarily true that the endpoints of a realizable tuplespace edge are concurrent. Thus, their framework is unsuitable for eliminating tuplespace edges; it may incorrectly remove communication edges that are possible.

The problem of monitoring distributed programs shares many of the difficulties of optimizing for the tuplespace model. In particular, Spezialetti and Gupta [30] developed compile-time analysis that establishes relationships between the SEND and RECEIVE communication statements of a distributed program in order to determine the program points at which monitoring instrumentation must be inserted. The analysis results are used to eliminate unnecessary instrumentation.

8 Conclusions and Future Work

The primary contribution of this paper is demonstrating the viability of existing intraprocess data flow analysis within the tuplespace parallel programming environment, and developing a technique for interprocess data flow analysis through tuplespace. Our experiences contrast that of others examining the problem of data flow analysis in parallel programming where processes share the same global address space, rather than the same data space. The intraprocess and interprocess reaching definitions data flow analyses described in this paper have been implemented within our Linda optimizing compiler. Currently, we are implementing the classic transformations that use reaching definitions as described in section 6, and plan to perform a thorough experimental study of the benefits of these optimizations on real Linda tuplespace programs. We are utilizing this analysis to help identify opportunities for communication optimizations

that target tuples used as messages between logically known processes; that is, where the physical location of the process destined to receive the message tuple is not known.

References

- [1] Robert D. Bjornson. *Linda on Distributed Memory Multiprocessors*. PhD thesis, Yale University, November 1992.
- [2] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 47–56, 1988. SIGPLAN Notices 23(7).
- [3] Nicholas Carriero and David Gelernter. A foundation for advanced compile-time analysis of Linda programs. In *Languages and Compilers for Parallel Computing*, pages 389–404. Springer-Verlag, 1992.
- [4] Nicholas John Carriero, Jr. *Implementation of Tuple Space Machines*. PhD thesis, Yale University, December 1987.
- [5] Jyh-Herng Chow and William Harrison. Compile time analysis of parallel programs that share memory. In *ACM SIGPLAN Symposium on Principles of Parallel Programming Languages*, January 1992.
- [6] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure cloning. In *The 1992 International Conference on Computer Languages*, pages 96–105, 1992.
- [7] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 57–66, June 1988.
- [8] Ashish Deshpande and Martin Schultz. Efficient parallel programming with Linda. In *Supercomputing '92 Proceedings*, pages 238–244, Minneapolis, Minnesota, November 1992.
- [9] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*, pages 37–48, 1995.
- [10] James B. Fenwick, Jr. *Compiler Analysis and Optimization of Linda Programs for Distributed-memory Systems*. PhD thesis, University of Delaware, expected completion in May 1998.
- [11] James B. Fenwick, Jr. and Lori L. Pollock. Issues and experiences in implementing a distributed tuplespace. *Software-Practice and Experience*. (accepted for publication February 1997).
- [12] James B. Fenwick, Jr. and Lori L. Pollock. Global compiler analysis for optimizing tuplespace communication on distributed systems. In *Eighth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, October 1996.
- [13] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
- [14] D. Grove and L. Torczon. Interprocedural constant propagation: A study of jump function implementation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 90–99, 1993. SIGPLAN Notices 28(6).
- [15] Dirk Grunwald and Harini Srinivasan. Data flow equations for explicitly parallel programs. In *The Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 159–168, May 1993. SIGPLAN Notices 28(7).
- [16] Jens Knoop. Partial dead code elimination for parallel programs. In *Proceedings of the Second International Euro-Par Conference*, 1996.
- [17] Jens Knoop, Bernhard Steffen, and Jurgen Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, May 1996.
- [18] William Landi and Barbara G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248. June 1992.
- [19] Kenneth Landry and John D. Arthur. Achieving asynchronous speedup while preserving synchronous semantics: An implementation of instructional footprinting in Linda. In *The 1994 International Conference on Computer Languages*, pages 55–63, May 1994.
- [20] Douglas Long and Lori Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of the Fourth Conference on Testing, Analysis, and Verification*, October 1991.
- [21] Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, May 1993. Appearing in SIGPLAN Notices 28(7):129–138.
- [22] Timothy G. Mattson. The efficiency of Linda for general purpose scientific programming. *Scientific Programming*, 3(1):61–71, 1994.
- [23] Samuel P. Midkiff and David A. Padua. Issues in the optimization of parallel programs. In *Proceedings of the International Conference on Parallel Programming*, volume II, pages 105–113, 1990.
- [24] Samuel P. Midkiff, David A. Padua, and Ron Cytron. Compiling programs with user parallelism. In David A. Padua David Gelernter, Alexandru Nicolau, editor, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, pages 402–422. MIT Press, 1990.
- [25] Hemant D. Pande, William A. Landi, and Barbara G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, 1994.
- [26] John Plevyak, Vijay Karamcheti, and Andrew A. Chien. Analysis of dynamic structures for efficient parallel execution. In *Languages and Compilers for Parallel Machines*, 1993.
- [27] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*, pages 49–61, 1995.
- [28] B. Rosen. Data flow analysis for procedural languages. *Journal of the ACM*, 26(2):322–344, 1979.

- [29] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice-Hall, 1981.
- [30] Madalene Spezialetti and Rajiv Gupta. Exploiting program semantics for efficient instrumentation of distributed event recognitions. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, 1994.
- [31] Harini Srinivasan, James Hook, and Michael Wolfe. Static single assignment for explicitly parallel programs. In *ACM SIGPLAN Symposium on Principles of Parallel Programming Languages*, January 1993.
- [32] Stanford SUIF Compiler Group. *The SUIF Parallelizing Compiler Guide*. Stanford University, 1994. Version 1.0.