STATIC OPTIMIZATION OF DISTRIBUTED TUPLESPACE MESSAGE COMMUNICATIONS

JAMES B. FENWICK, JR. LORI L. POLLOCK
Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716
(302) 831-1953 (302) 831-8458 (Fax)
{fenwick,pollock}@cis.udel.edu

ABSTRACT

Distributed memory parallel systems, such as workstation clusters, stubbornly remain in need of software systems that provide programmers with an effective but uncomplicated means of realizing improved application performance. Distributed shared memory systems are maturing to fill this need. Tuplespace is a structured distributed shared memory that embodies the generative communication model. This paper presents static analyses and transformations of tuplespace parallel programs that experimentally show significant average reductions of 33% in the latency of using a class of tuplespace communication called message tuples, which seek to achieve directed communication within a paradigm of uncoupled communication. Message tuples are identified statically, and an alternative runtime communication strategy implements the improved handling of these tuples during execution.

Key Words: Tuplespace, Communication analysis

1 INTRODUCTION

Parallel computing via network clusters offers a cost effective solution to obtaining scalable performance gains for large parallel programs. Moreover, current designs of parallel multiprocessor architectures are utilizing distributed memory organizations. Several options are currently available for programming these systems to exploit application parallelism: message passing, 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 (DSM) paradigm is receiving increased attention. The shared memory abstraction of DSM provides an easier transition from a sequential to a correct parallel program.

Gelernter introduced the tuplespace paradigm of parallel programming as an actualization of his generative communication model [1]. Explicitly created processes share a data space rather than sharing variables. Messages are not sent between processes, but are instead placed in the shared data space for other processes to access. To reinforce this differentiation, messages in this paradigm are called tuples, and the shared data space holding these tuples is called *tuplespace*. Tuplespace is an associative memory meaning that tuples do not have addresses but rather are referenced by their content. Therefore, tuplespace is a *structured* DSM that provides parallel programmers with an abstraction that hides the specific underlying mechanisms implementing process creation, communication, and synchronization.

Unfortunately, any abstraction of this kind necessarily introduces a trade-off for the application programmer between ease-of-use and control over performance. Indeed, implementation of tuplespace on a distributed memory architecture has raised concerns regarding efficiency and performance [2]. However, several researchers have demonstrated that distributed tuplespace implementations can be efficient [3, 4]. While the tuplespace paradigm can be efficient, there is still room for additional improvement. In particular, compiler analysis of tuplespace parallel programs can further increase efficiency [5, 6, 7].

This paper describes a new technique that can significantly improve the use of a large, frequently-used class of tuples. Specifically, a static analysis of tuplespace parallel programs identifies tuples that are directed to a specific process and transforms the program to utilize a new runtime communication strategy. The analysis and transformation are implemented within our Linda optimizing compiler, and Deli, our runtime tuplespace system [8], integrates the runtime modifications. The remainder of this paper summarizes the essential aspects of tuplespace programming, provides background on current static analysis of tuplespace, describes a class of tuples used as messages, presents static analysis to identify message tuples, introduces an alternative runtime communication strategy to improve the handling of message tuples, and describes a compiler transformation to take advantage of this alternative handling. Experimental evidence revealing significant improvements is also presented.

2 LINDA

The best known implementation of the generative communication model is Linda[1] [1, 9]. Our work is based on Linda although it applies more generally to any generative communication model implementation. Linda is a coordination language consisting of a small number of primitives that are added into existing sequential languages to create a parallel dialect of that existing language (e.g., C-Linda). These operations perform the communication and synchronization necessary for parallel programming. Communication between processes is achieved through tuples in the associative, global memory called tuplespace. The tuples in tuplespace are manipulated by these six operations: OUT, EVAL, IN, RD, INP, RDP. 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.

An OUT operation is a non-blocking operation that asynchronously inserts a tuple into the associative, global memory known as tuplespace. The IN primitive is a blocking operation which synchronously extracts a tuple from tuplespace. Usually the IN operation has one or more fields that act as place holders and are indicated using the character "?". The values of the corresponding fields of a tuple in tuplespace are copied into the place holders of the IN operation. This is how data is passed between processes. The RD operation is another synchronous operation that acts like the IN, only it does not remove the tuple from tuplespace. The INP and RDP operations are predicate versions of their counterparts. Thus, they do not block if no matching tuple is present in tuplespace but rather return a false value. Lastly, the EVAL operation creates an *active* tuple in tuplespace, which means that some field(s) of the tuple is currently being evaluated and the tuple is unavailable for matching until this field has completed evaluation. At that time, the tuple becomes a *passive* tuple, like those created using the OUT operation. New processes are created to evaluate the fields of an EVAL operation. This is how the programmer explicitly creates parallelism. A Linda terminology convention specifies that OUT and EVAL operations create tuples, while IN, RD, INP, and RDP operations create *templates* that describe desired tuples.

Partitioning is a tuplespace improvement that groups Linda operations into sets such that all tuples produced by operations of a set will only match templates generated by operations of the same set [10]. Tuplespace partitioning is a function of the compiler or preprocessor. Partitioning significantly reduces the time required to match templates to tuples. A successful strategy to distribute tuplespace uses a static hash function to map partitions to nodes in the parallel machine [11]. These nodes are called *rendezvous* nodes

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

```
process node(int mypos) {
   /* "previous" processor sent message here */
   IN("message", mypos, ?data);

   /* send message to "next" processor */
   if (mypos < RINGSIZE-1)
     OUT("message", mypos+1, data);
}
```

Figure 1: PROGRAM PIECE USING MESSAGE TUPLES

because a partition's tuples and templates, which can be produced by processes on nodes throughout the machine, rendezvous at this one node.

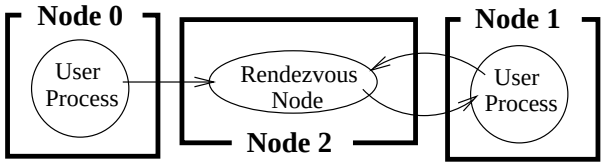
3 RELATED WORK

Bjornson describes some runtime techniques for improving performance of general tuplespace rendezvous communication in [11]. *Rendezvous reassignment* is a strategy whereby the runtime system keeps careful statistics about the communication patterns of tuples and reassigns rendezvous node responsibility based on those patterns. Our technique performs a similar kind of improvement, but with the detection occuring at compile time when there is useful source program information and no lost optimization opportunities due to statistics gathering and heuristic decision making at runtime. We envision a combination of static and dynamic reassignment detection as the best solution. The *randomized rendezvous node* technique also uses detailed runtime system statistics to determine that many processes are bottlenecked at a single rendezvous node. The runtime system then distributes rendezvous responsibility for this partition to a set of nodes. There is additional runtime communication required, but the method has proven beneficial.

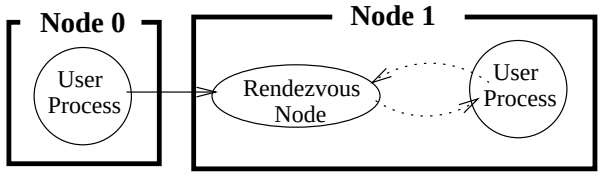
In addition to the dynamic techniques, Bjornson performs some optimizations based on limited compile time analysis [11]. *Tuple replication* broadcasts a tuple to all nodes so that requests may be found locally. A peephole compiler analysis decides if tuple replication is applicable. *Local data caching* reduces the number of times large tuples are transmitted by the network at the cost of more smaller transmissions. In general a tuple is transmitted twice, once to the rendezvous node and once to the data requester. Peephole compiler analysis identifies tuples with large data fields (e.g., arrays) and tags these fields so that they are *not* transmitted with the rest of the tuple, but cached on the local node. The process receiving the tuple uses the tag to request the large data fields directly from the producing node. *Inout collapse* is another peephole analysis that combines an IN operation and an OUT operation into a single operation. Performance is improved by reducing the number of communications and by the rendezvous node not having to deallocate and reallocate space for the tuple. The increment op-

eration is now done inside the rendezvous node rather than in the application program.

Landry and Arthur extended the idea of instruction footprinting to Linda operations [5] by dividing IN and RD operations into two sub-operations, an initialization (e.g., IN_{init}) and its receive (IN_{recv}). IN_{init} sends the template to the rendezvous node, and IN_{recv} gets the tuple data from the rendezvous node. Performance is improved by increasing the distance, or footprint, between an IN_{init} and its IN_{recv}. Thus, useful computation overlaps template and tuple communication.



(a) General: static rendezvous assignment does not consider target



(b) Improved: rendezvous node reassigned

Figure 2: INEFFICIENCY OF MESSAGE TUPLES

Previous work by Fenwick and Pollock[6] involves *global* compile time analysis to uncover common patterns of tuplespace programs. Identifying shared variable tuples enables optimizations such as Bjornson's inout collapse. In addition, shared variable tuples are often used as part of multi-partition distributed data structures. A common example is a distributed queue. Compiler analysis uses the shared variable tuple analysis and then sophisticated global data dependence analysis to relate operations using the shared variable tuple and the queue data tuples [7].

4 MESSAGE TUPLES

The generative communication embodied by tuplespace has two unique characteristics that distinguish it from other parallel systems: time and space uncoupling. Tuplespace processes are uncoupled in time because the tuple producer and tuple consumer do not have to co-exist for the communication to occur. Being uncoupled in space means that a tuple producer does not know the *location* of the consumer of the tuple. Orthogonally, the consumer does not know the location of the tuple producer. While many systems allow a message to be received from anyone, the sender of a message in a point-to-point communication must specify a receiving node in the distributed machine. Tuplespace achieves space uncoupling because all send and receive requests are routed to the rendezvous node.

While these properties of tuplespace provide parallel programmers with a unique flexibility, they can undermine performance when one wants to send a tuple to a specific process. For example, figure 1 shows a tuplespace program fragment where processors have been logically organized into a ring, and a node receives a communication from the "previous" processor and forwards it to "next" processor. In cases like this, the tuple is behaving like a *message*; that is, the tuple is intended to be received by a specific target process (hence, processor). We call this kind of tuple a *message tuple*.

Unfortunately, due to the space uncoupling property of tuplespace, there is no communication primitive that provides a directed communication, such as a message tuple, from one process to another. Since the rendezvous node is determined statically without considering such intended communication patterns, chances are high that the rendezvous node will *not* be the intended destination. In this general situation, three network accesses are required for a single communication: sending the data to the rendezvous node, sending a request for data to the rendezvous node, and receiving the data from the rendezvous node. If, by chance, the rendezvous node is the intended receiver's node, then only one network access is required—to send the data to the rendezvous node. The request and receive operations can, in this case, occur locally thereby decreasing latency. Figure 2 illustrates the general case and the improved case. This paper presents a compiler technique that forces the improved situation to occur. The communication latency is decreased because two of the tuplespace communications, shown as dashed lines in figure 2(b), no longer involve network access.

5 STATIC IDENTIFICATION

This section describes an advanced compiler technique to identify tuples used as messages.

5.1 Program Representation

Message tuple identification occurs within an optimizing compiler; thus, the analysis is performed on an intermediate representation of the tuplespace parallel program. The entire program can be viewed as a forest of process intermediate representations, where each process is a collection of procedures. The *main()* procedure, or *real_main()* in C-Linda parlance, is the initial entry point for the program. EVAL operations specify procedures that are entry points for other processes. The current version of our compiler builds an interprocedural flow graph similar to [12]. Each procedure is represented in the form of a control flow graph.

Interprocedural execution paths are not explicitly represented by edges in the IR, and neither are there edges from process invocation sites to process entry points.

Each process entry point is annotated with an estimate of the number of invocations of the process. For example, the processes shown in figure 3 are annotated to reflect that each is invoked only once. If an EVAL operation is contained in a loop whose bounds cannot be computed at compile time, the process entry points specified in the EVAL operation are annotated with a special value, ∞, to indicate an unknown number of multiple invocations.

Within each control flow graph, tuplespace operations are represented by separate nodes. Communications are represented by adding a directed tuplespace *communication* edge from tuple generating operation nodes to tuple extracting operation nodes of the same partition. These edges are distinguished from control flow edges. Figure 3 shows two processes, each consisting of a single procedure. The control flow graphs are connected only by these communication edges, shown as dotted lines.

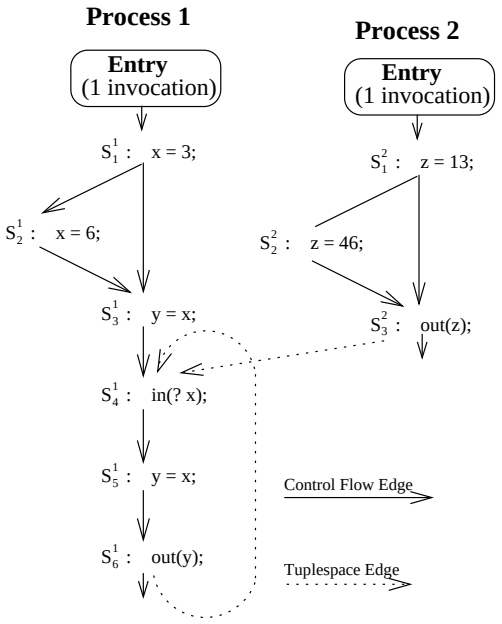


Figure 3: EXAMPLE PROGRAM REPRESENTATION

5.2 Identification of Message Tuples

The identification of a message tuple requires the identification of a tuple that is intended to be received by a single, specific target process. In our representation of a tuplespace program, this is characterized by a generative operation (i.e., OUT or EVAL) that has *all* of its communication successor nodes (i.e., IN, RD, INP, or RDP operations) residing in the same process, and this process must have an invocation multiplicity of one. Thus, a tuplespace partition with only a single IN, RD, INP, or RDP extraction operation that is contained within a process invoked no more than once trivially receives message tuples from each of the generative operations associated with this partition. While such message tuples do exist, it is more common that partitions contain more than one extraction operation, and these operations are located in different processes or multiple process invocations.

Fortunately, message tuples can still be discovered if *false* tuplespace communication edges can be eliminated. A false communication edge is one that was created during construction of the program representation, but is infeasible during execution due to the program control flow.

This elimination of tuplespace communication edges can be achieved by propagating control flow information. A simple example illustrating the idea of edge elimination is shown in figure 3. 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.

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. For each tuplespace operation n, we define $NREACH(n)$ to be the set of tuplespace extraction operations that n cannot reach. An edge can be eliminated if the target of the tuplespace edge (e.g., the IN operation) is in the NREACH set of the source (e.g., the OUT), meaning that there is no execution path in which the IN can consume the tuple generated by 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 [13]. Initialization consists of setting the $NREACH$ set to $\emptyset$ for all tuplespace extraction operations. 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. 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

```
algorithm edgeElimination(G)

/* INPUT: G - a forest of CFGs representing
   processes and augmented with communication
   edges O → I connecting generation operations
   to corresponding extraction operations */

   Initialize NREACH sets
   Worklist = {O → I : O → I is a comm. edge}

   while (Worklist ≠ ∅) {
     remove O → I from Worklist
     if (I ∈ NREACH(O))
       remove O → I from G

     NREACH(I) = ⋂_{O' → I} NREACH(O')

     for each O' ∈ REACH(I) {
       NEW = NREACH(O') ∪ NREACH(I)
       if (NEW ≠ NREACH(O') {
         NREACH(O') = NEW
         for all edges O' → I'
           Worklist = Worklist ∪ O' → I'
   } } }
end algorithm
```

Figure 4: EDGE ELIMINATION ALGORITHM

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. [14] has details regarding algorithm termination and single CFG representation of multiple process invocations.

After edge elimination is performed and message tuples are identified, the compiler annotates each process's program representation to indicate the tuplespace partitions of any message tuples that it could receive. It may be possible for edges to be eliminated so that a partition has message tuples received by more than one process. Since there can be only one rendezvous node, only one of the processes can be annotated. The compiler can make an informed decision about which process to select in this situation, for instance by using loop nesting depth to estimate the number of runtime tuples generated and consumed.

6 OPTIMIZATION

This section describes how message tuple annotations are utilized to automatically restructure the parallel program and realize reduced latencies when using message tuples. The first contribution is a program transformation that requests a new service from the tuplespace runtime system. The second contribution is the runtime system support enhancement.

6.1 Compiler Transformation

The decreased latency from identifying message tuples is achieved by ensuring that the rendezvous node is also the receiver of the message tuple. The compiler has already analyzed the program and identified the partitions of message tuples that each process could receive. The node executing a given process should be the rendezvous node for the partitions identified by the compiler in order to realize the improved performance. However, the mapping of processes to processor nodes is not known until runtime. Successful distributed tuplespace implementations execute the *main()* entry point on the host node, and an *eval server* is run on every other node to execute processes specified by EVAL operations [11]. The compiler replaces each source program EVAL operation with an OUT operation that generates a process description tuple, which describes the process specified in the EVAL. The eval servers, then, repeatedly request a process description tuple from tuplespace with an IN operation and execute the specified process. Figure 5(a) shows the default compiler output for an example source program tuplespace operation of EVAL(`foo()`). Lines 1 and 2 of figure 5(a) set up a structure containing the arguments to be passed to the new process. The second field of the process descriptor tuple of line 3 specifies the appropriate execution entry point to the eval server.

To convey rendezvous node responsibility information to the eval server receiving the process description tuple, the compiler performs an additional transformation when generating the process description tuples.The compiler checks to see if the process being de-

```
1: procArgs[0] = i;
2: procArgs[1] = NULL;
3: out("descrip",FOO,procArgs);
```

(a) Default transformation of EVAL(`foo(i)`) generates a process descriptor tuple that an eval server handles

```
1: procArgs[0] = i;
2: procArgs[1] = NULL;
3: rendezvous[0] = 6;
4: rendezvous[1] = NULL;
5: out("descrip",FOO,procArgs,rendezvous);
```

(b) New transformation of EVAL(`foo(i)`) encodes partitions that should rendezvous at the eval server node

Figure 5: COMPILER TRANSFORMATIONS

scribed has been determined to act as the rendezvous node for any message tuples. If so, the partitions of these message tuples require reassignment and are encoded in the process description. Figure 5(b) shows an example of this transformation in lines 3, 4, and 5; process *foo* receives a message tuple of the sixth tuplespace partition. The process descriptor tuple includes an extra field to indicate the rendezvous node reassignment to the eval server handling this process

descriptor. At runtime, this eval server dynamically forces the reassignment of rendezvous node responsibility as indicated by the compiler.

6.2 Runtime Support

In addition to the compiler analysis and transformations, the tuplespace runtime system must be enhanced to support the dynamic reassignment of a rendezvous node. This enhancement is implemented by two modifications. The first adjustment occurs within the eval server function. After receiving a process descriptor tuple, the eval server uses the new field (the fourth field of line 5 in figure 5(b)) to check if there is any partition that requires reassignment. If so, the eval server notifies the statically assigned rendezvous node of the reassignment; the statically assigned rendezvous node forwards any stored tuples and templates to the new rendezvous node. The second modification lazily handles requests to the statically assigned rendezvous node. Any process that remains unaware of the reassignment will continue to contact the statically assigned rendezvous node. Thus, this node lazily propagates the reassignment to processes as needed.

7 EXPERIMENTAL RESULTS

We have built an optimizing Linda compiler based on the SUIF compiler infrastructure [15], and a distributed tuplespace runtime system [8] that executes on a network of Sun 4 workstations connected by Ethernet. The compiler accepts C-Linda programs, represents the program in an intermediate form, performs communication optimization analyses and transformations, transforms Linda operations into procedure calls to a runtime library, and outputs C code which is then compiled and linked by a native compiler. In particular, the compiler implements the analyses and transformations presented in this paper to reduce the latency of using message tuples. The distributed tuplespace runtime system was modified to support dynamic rendezvous node reassignment.

Synthetic programs were carefully developed to measure message tuple latency. Each experiment consisted of running an unoptimized program and then an optimized version of the program. The experimental data showed an average reduction of 33% in the latency of using message tuples in a distributed tuplespace.

The runtime rendezvous reassignment imposes a onetime overhead. The greatest portion of this overhead is the forwarding of any stored tuples and templates from the old rendezvous node to the new one. This overhead is then amortized over the total number of actual, runtime occurrences of the message tuples. In fact, the compiler analysis could decide to *not* indicate rendezvous reassignment even in the presence of a message tuple if the analysis can determine that the number of message tuple uses does not offset the overhead; however, the current implementation of our analysis does not perform this additional analysis.

8 CONCLUSION

While a Linda programmer can sometimes identify the use of a message tuple, it is not possible to realize any improvement using the standard tuplespace operations. A Linda programmer informed of available runtime support for the optimization could conceivably make direct calls to the runtime library, but this is deemed unsatisfactory because the complexity of the programming increases dramatically. A possible compromise is to allow programmer directives, but this requires changes in the compiler to recognize and process the directives. Our analysis hides these concerns from the user, requires minor changes in the runtime system, and adds little to compiler analysis overhead.

References

[1] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[2] C. Davidson. Technical correspondence on *linda in contex*. *Communications of the ACM*, 32(10):1249–1252, Oct 1989.

[3] Ashish Deshpande and Martin Schultz. Efficient parallel programming with linda. In *Supercomputing '92 Proceedings*, pages 238–244, November 1992.

[4] Timothy G. Mattson. The efficiency of linda for general purpose scientific programming. *Scientific Programming*, 3(1):61–71, 1994.

[5] 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, Toulouse, France, May 1994.

[6] James B. Fenwick, Jr. and Lori L. Pollock. Global static analysis for optimizing shared tuple space communication on distributed memory systems. In *Proceedings of 8th IASTED International Conference on Parallel and Distributed Computing and Systems*, October 1996.

[7] James B. Fenwick, Jr. and Lori L. Pollock. Optimizing the use of distributed queues in tuplespace. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, volume I, pages 212–217, June 1997.

[8] James B. Fenwick, Jr. and Lori L. Pollock. Issues and experiences in implementing a distributed tuplespace. *Software–Practice and Experience*, 27(10):1199–1232, October 1997.

[9] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs, A First Course*. The MIT Press, 1992.

[10] Nicholas John Carriero, Jr. *Implementation of Tuple Space Machines*. PhD thesis, Yale University, December 1987.

[11] Robert D. Bjornson. *Linda on Distributed Memory Multiprocessors*. PhD thesis, Yale University, November 1992.

[12] 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*, pages 37–48, 1995.

[13] 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.

[14] James B. Fenwick, Jr. *Compiler Analysis and Optimization of Tuplespace Programs for Distributed-memory Systems*. PhD thesis, University of Delaware, expected in May 1998.

[15] Stanford SUIF Compiler Group. *The SUIF Parallelizing Compiler Guide*. Stanford University, 1994. Version 1.0.