Paper Submission to HPCN'99 by

James B. Fenwick, Jr, Computer Science, Appalachian State University, Boone, NC 18608
Lori L. Pollock, Computer and Information Sciences, University of Delaware, Newark, DE 19816
Corresponding Author: Lori L. Pollock pollock@cis.udel.edu
Presenting Author: Yet to be determined.

Tuple Counting Data Flow Analysis and its Use in Communication Optimization

James B. Fenwick, Jr.[1] and Lori L. Pollock[2]

[1] Computer Science, Appalachian State University, Boone, NC 28608
jbf@cs.appstate.edu
[2] Computer and Information Sciences, University of Delaware, Newark, DE 19716
pollock@cis.udel.edu

Abstract. Tuplespace provides parallel programmers with an abstraction that hides the specific underlying architecture, allowing the architecture to be any number of platforms ranging from shared or distributed memory to a cluster of workstations. Unfortunately, any abstraction of this kind necessarily introduces a trade-off for the application programmer between ease-of-use and control over performance. This paper presents a data flow analysis framework which plays a key role in identifying opportunities for communication optimization in tuplespace parallel programs. The enabled optimizations are particularly important for implementations on distributed memory multiprocessors and cluster environments where tuplespace acts as a structured distributed shared memory abstraction and communication overhead is high.

1 Introduction

Distributed memory parallel systems, including workstations clusters, remain in need of software systems that allow programmers to effectively gain performance from the parallel processing resource without undue programming effort. The distributed shared memory paradigm has received increased attention because it offers the programmer the ease-of-programming benefits of a shared memory abstraction despite a physically distributed memory. Tuplespace is a *structured* distributed shared memory paradigm, as it offers programmers a shared space of structures as opposed to a linear array of bytes, and each structure is an individual shared unit[10]. 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; hence, the shared data space holding these tuples is called *tuplespace*[10]. In short, tuplespace offers the simplicity of shared memory programming and the benefits of distributed memory architectures, without the false sharing and memory consistency concerns of unstructured distributed shared memory systems.

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 the tuplespace paradigm on a distributed

memory architecture has raised concerns regarding efficiency and performance. However, several researchers have demonstrated that distributed tuplespace implementations can be efficient [5, 13, 3]. These experimental studies considered a wide variety of real applications that encompassed a large scope of parallel algorithm classifications. 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 [12, 7, 8]. However, previous efforts in optimizing tuplespace parallel programs have concentrated on runtime optimizations, and compile-time optimizations performed with no global analysis of the program.

This paper presents a data flow analysis framework which plays a key role in identifying opportunities for communication optimization in tuplespace parallel programs. We have implemented the analysis and several code-improving transformations that use the analysis within our Linda optimizing compiler[6] and integrated the runtime modifications into Deli, our runtime tuplespace system[6]. In previous papers, we have described the details of the identification algorithms and necessary transformations to the runtime system for these transformations[8, 7]; this paper focuses on the details of the data flow analysis component and only briefly overviews the transformations to show its application. Another paper describes our work in enabling classical reaching definitions data flow analysis over tuplespace parallel programs[9]. In contrast, this paper presents a new data flow analysis framework to gather information for answering a very important question, *Can the number of tuples be estimated?*, for tuplespace optimization. After describing the basics of tuplespace programming and our intermediate program representation, we describe our tuple counting data flow analysis. Some common tuplespace usage patterns that can be identified with the use of this data flow information are briefly described along with descriptions of how they can be transformed to improve a runtime performance. Results from experimental evaluation of the usefulness of this analysis are presented along with conclusions and future work.

2 Tuplespace

Tuplespace is the shared data space central to the generative communication parallel programming model, most notably embodied by Linda. 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, interprocess 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.

Partitioning tuplespace is a function of the compiler or preprocessor; it 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. The partitioning is performed by considering the number and types of fields as well as the values of constant fields. 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 [2]. These nodes are called *rendezvous* nodes because a partition's tuples and templates, which can be produced by processes on nodes throughout the machine, rendezvous at this one node.

3 Program Representation

To retain necessary high level information, we perform our analyses on a high level intermediate program representation. We have extended the high level representation developed for the SUIF shared memory parallelizing compiler[14] to serve as our high level representation. Each tuplespace operation outwardly appears like a procedure call, but is further annotated with high-level information, including its tuplespace partition, an indication of whether each field is formal or actual, and other useful information.

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. Figure 1 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 Tuple Counting Data Flow Analysis

It is impossible at compile-time to know exactly how many tuples, the *tuple count*, may be present in a tuplespace partition at an arbitrary time during execution. A data flow analysis framework is presented that answers the question, *For each tuplespace partition,* ***may*** *there ever be more than one tuple in that partition?* This data flow analysis computes a conservative estimate that is at least as large as the true tuple count.

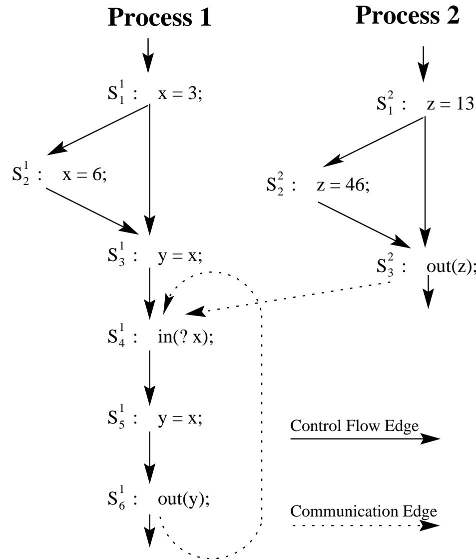


Fig. 1. Example program representation

The tuple counting data flow analysis approximates the existence of tuples in tuplespace at runtime by identifying them with their generative and extraction operations. That is, the analysis is interested in the tuples of a partition that are generated by OUT and EVAL operations and extracted by IN and INP operations.

The data flow framework $\mathrm{TUPLE_CNT} = \langle \mathcal{L}, \wedge, \mathcal{F} \rangle$ is defined to approximate the tuple count. The TUPLE_CNT framework is inspired by Hederman's framework that statically estimates reference counts for garbage collection [11]. Let $\mathcal{L} = \{\top, -1, 0, 1, \infty\}$ be the bounded set of potential tuple count values that a partition can have at any given time. The -1 element of $\mathcal{L}$ represents the fact that a tuplespace process can have at most one unsatisfied IN or INP operation affecting tuplespace, because these are blocking operations. The ∞ element represents the fact that the data flow analysis is not concerned with how much larger a tuple count becomes after it reaches greater than one. The top element, $\top$, is used for initialization. The binary meet operator, $\wedge$, is arithmetic

max, where $\top$ has the smallest arithmetic value in $\mathcal{L}$ and ∞ the largest. This choice for the meet operator reflects the fact that at a join point, there can be no more tuples in a given partition of tuplespace than there are along any one incoming path. Thus, the semilattice is ordered by arithmetic $\geq$; moreover, it is a bounded semilattice.

The TUPLE_CNT framework also consists of a monotone function space, $\mathcal{F}$, which reflects the transfer of information from the beginning to the end of any basic block. We first give a "basis" of functions that describes the transfer of information for single statements. The entire set $\mathcal{F}$ can then be constructed by composing the functions of this basis set.

1. The identity function $I(x) = x$ is in $\mathcal{F}$, and reflects a statement that does not affect the tuple count of a partition.
2. The entry node of each process requires a special function $Z(x) = 0$ to commence the propagation of conservative tuple count estimates. This version of the data flow analysis computes tuple count estimates for each process in isolation (incorporating the effects of other processes is described below). Thus, this function *initializes* the tuple count for each partition to 0.
3. In response to a generative tuplespace operation (i.e., OUT or EVAL), the tuple count estimate for a partition is *increased* by the following function.

$$G(x) = \begin{cases} 0 & \text{if } x = \top \\ x+1 & \text{if } x = -1 \text{ or } x = 0 \\ \infty & \text{if } x = 1 \text{ or } x = \infty \end{cases}$$

   The tuple count is increased from 1 to ∞, representing a value greater than one. The tuple count is conservatively increased from $\top$ to 0, since $\top$ represents an *unsure* value.
4. In response to an extraction tuplespace operation (i.e., IN or INP), the tuple count estimate for a partition is *decreased* by the following function.

$$E(x) = \begin{cases} -1 & \text{if } x = \top \text{ or } x = -1 \\ x-1 & \text{if } x = 0 \text{ or } x = 1 \\ \infty & \text{if } x = \infty \end{cases}$$

   The tuple count is *not* decreased from ∞, since the analysis is unsure of the actual value of ∞.

The TUPLE_CNT $= \langle \mathcal{L}, \wedge, \mathcal{F} \rangle$ data flow framework approximates the tuple count for a single tuplespace partition at each program point in a single tuplespace process. Computing the tuple count simultaneously for all tuplespace partitions is achieved by maintaining an array of functions for each basic block rather than a single function, and maintaining a corresponding array of tuple count values at each program point. The function in the ith position of the array computes the data flow information for the ith tuplespace partition, for i between 0 and the number of tuplespace partitions. Computing the data flow information

for a particular node involves the application of the appropriate function for each partition on the block's corresponding input data flow values.

Figure 2 provides an illustration. An example program and the associated control flow graph portion of the TS_IR are shown. The program utilizes three tuplespace partitions, P_1 for the "head" tuples and templates, P_2 for the "data" tuples and templates, and P_3 for the "worker" tuples. The flow graphs are annotated to indicate two items of information. First, to the left of each node is a function array for that node. For example, the function array $[G, I, I]$ for node M_4 indicates that when M_4 is visited during the data flow analysis, the G function will be applied to the current tuple count for P_1, and the I function will be applied to the current tuple count for the other two partitions. The second annotation is on the edges (or points) between nodes, and indicates the tuple count values computed on each pass of the TUPLE_CNT data flow analysis using the traditional iterative data flow algorithm. For example, the program point located between nodes M_4 and M_5 shows tuple counts for the initialization pass and four computation passes for each of the three partitions P_1, P_2, and P_3. In particular, for partition P_3, the tuple count was initialized to $\top$, was lowered to 0 after the first pass, lowered to 1 after the second iteration, and lowered to ∞ after the third pass where it remained for the fourth, and final, pass.

TUPLE_CNT is both *safe*, because it does not determine a tuple count of 1 for a partition whose true tuple count may be more than one, and *conservative*, because it may estimate a tuple count of ∞ when the partition's true count is equal to 1.

Because more than one process may access a single tuplespace partition, the TUPLE_CNT tuple counting framework is extended to accommodate the explicit parallelism of the tuplespace paradigm. The data flow information for each tuplespace partition in each process is summarized, and this summary information is used at the EVAL operations, which spawn processes. The summary information for a partition is computed as the *max* tuple count for that partition over all the points in the process. If the summary information for a partition is computed to be either $\top$ or -1, it is lowered to 0. This is done to ensure that negative tuple count summary information is not factored into the analysis, due to the uncertainty of when a parallel process actually performs its operations. Thus, the framework conservatively does not consider those processes that cumulatively reduce the tuple count. Let S_i represent the summary information for process i. Note that S_i can be an array to hold summary information for all tuplespace partitions. The summary information is initialized to $\top$. Two new functions, P and Q, are added to $\mathcal{F}$ and utilized at nodes containing an EVAL operation. These functions are defined as

$$P(x) = G(x) \wedge S_i$$
$$Q(x) = I(x) \wedge S_i,$$

and accommodate the parallel aspect of tuplespace processes by using the summary information, S_i.

Tuple counting is now performed by repeating the TUPLE_CNT analysis for each process followed by a process summarization step until there is no

```
main() {                                worker() {
  DATA_TYPE data;                         DATA_TYPE data;
  int i, j=0;                             int item;

  for (i=0; i < WORKERS; i++)             while (1) {
    EVAL("worker", worker());                 IN("head", ?item);
                                              OUT("head", item+1);
  OUT("head", 0);
  while (data = get_data())                   IN("data", item, ?data);
    OUT("data", j++, data);                   /* process(data) */
}                                       }   }
```

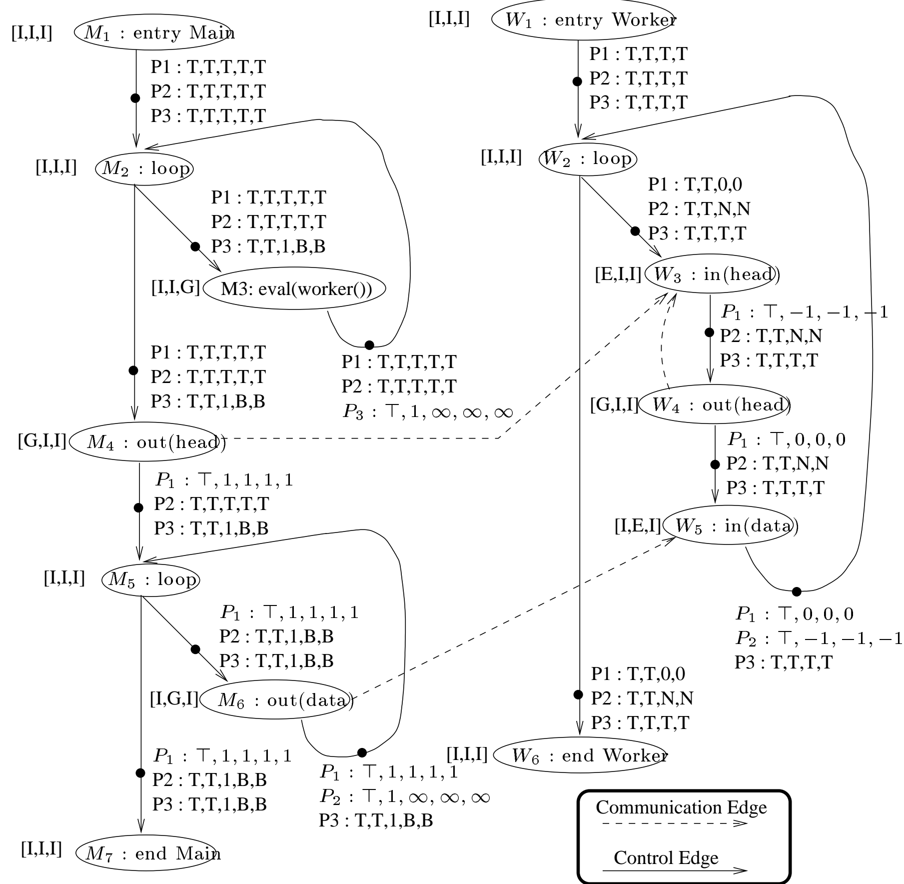


Fig. 2. Example of tuple counting data flow

change in data flow information at any program point in any process. Because the analysis of each process using TUPLE_CNT is guaranteed to terminate, and the summary information values are bounded by the semilattice, the analysis-summarization repetition also terminates. In the worst case, the summary information for each process assumes each tuple count value on a separate pass. Thus, this analysis is $\mathcal{O}(N^2 \cdot P^2)$, where N is the number of intermediate statements in the program and P is the number of tuplespace process definitions. However, since $P \ll N$ and typically P is a small number, the analysis is effectively $\mathcal{O}(N^2)$.

Returning to the example in figure 2, the function array for node M_3 is changed to $[Q, Q, P]$. The tuple count values shown are those after the first application of the TUPLE_CNT data flow analysis. This results in the summary information, $S_{main} = [1, \infty, \infty]$ and $S_{worker} = [0, 0, 0]$. The updated summary information S_{worker} is utilized by the Q and P functions at node M_3. After another repetition computing data flow within each process, the summary information does not change, and so tuple counting is complete.

In [6], the TUPLE_CNT data flow framework is shown to be monotone and distributive.

5 Some Enabled Transformations

Shared data is common in tuplespace parallel programs. In a shared memory context, a shared location contains only a single value, and processes must synchronize among themselves to ensure exclusive access to the shared location. In a tuplespace context, a partition that never contains more than a single tuple is said to contain a shared variable tuple. Removal of a shared variable tuple by a process implicitly synchronizes access to the tuple. Inserting the shared variable tuple into the empty partition makes the value available for other processes. Figure 3 depicts shared variable tuplespace operations.

```
IN("shared variable", ?value);
newvalue = compute(value);
OUT("shared variable", newvalue);
```

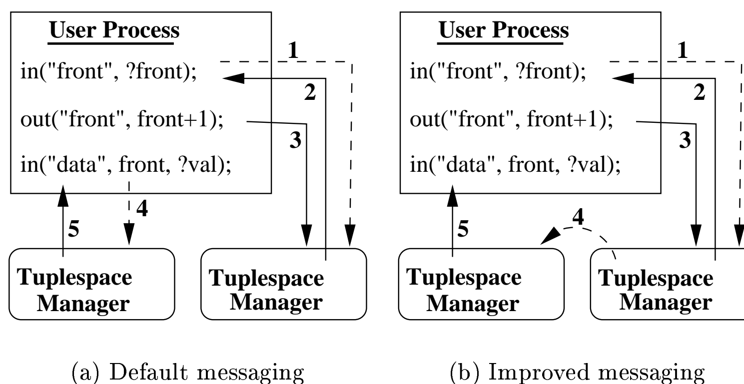
Fig. 3. Shared variable tuplespace operations

A shared variable tuple can be characterized as a set of tuples that satisfy the following conditions: (1) only one tuple of the partition is ever in tuplespace at a given time, (2) there is an actual field in every tuple that has a corresponding formal field in every template, and (3) the placeholder of this field in the template must be the source of a data dependence indicating that its value is used for computation. To detect as many shared variable tuples as possible, more than local pattern matching is needed.

The *in/out collapse* optimization [2] can be applied to many shared variable tuples. This transformation collapses an IN and a subsequent OUT into a single operation. This reduces underlying communication and the time spent allocating storage for the tuple. In addition, the basic synchronization tuple is a specialization of the shared variable tuple and can be optimized by identifying the synchronization and replacing it with a more efficient method native to the host architecture.

In a comparative study of several parallel programming languages, Bal observed that the concept of distributed data structures is an important contribution of the tuplespace communication model [1]. Distributing a queue essentially involves distributing the individual data elements across the memories of the cluster. A distributed queue used by a group of processes requires additional shared variable tuples to coordinate access to the front and rear of the queue.

Figure 4 shows operations used to remove an item from a queue. The "front" shared variable tuple coordinates removing data from the distributed queue among multiple processes, but at the cost of efficiency. As figure 4(a) shows, five network accesses may be necessary. Figure 4(b) illustrates an improvement to this default tuplespace handling of distributed queues, which Wilson terms *triangular messaging* [15]. Triangular messaging does not eliminate any of the required messages, but rather changes when and who initiates messages. At best, the triangular messaging scheme results in the user process experiencing no delay at the second IN operation because the tuplespace manager has already sent the tuple.



(a) Default messaging (b) Improved messaging

Fig. 4. Messages required to access a distributed queue.

In [8], Fenwick and Pollock present a compiler analysis to detect and transform the set of tuplespace operations acting on a distributed queue. The analysis links the value of the shared variable tuple to the IN operation that uses this

value as a position in the queue. After detecting distributed queue operations, the compiler transforms the program so that triangular messaging is performed during program execution. This involves augmenting the shared variable tuple request with information so that the tuplespace runtime system can send the template for the queue data item, and ensuring that the user process does not send this template. In addition, triangular messaging requires that the runtime system support the sending of a template on behalf of a user process.

6 Evaluation

The transformations that depend on the tuple counting data flow analysis were evaluated in terms of how often the opportunity for the targeted optimization occurs in real programs and how much performance gain can be achieved when the optimization is applied. The results are based on analysis of a set of sixteen application benchmarks including both synthetic and real codes gathered from a variety of sources[6]. The benchmark suite included small to medium-sized codes including several matrix computations, dining philosophers coordination code, database search, and molecular dynamics simulations.

Nearly one third of all partitions were used for shared variable tuples that qualified for the IN/OUT collapse transformation. The Deli implementation of this transformation yielded an average 25% decrease in the latency of using a shared variable tuple. About 37% of the programs used distributed queues. Experimental studies of applying the triangular messaging optimization for distributed queue operations showed an average decrease of 18% in the latency associated with using a distributed queue. All of the analyses involved in the identification of these transformations required either linear or quadratic time in the number of tuplespace operations or statements in the program in the worst case.

7 Conclusions

The primary contribution of this paper is a data flow analysis framework that statically estimates the number of tuples in tuplespace at runtime adequately enough to detect several communication optimizations. In our experiments, all of the tuplespace usage patterns which could be detected with the tuple count data flow analysis were fairly prevalent in the benchmark suite, resulted in significant reduction in communication latency, and did not require impractical compiler overhead.

References

1. Henri E. Bal. A comparative study of five parallel programming languages. In *Distributed Open Systems*, pages 134–151. IEEE, 1994.
2. Robert D. Bjornson. *Linda on Distributed Memory Multiprocessors*. PhD thesis, Yale University, November 1992.

3. Nicholas Carriero and David Gelernter. Learning from our successes. In Janusz S. Kowalik and Lucio Grandinetti, editors, *Software for Parallel Computation*, pages 37–45. Springer-Verlag, 1992.
4. Nicholas John Carriero, Jr. *Implementation of Tuple Space Machines*. PhD thesis, Yale University, December 1987.
5. Ashish Deshpande and Martin Schultz. Efficient parallel programming with Linda. In *Supercomputing '92 Proceedings*, pages 238–244, Nov 1992.
6. James B. Fenwick, Jr. *Compiler Analysis and Optimization of Tuplespace Programs for Distributed-memory Systems*. PhD thesis, University of Delaware, August 1998.
7. 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.
8. 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.
9. James B. Fenwick, Jr. and Lori L. Pollock. Data flow analysis across tuplespace process boundaries. In *Proceedings of the International Conference on Computer Languages*, Chicago, IL, May 1998.
10. David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
11. Lucy Hederman. Compile time garbage collection using reference count analysis. Master's thesis, Rice University, August 1988. Technical Report COMP TR88-75.
12. Kenneth Landry and James 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.
13. Timothy G. Mattson. The efficiency of Linda for general purpose scientific programming. *Scientific Programming*, 3(1):61–71, 1994.
14. Stanford SUIF Compiler Group. *The SUIF Parallelizing Compiler Guide*. Stanford University, 1994. Version 1.0.
15. Gregory V. Wilson. *Practical Parallel Programming*. The MIT Press, 1995.