Optimising the Linda in primitive: Understanding tuple-space run-times

Antony Rowstron
Microsoft Research
1 Guildhall Street
Cambridge, CB2 3NH, UK
antr@microsoft.com

ABSTRACT

In this paper we examine tuple space systems from a distributed viewpoint. We show that current implementations are pessimistic about the timing of removal of tuples from a tuple space when an `in` is performed; this leads to agents having to unnecessarily block and to lowering systems performance. After providing evidence of the problem by examining distributed execution traces we then describe an implementation strategy that is highly efficient and is more optimistic about tuple removal. We discuss also the generalisation of the approach to support other primitives, which have been proposed as additions to Linda, such as the `collect` and `copy-collect` primitives.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures, Patterns*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.5.5 [**Computer System Implementation**]: Servers

General Terms

Performance, Design, Languages

1. INTRODUCTION

There is currently a resurgence in interest in tuple space based co-ordination languages, as in Linda [3]. Examples of the new wave of languages are WCL [11], PageSpace [7], TuCSoN [9], Jada [6], TSpaces [16], KLAIM [8], Lime [10] and JavaSpaces [15]. A good review of the current trends is presented in Ciancarini et al. [5].

Like many implementations, we are interested in the development of centralised open servers to support large-scale enterprise wide tuple space usage by distributed agents. It was whilst working on optimisations for centralised servers we began to question the traditional semantics of the Linda `in` primitive. In this paper we present an interesting interpretation of the semantics of the Linda `in` primitive, or to generalise, of any blocking primitive that destructively removes tuples from a tuple space.

Throughout the paper, we will refer to the components of the system that communicate as agents, although this term is used in its loosest possible definition, and therefore an agent could either be a process, a "traditional" agent, or a program. In addition, throughout most of the paper we will just use the three standard Linda primitives:

out(tuple) Insert a tuple into a tuple space.

in(template) If a tuple exists that matches the template then remove the tuple and return it to the agent. If no matching tuple is available then the primitive blocks until a matching tuple is available.

rd(template) If a tuple exists that matches the template then return a copy of the tuple to the agent. If there is no matching tuple then the primitive blocks until a matching tuple is available.

The (informal) semantics of the `in` primitive leads implementers to remove the tuple that is returned to the agent from the tuple space as soon as the `in` primitive is completed. Our claim is that a tuple that has been destructively removed using an `in` does not actually have to be removed from the tuple space but it has to be made "partially visible". By partially visible we mean that it can be used as a valid result for a subset of the access primitives. When a set of further conditions is met the tuple then becomes invisible to all agents and has to be discarded.

In Linda programs, it is common to store data structures in the tuple spaces. This means that when parts of the data structure being held in the tuple space are being updated, tuples are removed from the tuple space, updated by a client and then re-inserted.

For example, consider a list stored in a tuple space, where the items of the list are stored as tuples. Each tuple has a unique number as the first field which represents its position in the list. In the tuple space there is a single tuple that contains a shared counter. In order to add an element to the list, the shared counter is removed using an `in` and the value of counter is incremented and the tuple re-inserted, and then a new tuple is inserted containing the number of the counter and the data.

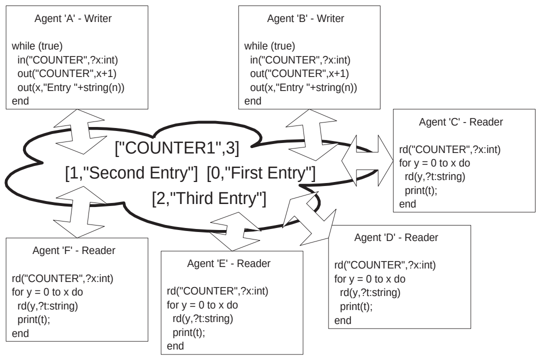
This is a common operation and there have been proposals for the addition of new primitives to help perform the update of the shared counter, (see e.g. Eilean [2]) and when using compile time analysis to convert the `in` followed by the `out` into a single operation [4]. These proposals were made because once the shared counter is removed, anyone else attempting to *read* the counter could not. Therefore, even if they wished to read the elements in the list they had to wait until the counter tuple was reinserted. This tuple is always acting as a bottleneck, which degrades performance. The use of compile time analysis to transform the two operations into a single operation relies on complex analysis, and many cases cannot be captured. The addition of new primitives at first appears attractive; a primitive that removes a tuple of a certain pattern and then inserts a new tuple of a defined pattern. However, specifying the contents of the new tuple in a generic way is difficult. Most make it a restriction that counter tuples be always of the same form (e.g. they have the counter in the same position within the tuple), and there are restrictions on the types that can be used for the counter [2]. Given that Linda is computation language independent it is difficult to see it being possible to create a primitive that can provide the functionality to deal with arbitrary tuples. It should also be noted that these new primitives would be there to provide support for a very specific case. Therefore, these types of primitives have never been widely adopted.

Agents using tuple space access primitives should only block if the required tuple is not available when the agent requests it. However, within any practical Linda system an agent could block even if the required tuples were available because of the overheads associated with finding the matching tuple. The most noticeable delay is due to network latency. However, for systems supporting tuple spaces over local area networks (LANs) network latency is relatively small, so the time taken to remove a tuple, update it, and reinsert it is relatively small. However, in a wide area network (WANs) the latency will be relatively greater and the time taken to remove, update and reinsert a tuple is larger. This means maximising the time a tuple is present within a tuple space provides better concurrency. (For more information on the costs of performing tuple space accesses see Rowstron et al. [13]).

All this means that if we can somehow leave a tuple visible after it should have been removed, and not alter the semantics of Linda, then we can potentially overcome some of the time cost of moving data out of the tuple space, across a network to an agent and then back. By doing this we can increase the level of concurrency within the system by reducing the effect of the tuple space acting as a bottleneck. This has increased concurrency because agents that would have blocked accessing a tuple space do not. Figure 1 shows an example, where there are multiple readers and multiple writers for a list stored in a tuple space; in the figure the cloud represents the tuple space. Using a traditional run-time system whenever the tuple [*"COUNTER",int*] is removed by agents A or B if any of the readers C, D, E, F is started they will block when accessing that tuple. In a run-time system using the technique described in this paper they will not block, thereby increasing concurrency.

Another advantage of not blocking the primitive is that you do not need to deal with the overhead of blocking the primitive. This increases computational load in the server and drops the performance, the number of operations per second that it can perform.

In Section 2, we informally show, using histories why the tuple can remain visible to some tuple space primitives. Sec-



Agent 'A' - Writer
```
while (true)
  in("COUNTER",?x:int)
  out("COUNTER",x+1)
  out(x,"Entry "+string(n))
end
```

Agent 'B' - Writer
```
while (true)
  in("COUNTER",?x:int)
  out("COUNTER",x+1)
  out(x,"Entry "+string(n))
end
```

Agent 'C' - Reader
```
rd("COUNTER",?x:int)
for y = 0 to x do
  rd(y,?t:string)
  print(t);
end
```

Agent 'D' - Reader
```
rd("COUNTER",?x:int)
for y = 0 to x do
  rd(y,?t:string)
  print(t);
end
```

Agent 'E' - Reader
```
rd("COUNTER",?x:int)
for y = 0 to x do
  rd(y,?t:string)
  print(t);
end
```

Agent 'F' - Reader
```
rd("COUNTER",?x:int)
for y = 0 to x do
  rd(y,?t:string)
  print(t);
end
```

Figure 1: Example of multiple reader, multiple writer to a list data structure.

tion 4 presents a set of rules that describe when a tuple is partially visible. Section 4 describes how a prototype implementation supports this optimisation. In the final section, we expand our view to consider other primitives, and the properties a set of primitives must have in order for the optimisation discussed in this paper to work.

2. HISTORIES

Sequences of primitive traces representing histories (traces) of tuple space access can be constructed according to a global observation, a tuple space observation or an agent observation. The different observations are shown in Figure 2, where the points label *observation* A_n create agent traces, the point label *observation* B creates global traces, and points labelled *observation* C_n create tuple space based traces. It should be noted that the "solid line" show the flow of primitives when tuple space based or agent based traces are being created, and the "dashed line" show the flow of primitives when global traces are being created. For a global observation the stream of all primitives is observed, for tuple space observation the stream of primitives to and from a particular tuple space is observed, and for agent observation the stream of all primitives to and from a particular agent is observed. It is assumed that there are no hidden communication channels between the two agents; the only way the two agents can communicate is via a tuple space.

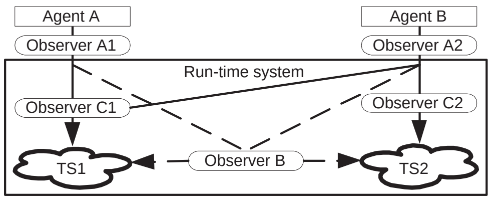


Figure 2: The different observation points in a tuple space system.

As Figure 2 implies if *observer B* is being used (global trace) then all tuple space accesses are sequential. In this case, *observer B* must decide how to convert concurrent tuple space requests into a sequential stream. If *observers C* are used then the primitives to each tuple space are sequential. However, different tuple spaces can be accessed concurrently. Within this paper, we assume that a primitive only appears in the trace when the primitive is completed. Given the basic informal semantics of Linda (and assuming all agents terminate) it is possible to create a finite set of all possible traces for a set of agents. However, both these traces enforce a sequential ordering on the primitives. In reality, the primitives can occur in parallel and indeed many LAN based implementations support parallel access to a single tuple space. The *observers A* can capture this, by observing the stream of operations into and out of single agents. Before considering these traces let us consider the global trace (*observer B*) and tuple space trace (*observer C*) using an example. In the example, let us consider two very simple agents that interact through a single tuple space, and their actions are represented by

	Agent A		Agent B
A_1	out(a)	B_1	in(a)
A_2	rd(a)	B_2	out(b)
A_3	rd(b)	B_3	out(a)

The Petri Net and case graph for these two agents can be seen in Figure 3. Initially, ignore the dotted links in the figure, and this Petri Net and case graph are created according to the semantics for the primitives as given in the introduction. In a Petri Net the circles represent places, and the squares represent transitions. A transition can fire only when all the places that are preconditions for that transition contain tokens. When a transition fires it consumes the tokens in its preconditions and places a token in each of the output places that are linked to it by arcs.

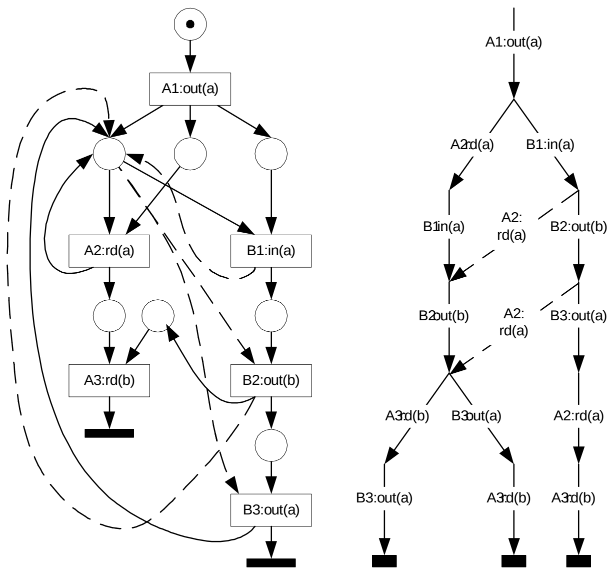


Figure 3: A Petri Net and case graph for Agents A and B.

In Figure 3 the token starts in the initial place, and the only transition that can fire is `A1:out(a)`. When this fires, a token is placed in the three output places connected to the transition. This means that either the transitions `A2:rd(a)` or `B1:in(a)` can fire. If `B1:in(a)` fires then the other can not fire, because the token is removed from one of its preconditions. This token is replaced when the transition `B3:out(a)` is fired. If `A2:rd(a)` fires, then the precondition tokens are consumed, but the transition is linked to one of its own preconditions. So a token is reinserted in that place. However, the same rule cannot re-fire because the other precondition no longer has a token in it. This means that the transition `B1:in(a)` is the only one that can fire, as it is the only transition that has all its' precondition places filled with a token. The case graph shown in the same figure, shows the different ordering of the transition firings that are possible.

In Figure 3 the dotted arcs represent the optimisation that we are proposing. We allow the transition `A2:rd(a)` to fire after the transition `B1:in(a)` fires or after the transition `B2:out(b)` fires. This means that the manipulation of a tuple has been suspended in the middle of the operation; agent B has performed the `in` operation and has received the tuple and can continue, but the tuple is not actually removed whilst Agent A cannot know that Agent B has received the tuple. This only occurs when there is a synchronisation between the two agents, which happens using the tuple `b`. From the global perspective, this appears to be incorrect; it allows the reading of a tuple that should have been removed. However, when a programmer is writing a program they will assume that this can happen because of the non-deterministic nature of Linda – if two agents perform an `in` and a `rd` concurrently there is no way of excluding that the `rd` will see the tuple and it is this property we are exploiting. By looking at the agent observer traces, it is easier to see why this is valid. The agent traces for Agent A will be: $\{A_1, A_2, [B_1, B_2]\ A_3\}$, and $\{A_1, [B_1, B_2, B_3]\ A_2, [B_1, B_2]\ A_3\}$, and the agent trace for Agent B it will be: $\{[A_1]\ B_1, B_2, B_3\}$.

The trace executions are relative to a specific agent and are composed of the actions performed by an agent. The trace is augmented to show the actions that need to have been completed by other agents in order for the agents' current action to complete, and these appear in the [ ] before the action entry in the trace. It should be noted that an agent does not know *when* the other agents' primitives were performed in relation to the primitive it has just performed, but it knows that primitives must have been performed and completed when the current primitive completes. It is assumed that at any point in the trace an agent can deduce which operations it has performed (so this information is omitted). At any point in the trace the union of the operations appearing in [ ] before that point in the trace represents what an agent can deduce about what other agents have done. If there are more than two agents, then the information about what other agents have completed (e.g. The [ ] entries) can contain entries for each agent.

We believe that this representation is closer to the model that the programmer has when working out the co-ordination patterns of a program using tuple spaces. This trace allows us to consider exactly what an agent (or correctly the author of an agent) can *assume* has occurred up to any stage. Now let us concentrate on the operations numbered A_2 and B_1. These are an `rd` and `in` operation on the same tuple, respectively. What is interesting is that Agent B does not know whether the operation A_2 is ever performed (the two agents never exchange tuples after B_1 and therefore agent B can not know what has or has not been executed). Also Agent A only knows that operation B_1 has been performed only when it *observes* that B_2 or B_3 has occurred. This is either when A_2 completes *or* when A_3 completes, depending on which trace is being generated. However, the programmer of the Agent A cannot assume which of the traces has occurred so must write the code in such a way to assume that B_1 has definitely occurred only when A_3 completes. This means that for the programmer of Agent A the operation A_2 is independent of operation B_1. When A_2 is performed Agent A has to assume that it *he does not know* whether the tuple has been removed. Therefore, even if the tuple was destructively removed they have to assume that this has not happened. Programmers are quite used to this as part of the asynchronous and non-deterministic behaviour of tuple space based co-ordination. Using this observation we then say it is quite acceptable for a run-time system to give the same tuple to both A_2 and B_1 regardless of whether A_2 or B_1 is serviced first provided that B_3 has not been performed. Traditionally, one would say that if B_1 has been serviced then A_2 must block until B_3 is performed. However, it should be noted once the `in` has been performed the tuple must become read-only as there can only ever be one copy of a tuple destructively removed from a tuple space. If this were not the case, we would end up with potentially multiple copies of the same tuple.

3. WHEN SHOULD A TUPLE DISAPPEAR?

3.1 Linda primitives

Although we have shown why the tuple can reside in a tuple space after it has been destructively removed, it is impractical in an efficient implementation to pass the agent traces around with the tuples (they could become very large!). However, by generalising the principle it is possible to create a simple set of rules that can be easily implemented, with little overhead.

What the traces show is that an agent can read a tuple that has been removed, provided that agent has no way of knowing that the other agent has removed the tuple. Therefore, when a tuple is matched by an `in` primitive it will remain in the tuple space, but:

i. It can not be returned as a result of another `in`.

ii. The agent which performed the `in` that matched the tuple cannot see the tuple anymore.

iii. When the agent which performed the `in` on the tuple inserts any other tuple or terminates the tuple must be removed[1].

If these rules are followed, the traces describing the agents activity remain the same and the semantics of the access primitives are preserved.

It is rule three above that enables one not to keep information about the primitives performed by other agents ([ ] in the traces). The traces contained the [ ] information to describe if an agent had *directly* or *indirectly* synchronized with another agent, and therefore, whether we could not use a tuple the other agent had consumed. By generalising the rule to say that whenever a tuple is inserted the tuples which the agent has consumed are no longer available as results, we make the use of this technique in implementations feasible, and this will be discussed in more detail in Section 4.

It should be noted there is no defined relationship between the tuple removed by the `in` and the tuple inserted by the next `out`. Indeed, multiple tuples could be destructively read before an `out` is performed and they would all remain visible until this `out` is performed.

[1]Termination, agent spawning and creation are examples of potential hidden and these must cause the tuple to be removed.

We have considered this so far in the restrictive case where there are only three primitives. Most modern tuple space based co-ordination languages have many more tuple space access primitives. In the next section we generalise the work to provide support for different types of primitives.

3.2 Extensions of Linda

Many of the modern implementations use a form of transactions to provide fault tolerance (although better alternatives exist using mobile code [12]). The approach to how these are implemented varies, but essentially, tuples that are destructively removed within a transaction are cached locally and tuples inserted within a transaction are also cached and not inserted. If the transaction aborts removed tuples are reinserted and inserted tuples are discarded. If the transaction completes the inserted tuples are actually placed in to the tuple space and the cached read tuples are discarded. The optimisation described here works with transactions. Inserted tuples are considered inserted at the end of the transaction (when they become accessible to other agents) and the destructively read tuples are partially visible until this point. The rules as outlined in the previous section apply.

The introduction of other primitives is commonplace, and it is important that any realistic optimisation should work with the current generation of tuple space languages. Each language has its own particular set of tuple space access primitives but it is possible to create a generic set of rules that potentially cover all sets of possible tuple space access primitives. In order to create them we need to consider the information that these primitives add. As an example, let us consider the `inp` and `rdp` primitives, although not necessarily widely supported they represent a different class of access primitive (non-blocking). Consider the two agents, sharing a tuple space with no other agents able to access that tuple space:

Agent D	Agent E
while (rdp(a));	in(a)
out(b)	in(b)

In this case Agent D uses the `rdp` primitive to poll the tuple `a`. Whilst the tuple exists the tuple `b` will not be produced. With the rules outlined in the previous section Agents D and E would never terminate, because the tuple `b` would not be produced, as the tuple `a` would remain visible. The problem is caused because `rdp` is not a blocking primitive. However, this can easily be solved, by adding another statement to when the tuple should be removed:

iv. If a `rdp` is performed and the matched tuple has been tagged as read-only then the tuple should not be used as the result for the primitive. Furthermore, if there are no other matching tuples available then the marked tuple should be discarded. If another matching tuple is available as the result then the read-only tuple can be left.

At this point it now makes more sense to describe tuples as being marked rather than read-only (as `rdp` is a non-destructive primitive). Also, some implementations support primitives beyond the basic Linda primitives and `inp` and `rdp`. It is possible, to generalise the rule yet further:

iv. If any primitive is performed which does not block the user thread of execution until a matching tuple is found, and a tuple which is to be the result (or part of the result) of that primitive is a marked tuple then the marked tuple should not be used as the result or part of the result. If the marked tuple is the only available result or should be part of the result it must be removed, otherwise it can remain.

This rule generalisation is slightly conservative, in that sometimes tuples will disappear before they need to, but ensures that the rules should work with any set of access primitives. Bellow all the rules are shown, rewritten in general terms to provide a set of rules for when a tuple can be returned and when a tuple should be discarded.

When a tuple is to be used as a result or part of a result for a primitive and the tuple has been the result or part of a result for a destructive primitive then the tuple can still be used as a result to another primitive providing the following are all true:

i. The primitive being performed is not destructive.

ii. The primitive is not being performed by the same agent that performed the destructive primitive.

iii. The agent that performed the destructive primitive has not inserted any tuple nor caused the insertion of any tuple into any tuple space.

iv. The primitive being performed blocks the agents thread of execution until a result is returned, where a result is either a tuple or an indication of completion of some movement of tuples[2].

The system must subsequently discard this tuple when:

i. The agent that performed the primitive that removed the tuple inserts any tuple into any tuple space or performs a primitive that causes any tuple to be inserted into any tuple space.

ii. The agent that performed the primitive that removed the tuple terminates.

iii. The current non-destructive primitive does not block the user thread of execution until a matched tuple(s) is found or operation on a set of tuples is complete, and is forced to use the tuple to provide the results correctly.

It is assumed that there are no hidden communication channels between the agents communicating. The only way to agents can communicate is via a tuple space. The second rule covers hidden communication, by an agent knowing something has happened because of when it was created.

It should be noted that this optimisation does not interfere with the asynchronous "event" style primitives added in many new versions of Linda which propagate inserted tuples to agents automatically. When a tuple is inserted, it can be propagated provided the `out` is treated as an insertion of a tuple for discarding tuples.

4. IMPLEMENTATION

We have implemented the scheme outlined in the last section, and the implementation has proved simple and efficient. A Java based kernel has been extended to use this optimisation. It is a centralised kernel, as are most of the kernels currently being used for the new set of co-ordination languages. It supports the standard Linda primitives and `collect` [1] and `copy-collect` [14]. The `collect` primitive moves all tuples matching a given template from one tuple space to another and returns a count of the number of tuples moved (therefore, it is a blocking primitive). The `copy-collect` primitive is the same except it copies rather than moves the tuples. Therefore, it is based on the extended rule set given in the previous section.

[2]Primitives such as `rdp` fail this rule, because they do not block the thread of execution – it returns false if a tuple is not available. However, a primitive like `copy-collect` passes the rule because it copies tuples and then returns a counter.

	out	in	rd
Normal	0.193	0.122	0.271
Optimised	0.194	0.122	0.146

Table 1: Tuple space access times (ms).

Every agent using the run-time system has a Globally Unique Identifier (GUID) created dynamically as it starts to execute. When the agent registers with the run-time system the GUID is passed to the run-time server and it creates a counter associated with the agent. Each time an agent performs either an `out` or `collect` the counter associated with the agent is incremented by one (before the primitive is performed). When an agent requests a tuple using an `in` or when a set of tuples are moved from one tuple space to another tuple space using a `collect` the effected tuples are marked as "special" and tagged with the identity tag of the agent that removed or moved the tuple and with the current value of the primitive count associated with the agent. Any other agent can then perform a `rd` or `copy-collect` and have this tuple as the result or part of the result. However, whenever the tuple is matched the system checks the current primitive count associated with the GUID attached to the tuple with the primitive count attached to the tuple. If the primitive counts differ or if the agent has terminated then the tuple is discarded, and not used as a result for the `rd` or `copy-collect`.

Checking of the tuples is performed on the fly, and the data structure is so organised that newest inserted tuple will be found first therefore maximising the chances that the tuple found as a result for a primitive is unmarked. If it is feared that tagged tuples will remain in the system for some time then some form of garbage collection can be added. This garbage collection can be run as a background task, or performed when the data structure becomes too large.

Table 1 show the performance results for the data structures used within the Java kernel. The kernel is written in Java and is not written to be optimal and therefore the performance is not very good. However, the relative speeds demonstrate some interesting things about the implementation. The results were gathered on a 450MHz Pentium III processor running Windows NT by timing individually the insertion (`out`), then reading (`rd`) and then removal (`in`) of 10000 identical tuples. The average was worked out for each task over 10 executions, and then the results were scaled down from 10000 operations to individual tuple space access times. The test program was embedded into the kernel, so removing communication overheads and marshalling costs. The row labelled *Normal* represents the results when the optimisation is not used (and the code for providing the optimisation is removed) and the row marked *Optimised* represents the results when the optimisation is being used. All timings are given in milliseconds.

As one would expect the time taken to insert and read the tuples are the same regardless of whether the optimisation is being used. This is because the overhead of accessing the counter on insertion is negligible and the `rd` only has to perform an extra check to see if the tuple is ghosted (and in the results shown in the table a tuple is never ghosted.). The time taken to perform an `in` drops when the optimisation is being used. This is because in the tuple is not removed from the data structure; it is simply marked as ghosted. In a more optimal kernel (written in C++) we would expect the time taken to perform a `rd` and an `in` would be similar in both the optimised and normal senarios. In the current version there is an overhead of no more than 0.153 ms added to a `rd` primitive for every ghosted tuple that it removes from the data structure because it is no longer valid. So if, a single tuple is being checked and removed in a `rd` it takes 0.275 ms, which is the same as a `in` (we have passed the expense of removing tuples from the data structure from the `in` primitive to the `rd` primitive). We created two versions of the `rd` one that left the invalid ghosted tuples in the data structure and one that removed them, and added an explicit process that performed garbage collection on the tuple structure. The ideal may be an adaptive data structure that adapts its use based on the way a set of tuples is being accessed. Switching between dynamically garbage collection of the data structure and waiting for periods of low load and perform a static garbage collection.

We created a simple demonstration program based on the example given in Figure 1, where the reader agents repeatedly read the counter and printed all elements in list. When the technique described in this paper was enabled, none of the readers blocked awaiting a tuple, and regularly a tuple was returned when the primitive should have blocked. When the technique described in this paper was disabled, the readers blocked a significant number of times. However, the number of times a single reader is blocked is highly dependent on each experimental run, because of the impact of network latency on the interleaving of when primitives arrive.

5. CONCLUSIONS

We have described a method by which a run-time system can transparently optimise when tuples should be removed from a tuple space that does not alter the semantics of the access primitives. We have described the algorithm using the standard Linda primitives as an example and then generalised it to other primitives. We then described how the algorithm is implemented, cheaply and efficiently and presented performance results to support our claim.

This optimisation increases the level of concurrency in the system because primitives which would normally block no longer block. It also reduces the load in the server by reducing the number of primitives that need to be blocked. The optimisation can be used with asynchronous notification primitives and with transactions. It works with arbitrary tuples, and is not dependent on one particular coordination pattern (although one is used as the example in this paper). The optimisation is performed on-the-fly and does not require compile time analysis or the addition of other primitives.

Tuples are left partially visible until a set of rules is no longer satisfied and then they are discarded. It is not necessary that a `in` is followed by a `out` for this optimisation to work.

Although the optimisation is clearly correct we are currently working on a process algebra with formal proofs to show this is indeed the case.

6. ACKNOWLEDGEMENTS

I would like to thank Cedric Fournet and Rocco de Nicola for their detailed comments and corrections (and Rocco for his Petri-net diagram and case graph).

7. REFERENCES

[1] P. Butcher, A. Wood, and M. Atkins. Global synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, 1994.

[2] J. Carreria, L. Silva, and J. Silva. On the design of Eilean: A Linda-like library for MPI. Technical report, Universidade de Coimbra, 1994.

[3] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[4] N. Carriero and D. Gelernter. Tuple analysis and partial evaluation strategies in the Linda precompiler. In *Languages and Compilers for Parallel Computing*, pages 114–125. MIT Press, 1990.

[5] P. Ciancarini, A. Omicini, and F. Zambonelli. Coordination technologies for internet agents. *Nordic Journal of Computing*, 6(3):215–240, 1999.

[6] P. Ciancarini and D. Rossi. Coordinating Java agents over the WWW. *World Wide Web Journal*, 1(2):87–99, 1998.

[7] P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche. Coordinating multiagent applications on the WWW: A reference architecture. *IEEE Trans. on Soft. Eng.*, 24(5):362–366, 1998.

[8] R. D. Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Trans. on Soft. Eng.*, 24(5):315–330, 1998.

[9] A. Omicini and F. Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-agent Systems*, 2(3):251–269, 1999.

[10] G. Picco, A. Murphy, and G.-C. Roman. Lime: Linda meets mobility. Technical Report Technical report WUCS-98-21, Washington University, Department of Comp. Sci., St. Louis, Missouri, 1998.

[11] A. Rowstron. WCL: A web co-ordination language. *World Wide Web Journal*, 1(3):167–179, 1998.

[12] A. Rowstron. Mobile co-ordination: Providing fault-tolerance in tuple space based co-ordination languages. In *Coordination Languages and Models: Coordination99*, volume 1594 of *LNCS*, pages 196–210. Springer-Verlag, 1999.

[13] A. Rowstron and A. Wood. BONITA: A set of tuple space primitives for distributed coordination. In *HICSS-30*, volume 1, pages 379–388, 1997.

[14] A. Rowstron and A. Wood. Solving the linda multiple `rd` problem using `copy-collect`. *Science of Computer Programming*, 31(2-3), July 1998.

[15] Sun Microsystems. Javaspace specification, revision 0.4. Final Specification., 1997.

[16] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. TSpaces. *IBM Systems Journal*, 37(3):454–474, 1998.