Guard Compilation in Logic Shared Dataspace Languages

Mauro Gaspari

Technical Report UBLCS-93-14
18th of June 1993

Laboratory for Computer Science
University of Bologna
Piazza di Porta S. Donato, 5
40127 Bologna (Italy)
The University of Bologna Laboratory for Computer Science Research Technical Reports are available via anonymous FTP from the area ftp.cs.unibo.it:/pub/TR/UBLCS in compressed PostScript format. Abstracts are available from the same host in the directory /pub/TR/UBLCS/ABSTRACTS in plain text format. All local authors can be reached via e-mail at the address last-name@cs.unibo.it.

**UBLCS Technical Report Series**

- **93-5** *Some Modifications to the Dexter Model for the Formal Description of Hypertexts*, by S. Lamberti, C. Maioli and F. Vitali, April 1993.
- **93-7** *Distributed Programming with Logic Tuple Spaces*, by P. Ciancarini, April 1993.
- **93-8** *Coordinating Rule-Based Software Processes with ESP*, by P. Ciancarini, April 1993.
- **93-11** *Specification and Detection of Behavioral Patterns in Distributed Computations*, by Ö. Babaoglu and M. Raynal, May 1993.
Guard Compilation in Logic Shared Dataspace Languages\footnote{This paper will also appear in the proceedings of the ICLP'93 Post-Conference Workshop on Blackboard-Based Logic Programming.}

Mauro Gaspari\footnote{Mauro Gaspari is a researcher associated to the Dipartimento di Matematica, Università di Bologna, Piazza S. Donato 5, 40127 Bologna, Italy, e-mail: gaspari@csr.unibo.it.}

Technical Report UBLCS-93-14

June 1993

Abstract

In this paper we study problems related to the implementation of a class of rule-based shared dataspace languages. First we give a survey of these languages. Then we concentrate on the rule evaluation problem investigating both semantics and implementation issues. A new semantics for the evaluation of the precondition of a rule is proposed and techniques to efficiently compile them are described. In particular an indexing technique to support the compilation of these languages into Prolog is presented.
1 Introduction

We use the term logic shared dataspace languages, to refer to a class of rule-based shared dataspace languages, designed as an attempt to apply the generative communication model [Gel86] to a logic programming framework [Bro91], [Buc91], [Cia91]. Such languages differ from the Linda model since the base language adopted is Prolog and the operations on the tuple space are performed exploiting unification through a set of rewriting rules.

Logic shared dataspace languages provide powerful and high level mechanisms to coordinate the execution of agents. Similarly to coordination languages [Gel92], they are composed of a computational component which is Prolog and a coordination component which is a rule-based language. Communication among agents is performed exploiting rule evaluation on a logically shared tuple space which is a multiset of unit clauses called blackboard. Rules are composed of a precondition (also called guard), a goal and a postcondition. The guard expresses read and delete (also called input) conditions on the blackboard. When a guard succeeds, the Prolog goal is sent to a computational component associated to the rule. The postcondition adds new unit clauses into the blackboard and is performed on the basis of the goal evaluation result.

Rules are grouped together into agents. Two are the main features of an agent: first it provides an autonomous computational component, i.e., the goals of the associated rules are evaluated with respect to the same Prolog program; second the rules are evaluated in mutual exclusion. When a rule of an agent fires and the sequential execution starts, all the other rules of the agent are locked until the sequential execution terminates and the associated postcondition is asserted on the blackboard.

In this paper we give a survey of logic shared dataspace languages and we focus our's attention on the guard evaluation problem i.e., how guards match with the blackboard contents. We investigate both semantics and implementation issues. First we claim that the semantics of the guard evaluation process for the current language proposals is not completely satisfying and we propose a new semantics for it. Second we present a compilation technique and indexing mechanisms which allow to embed data driven computations into Prolog.

The paper is organized as follows: in Section 2 we surveys logic shared dataspace languages; in Section 3 we concentrate on the semantics of guard evaluation. In Section 4 implementation issues are considered. In Section 5 related works are presented and the connection with them is discussed. Finally in Section 6 performance results are presented.

---

3 "The term shared dataspace refers to the general class of models and languages in which the principal mean of communication is a common content-addressable data structure." [Rom89].

4 The name blackboard is due to the analogy with the blackboard metaphor [Nii86], a cooperation model widely used in Artificial Intelligence systems.
2 Logic Shared Dataspace Languages

The first concurrent languages based on the logic shared dataspace model are Shared Prolog (SP) [Bro91] and Multi-Prolog [DeB89], [DeB93]. The main difference between them is that SP perform operations on the blackboard by means of rewriting rules, while Multi-Prolog, and other extension of Prolog, based on the Linda model [Sic92], extend Prolog with blackboard management primitives. In the rest of the paper we analyze languages with structure similar to SP.

An SP program consists of the definition of an initial blackboard, a multiset of unit clauses, not necessarily ground, and of a set of agents, where each agent is constituted of a set of rules. A rule is composed of a guard, a goal and an outset (the postcondition). A guard is a sequence of delete conditions followed by a sequence of read conditions on the blackboard. The structure of a Shared Prolog system is static as agents cannot be dynamically activated on the blackboard. The number and the structure of the agents is known at compile time.

The guard may be interpreted as a Prolog compound goal, including also database updating, on a program of unit clauses. If the guard succeeds with substitution $\theta$, the goal, instantiated with the substitution $\theta$, is sent to the computational component associated to the rule. If the goal evaluation succeeds with substitution $\psi$, the outset instantiated with the substitution $\theta \circ \psi$ is added on the blackboard.

Other languages which belong to the logic shared dataspace family are Extended Shared Prolog (ESP), a hierarchical extension of Shared Prolog [Buc91], and Polis (Poli Spaces), a general model to support synchronization and communication among agents which cooperate using multiple tuple spaces [Cia91].

ESP has been extensively used as the coordination language of Oikos, an environment to support software process specification and enactment [Amb91]. Oikos provides a distributed runtime support and a programming environment for ESP [Amb92]. The runtime support is constituted by a set of processes running on a workstation cluster. The programming environment includes a compiler, a user interface, and a set of defined libraries.

In ESP, the Shared Prolog blackboard is partitioned in multiple hierarchically organized blackboards, similar to multiple tuple spaces in Linda [Gel89]. Many agents can be connected to each blackboard. Each agent includes a sequential Prolog program, a set of rules and is connected to a single blackboard, which is identified as its blackboard. New agents and blackboards may be created dynamically by means of a set of constructors called activation goals. Activation goals may contain also final conditions for blackboard: such conditions specify when the associated blackboard must terminate. A guard is a Prolog compound goal with respect to the database of unit clauses. It may include built-in predicates (a subset of built-in predicates defined in Prolog), read conditions on the blackboard (unit goal evaluation) and input conditions on the blackboard (unit clauses retract operation). Unit goals may be also negated and the intended semantics is the so called close world assumption, i.e., the goal $\text{not } g$ where $g$ is ground succeeds with respect to a database of unit clauses $\mathcal{P}$, if does not exist a unit clause $g'$ in $\mathcal{P}$ which unifies with $g$. The sequence of read conditions is called $\text{read\_guard}$, while the sequence which performs side effects on the blackboard is called $\text{input\_guard}$. The semantics of guard evaluation will be extensively discussed in the next section.

Before them several systems integrating the blackboard metaphor into the logic programming framework have been proposed [Bro88], [Dav87]. These systems are built to develop blackboard systems for artificial intelligence applications [Nii86], rather than to define new parallel logic programming languages.
ESP differs from SP in the following features: the number of blackboards (an SP system is based on a single blackboard while the ESP system is based on a hierarchical dynamic tree); the structure and the semantics of guard evaluation; the presence of an outset that takes into account the possible failure of the Prolog agent; the dynamic activation of new agents and blackboards.

A first formal semantics for a logic shared dataspace language is presented in [Bro91] for the language SP. This semantics models the distributed features of the language through a truly concurrent approach. A transition system which describes the interleaving semantics of a subset of Extended Shared Prolog called Kernel ESP, is presented in [Gas92a]; the transition system is used to describe the fairness properties of the language. Montangero and Chen describe a full semantics for ESP based on an algebraic approach [Che92]. An interesting proposal based on the use of path expressions to control non-determinism is presented in [Amb93]. All these proposals, except [Bro91], do not discuss the semantics of guard evaluation that we analyze in the next section.

We choose as a representative language of the family for our experiments Kernel ESP [Gas92a] a language similar to SP extended with the guard structure of ESP. A Kernel ESP program is presented in figure 1.

```
agent mickey-mouse:-
  robbed(X,Date)
  {thief(Y)}
  not in_prison(Y,_)  
  | {suspected(Y)}
  read_guard
  input_guard
  read_guard
  outset

# robbed(X,Date)
{thief(Y), in_prison(Y, Since)}
| preceding(Date, Since).
{suspected(Y)}
success_set
outset

; {}
failure_set

with

preceding(the(X,Y,Z), since(V,W,T)):- Z<T.
preceding(the(X,Y,Z), since(V,W,Z)):- Y<W.
preceding(the(X,Y,Z), since(V,Y,Z)):- X<V.
```

Figure 1: A Kernel ESP program.

The box represents the blackboard, it includes the blackboard initial state and two agents, mickey-mouse and police. In the definition of the agent mickey-mouse the rules are separated by the symbol
and the Prolog program associated to the agent is defined after the symbol with. The syntax of a rule is the following:

\[
\text{guard} \mid \text{goal. success_set ; failure_set}
\]

Input guards, the success_set and the failure_set are enclosed in braces.

3 Guard Evaluation

The semantics of guard evaluation for the different languages of the logic shared dataspace family is not consistently defined. In the first specification of Shared Prolog [Bro91] a guard was syntactically defined as composed by an input_guard followed by a read_guard. The semantics of the SP guard evaluation is defined as follows: the evaluation of the guard succeeds with answer substitution \( \theta = \theta_1 \circ \theta_2 \), if the input_guard succeeds, updating the blackboard, with substitution \( \theta_1 \), and the read_guard succeeds with respect to the updated blackboard with answer substitution \( \theta_2 \).

The implementation of this solution is difficult and time expensive: the results of the input_guard evaluation must be immediately reported on the blackboard to evaluate the read_guard and these updates must be possibly undone if the read_guard fails.

In a late definition of Shared Prolog the syntactical structure of the guard is changed, but unfortunately the semantic issues are not discussed [Cas92]. The guard is composed by a read_guard followed by an input_guard. We argue that the intended meaning of the guard evaluation is exactly the opposite of the previous one, in the sense that the guard succeeds if both the read_guard is true with respect to the initial blackboard and it is possible to delete all the facts specified in the input_guard.

This semantics is intuitively complete with respect to the model theoretic semantics of the guard evaluation. Let BB a blackboard, G a guard and IG the input_guard. We say that G is enabled with respect to BB if the following conditions hold:

\[
\text{BB} \models G \\
\text{IG} \subseteq \text{BB}
\]

The guard G must be a logical consequence of the blackboard (a logic program composed of unit clauses), and the input_guard must be included in the blackboard, (\( \subseteq \) is the multiset inclusion operator).

In Extended Shared Prolog the guard is a sequence of read and input conditions in an arbitrary order. The semantics of guard evaluation is described operationally in two steps [Buc91]:

1. The whole guard is evaluated as a read_guard (input conditions are treated as read conditions);
2. If the first step succeeds with answer substitution \( \theta \). The input operations, instantiated with the substitution \( \theta \), are attempted. If they all succeed the blackboard is updated, otherwise the original blackboard is restored and the whole guard fails.

Problems arise with this semantics when the input_guard contains two instances of the same unit goal both of which unify with the same unit clause in the blackboard. If this is the case, the first evaluation step of the guard succeeds and the second fails when we attempt to retract the second unit goal. Thus the guard fails even though a substitution satisfying it exists. As an example consider the following situation:
blackboard: \{a(1), b(1), a(2)\}
guard: \{a(X), b(Z), \{a(Y)\} \mid \ldots \}

Step one can succeed with substitution: \[X=1, Z=1, Y=1\]. But step two fails because only the first instance of \(a(1)\) can be retracted from the blackboard and so the whole guard fails even though the alternative substitution \[X=1, Z=1, Y=2\] might be applied successfully. This means that the operational semantics of guard evaluation defined for ESP is not complete with respect to its model theoretic semantics, since the success or the failure depends from the order in which atoms are selected from the blackboard. The error is that a further commit is introduced after the first step. To overcome these problems an ESP programmer can insert additional conditions into the guard, for instance:

\[
\{a(X), b(Z), \{a(Y)\}, X \neq Y \mid \ldots \}
\]

This solution is not completely satisfying since it obliges the user to insert explicit tests in the guard, in contrast with the declarative nature of the language. For instance, if we add a further input operation on the predicate \(a\): \{a(W)\}, the user must introduce three explicit tests:

\[
\{a(X), b(Z), \{a(Y), a(Z)\}, X \neq Y, X \neq Z, Z \neq Y \mid \ldots \}
\]

We propose a complete operational semantics for the guard evaluation in Kernel ESP that overcomes the problems described above. When the guard evaluation starts all the unit clauses which will be involved in it are statically defined. Input-(retract-)operations succeed only if there is a unit clause in the blackboard which can be retracted. The selected clause will not be available for further input operations but it is visible for read operations. The database updates will be performed only if the guard succeeds. The guard evaluation is an atomic operation. The semantics of the set of database updating operations is based on the transaction concept: all the updates are visible only if the guard evaluation succeed. Note that the order in which unit clauses are tried from the blackboard and the order in which read and input operations are selected is not specified for this semantics. In the next section we present a compilative approach to implement efficiently this semantics.

---

6 This is similar to the logical view of the semantics of Prolog dynamic code.
4 Compilation

Different compilation strategies are possible for Kernel ESP as described in [Cas92]. In the following we concentrate on a scenario in which a centralized blackboard exists and rules are evaluated with respect to it with simulated parallelism. This technique has been used extensively for implementing both SP and ESP [Amb90], [Gas92b]. A key issue in this scenario is the implementation of blackboard, the guard evaluation and the relative updating techniques. Prolog seems to be an adequate language to cover this task since it supplies built-in mechanisms such as unification, backtracking and dynamic database handling. The main problem is that Prolog does not support data driven computations, moreover it is difficult to implement in Prolog standard data driven algorithms such as rete match [For82]. We overcome this problem exploiting an indexing technique.

We choose a compilative approach: our aim is to perform guard evaluation on the blackboard as Prolog goals on a database of Prolog unit clauses. Since compilative techniques have been recently extensively studied for Prolog dynamic code [Bue86], [Clo85], [Col88], [Dem89], [Lin87] and almost all the available Prolog implementations provide mechanisms to manage the dynamic data areas (garbage collection), we choose to implement the unit clauses in the blackboard as Prolog dynamic code. As a consequence, updating operations are implemented by Prolog assert and retract built-in predicates. Moreover, we choose the same strategies of Prolog for literal selection in the resolvent, and for clause selection.

4.1 Guard compilation

We define a compilation technique for guards which efficiently realizes the semantics proposed in Section 3; guards are translated into Prolog code. An example of translation is presented in Figure 2.

\[
\begin{align*}
a(X), b(Y), \{ f(Z) \}, g(U), \{ f(X) \} & | \\
a(X), b(Y), \text{clause}(f(Z), \_, R), g(U), \text{clause}(f(X), \_, R1), \\
R /== R1, \text{erase}(R), \text{erase}(R1), & !
\end{align*}
\]

Figure 2: Guard compilation.

The Prolog built-in predicate \text{clause}/3 retrieves a database reference for a clause whose head unifies with the first argument. This translation schema may be applied to any Prolog that supports different update views for the dynamic code [Boe91], [Lin87], since retract operations are inserted at the end of the guard.

Compilation of guards may be optimized. For instance, if only one operation exists in the input\_guard and it is the last one of the guard, it is possible to avoid \text{clause}/3 call. Moreover, the explicit test on the reference returned by \text{clause}/3 is not needed if the main functors of the input\_guard conditions are different (see the examples in Figure 2).
The code of compiled guards is inserted in the compilation scheme for rules described in [Gas92b]. In the next section we describe an indexing technique devised to speedup the matching process.

4.2 Indexing

Since guards are evaluated left to right, it may happen that a failure occurs because one of the last elements of the guard is not present in the blackboard. This is really a drawback if the number of rules of the agents activated on the blackboard is large. To overcome this problem we devise an indexing mechanism which allow us to select a subset of the rules which potentially might match the blackboard (the *candidate* rules). This technique is based on a bitmap representation of the blackboard and the guards. Bitwise operations are used to select the candidate rules. To do this we consider an abstract representation of guards and blackboards. The abstract blackboard and the abstract guards are represented by sets of abstract goals. Given a goal, the corresponding abstract one is represented by its name, its arity and by a tag. For instance \( a(X,Y) \) is represented as \( a/2 \) plus the tag. We associate the following tags to abstract goals: \texttt{read} indicates read goals in the guard, \texttt{not(g)} indicates a negated ground goal\(^7\) \( g \), and \( n \in \mathbb{N} \) indicates multiple occurrences in the blackboard or in the input guards. For instance given the blackboard \( \{ a(1), a(2), b(X) \} \) the corresponding abstract blackboard is \( \{ b/1, a/1 \} \), and given the guard \( a(X), \texttt{not} b(X), \{ f(Z) \}, \texttt{not} b(1) \) the corresponding abstract guard is: \( \{ a/1 \texttt{read}, b/1 \texttt{not} (b(1)), f/1 \} \).

The blackboard is implemented by Prolog dynamic predicates (facts), by a bitmap (BB), which stores the presence of facts in the blackboard and by a vector, the occurrence vector (OC), which stores the number of occurrences of each fact in the blackboard. \( \text{oc} \) is initialized with the tag \( n \) of the corresponding abstract goal. The guards are implemented as Prolog compound goals, as described in Section 4.1, plus two bitmaps, and two conditions to perform indexing. The first bitmap indicates the presence of positive goals (RG). The second bitmap indicates the presence of negated goals (NG). The first condition \( I_G \) is introduced to test if multiple occurrences of input goals in the guard can be retracted from the blackboard. The second condition \( NC_G \) is introduced to test negated ground goals.

Let \( \Sigma \) be the set of all the abstract goals, considering only name and arity, which appear in a SP program\(^8\). We define a mapping \( \varphi \) from \( \mathbb{N} | \Sigma | \) to \( \Sigma \) which associates an element of \( \Sigma \) to each natural number in the interval \( [1,|\Sigma|] \).

---

\(^7\) Negated non ground goals are not included in the set.

\(^8\) Note that in order to apply this technique it must be possible to statically calculate the set \( \Sigma \).
Let $B_i$ the initial blackboard and $AB_i$ the corresponding abstract blackboard $BB$ and $OC$ are defined in figure 4.

\[
BB(i) = \begin{cases} 
1 & \text{if } \varphi(i) \in AB_i \\
0 & \text{otherwise}
\end{cases}
\]

\[
OC(i) = \begin{cases} 
n & \text{if } \varphi(i) \in AB_i \text{ has tag } n \\
0 & \text{otherwise}
\end{cases}
\]

**Figure 4: Blackboard bitmaps**

Let $G$ a guard and $AG$ the corresponding abstract guard. $RG$, $NG$, $IG$ and $NC_G$ are defined in figure 5.

\[
RG(i) = \begin{cases} 
1 & \text{if } \varphi(i) \in AG \text{ has tag } \text{read or } n \\
0 & \text{otherwise}
\end{cases}
\]

\[
NG(i) = \begin{cases} 
1 & \text{if } \varphi(i) \in AG \text{ has tag } \text{not} \\
0 & \text{otherwise}
\end{cases}
\]

\[
IG = \bigwedge_i OC(i) \geq n \forall i \text{ s.t. } \varphi(i) \in AG \text{ has tag } n > 1
\]

\[
NC_G = \bigwedge_i P \forall i \text{ s.t. } \varphi(i) \in AG \text{ has tag } \text{not}(P)
\]

**Figure 5: Rules bitmaps**

We say that a rule is a candidate rule if its guard $G$ satisfies the tests in figure 6.

\[
RG \land BB = RG
\]

\[
NG \lor BB = 0 \lor NC_G
\]

\[
IG = \text{true}
\]

**Figure 6: Candidate rules**

As an example consider the program in figure 7.
Program:

blackboard: \{a(1), b(2), c(3)\}
guard1: \{a(X), a(Y)\} b(X) \mid \ldots

guard2: c(Y), b(X) \mid \ldots

Mapping:

a/1: 0
b/1: 1
c/1: 2

Bitmaps:

BB=111
OC(0) = OC(1) = OC(2) = 1
R_1=011
R_2=110
I_1= OC(0) \geq 2

This technique has been implemented in Sicstus Prolog using the foreign language interface to C and bitwise C operations. The size of the bitmap used depends on the cardinality of \(\Sigma\); large bitmap are also supported.

Input and output operations on the blackboard are extended with bitmap operations and increment and decrement operations on the vector OC. These operations are part of the runtime system.

To embed this technique into Prolog we first associate an index to each rule (a natural number) and we compile the rules as follows:

\[
\begin{align*}
  p(1) & : - G_1, !, \ldots \\
  p(2) & : - G_2, !, \ldots \\
  p(3) & : - G_3, !, \ldots \\
  p(4) & : - G_4, !, \ldots \\
  p(5) & : - G_5, !, \ldots \\
  \vdots
\end{align*}
\]

Where the guards \(G_1, G_2, \ldots, G_n\) are compiled following the schema defined in the previous section. The indexing is performed by a C routine which applies the tests described above and calls the appropriate rule exploiting Prolog indexing and unification. Additional parameters of \(P\) which are used to specify different agents and scheduling policies are not considered here.

Further optimizations can be applied to the compiled structure of guards described in Section 4.1. All the ground negated goals can be removed since they are tested in the indexing phase.
5 Related Work

The execution model of logic shared dataspace languages is similar to the one of forward chaining production systems such as OPS5 [For82]. Both of them use a data driven computation rule, in the sense that agents react to the modifications done on a set of assertions which represent the state of the computation. From the computational point of view there are three main differences:

- In forward chaining production systems the working memory elements are ground, and pattern matching (instead of unification in logic shared dataspace languages) is used in the precondition evaluation phase.
- To select a rule in OPS systems all the applicable rules must be selected, the so called conflict-set, and the rule to apply is chosen after a conflict-resolution phase.
- Forward chaining production systems do not include update operations in the precondition of rules. This also means that the conflict-set may change in logic shared dataspace languages after a guard evaluation, thus the database evolution is non monotonic during guard evaluation (the match phase of logic shared dataspace languages).

The implementation of such systems is based on network algorithms which allow to select efficiently all the applicable rules [For82]. The performance achieved with this kind of algorithm is 400-800 wme (Working Memory Element) changes per second on a one Mips machine [Gup89]. This is obtained compiling the network directly into machine code. The rule evaluation process in logic shared dataspace languages is different from the rule evaluation process in OPS5 like systems. The rete match algorithm cannot be applied for such languages since the guard evaluation is non monotonic with respect to the blackboard of unit clauses and because unification is needed. Only one rule must be selected at a time and the evaluation of the whole rule is not atomic, i.e., after the evaluation of a guard succeeds, many other guards which belong to rules of other agents can be evaluated. Finally, delete operations on the database are part of the precondition and cannot be inserted in the postcondition which allows only insert operations.

The main difference with Linda based systems is that in logic shared dataspace languages communication and synchronization are performed through the evaluation of a set of rules, while in Linda a few basic operations based on pattern matching are provided. Apart from this distinction, the same set of techniques used by Linda implementors to speed up the matching process [Lei90] can be used for logic shared dataspace languages. The goal of these techniques is to reduce the amount of searching. The main techniques are based on:

- using of preprocessors to reduce and eliminate runtime searching. This technique has also been exploited for logic shared dataspace languages [Cas92], although it must be revised in order to be fitted to our compiled approach.
- restricting the key to a single field of the tuple and using hashing to access the tuple. Indexing of dynamic clauses is also supported by most available Prolog implementations. However, the use of hashing is debatable for logic shared dataspace languages because it introduces an overhead for database updating operations.
- splitting the tuple space into smaller pieces, by using of multiple tuple spaces. This feature is present in languages of the logic shared dataspace family such as ESP and Polis.

6 Benchmarks

A performance comparison with other works is difficult because logic shared dataspace languages are based on different computation mechanisms. The evaluation of the guard is more complex than
in rule-based systems since it includes also database updating operations. Moreover, the matching is more general and is based on unification; it is a variant of the multiset unification, i.e., to check if a guard is satisfied with respect to the blackboard we must unify the multiset representing the blackboard and the multiset representing the guard. This means that results such as the 400 wme changes per second of rule-based production system must be considered an upper bound for logic shared dataspace languages. In the following benchmark we evaluate the impact of the proposed compilation techniques in the implementation of these languages. We design a simple rules language to perform the experiment and we test three different implementations: an interpreter (similar to the interpreter of the current ESP implementation), a compiler without exploiting indexing, and a compiler exploiting indexing. The tests are executed on a Sun sparcstation 2. The results are summarized in the tables in figure 8.

<table>
<thead>
<tr>
<th>11 Rules</th>
<th>Interpreter</th>
<th>Compiler</th>
<th>Indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sec</td>
<td>122.33</td>
<td>38.519</td>
<td>23.949</td>
</tr>
<tr>
<td>wme change/sec</td>
<td>327</td>
<td>1039</td>
<td>1673.6</td>
</tr>
<tr>
<td>Rule fire/sec</td>
<td>81.7</td>
<td>259.7</td>
<td>418.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>22 Rules</th>
<th>Interpreter</th>
<th>Compiler</th>
<th>Indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sec</td>
<td>425.670</td>
<td>122.555</td>
<td>55.314</td>
</tr>
<tr>
<td>wme change/sec</td>
<td>187</td>
<td>653</td>
<td>1446.2</td>
</tr>
<tr>
<td>Rule fire/sec</td>
<td>46</td>
<td>163</td>
<td>361.5</td>
</tr>
</tbody>
</table>

Figure 8: Benchmark Results

Sicstus compiles in bytecode abstract instructions. The compilation in native machine instructions does not provide a great benefit: 2% for the compiler, and no benefit for the indexing. This is due to the extensive use of dynamic code and to conversion problems in the Sicstus interface with C\(^9\). The programs are composed respectively of 11 rules and 22 rules with 10 terms in the precondition, 4 wme updates for each rule. The execution of the programs involves respectively 10000 and 20000 rule firing. The speed-up achieved with the indexing technique is very good considering that the larger program is composed of just 22 rules.

---

\(^9\) A careful study of these problems could provide further optimizations. This kind of applications, which exploit a lot of interaction between Prolog and C, could benefit of environments which support full interoperability between the two languages [Att91].
7 Conclusion and Further Work

We present a survey of logic shared dataspace languages. We discuss the semantics of guard evaluation for such languages and propose a compilation technique to translate logic shared dataspace languages into Prolog. The proposed indexing technique is an interesting attempt to embed data driven computation into Prolog.

A number of issues are still open. First, it will be interesting to investigate if some techniques related to the sequential implementations of concurrent logic programming languages [Sha89] can be used also for logic shared dataspace languages [Hou89]. Second, it is interesting to study the impact of the presented indexing technique in the WAM code of guards, and the possible optimizations. Further investigation is needed to take into account the behaviour of dynamic logic shared dataspace languages such as ESP. Finally, it will be interesting to investigate if our indexing schema can be applied together with the techniques to control non-determinism, described in [Amb93].

Acknowledgements

We would like to thank Paolo Ciancarini for interesting discussions and comments. Paola Bruscoli and Nino Salibra have provided precious help in reading the manuscript. This work has been partially funded by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo sotto Progetto 6 under contract 91.00920.PF69. We would like also to thank all the members of Oikos team: Vincenzo Ambriola, Tito Flagella, Carlo Montangero, Laura Semini. Finally we would like to thank the anonymous referee who provided precious comments for improving the paper.
Bibliography


