Design and Implementation of a Multithreaded Virtual Machine for Executing Linear Logic Programs

Flavio Cruz
Carnegie Mellon University
Pittsburgh, PA 15213, USA
fmfernan@cs.cmu.edu

Ricardo Rocha
CRACS & INESC TEC
University of Porto
Rua Campo Alegre 1021/1055
4169-007 Porto, Portugal
ricroc@dcc.fc.up.pt

Seth Copen Goldstein
Carnegie Mellon University
Pittsburgh, PA 15213, USA
seth@cs.cmu.edu

Abstract

Linear Meld is a concurrent forward-chaining linear logic programming language where logical facts can be asserted and retracted in a structured way. In Linear Meld, a program is seen as a database of logical facts and a set of derivation rules. The database of facts is partitioned by the nodes of a graph structure which leads to parallelism when nodes are executed simultaneously. Due to the foundations on linear logic, rules can retract facts in a declarative and structured fashion, leading to more expressive programs. We present the design and implementation of the virtual machine that we implemented to run Linear Meld on multicores, with particular focus on thread management, code organization, fact indexing, rule execution, and database organization for efficient fact insertion, lookup and deletion. Our results show that the virtual machine is capable of scaling programs with up to 16 threads and also exhibits interesting scalar performance results due to our indexing optimizations.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming—Parallel Programming; D.3.4 [*PROCESSORS*]: Interpreters; D.3.4 [*PROCESSORS*]: Run-time environments

General Terms Design, Languages, Performance

Keywords Linear Logic, Virtual Machine, Implementation

1. Introduction

The last decade has seen a tremendous growth in content available in the World Wide Web, and, more specifically, in information generated from online social networks. The structure of such content is usually a graph, a very flexible structure suited to represent content where pairs of items are linked. In order to process such information, there has been an increased interest in running graph-based algorithms concurrently and efficiently on top of distributed networks and computer architectures (multicores). Currently available libraries and frameworks are built on top of imperative programming languages, which require the programmer to know how to properly use the framework and the language. Reasoning about such programs requires knowing the intricacies of the framework, how computation is scheduled and how processing units coordinate between each other.

Some well known frameworks include Dryad, Pregel and GraphLab. The Dryad system [19] combines computational vertices with communication channels (edges) to form a data-flow graph. The program is scheduled to run on multiple processors or cores and data is partitioned during runtime. Routines that run on computational vertices are sequential, with no synchronization. The Pregel system [28] is also graph based, although programs have a more strict structure. They must be represented as a sequence of iterations where each iteration is composed of computation and message passing. Pregel is specially suited to work on big graphs and to scale to large architectures. GraphLab [27] is a C++ framework for developing parallel machine learning algorithms. While Pregel uses message passing, GraphLab allows nodes to have read/write access to different scopes through different concurrent access models in order to balance performance and data consistency. While some programs only need to access the local node's data, others may need to update edge information. Each consistency model will provide different guarantees that are better adapted to some algorithms. GraphLab also provides different schedulers that dictate the order in which node's are computed.

Logic programming is an attractive approach to the graph-based algorithms, since logic-based languages provide a high-level, declarative approach to programming. An important characteristic of logic programming is that it offers great potential for implicit parallelism, thus making logic programs much easier to parallelize than imperative programs. First, logic programs are easier to reason about since they are based on logical foundations. Second, logic programmers do not need to use low level programming constructs such as locks or semaphores to coordinate parallel execution, because logic systems hide such details from the programmer.

Logic programming split into two main groups: *backwards-chaining* and *forward-chaining* languages. In a backwards-chaining programming language, programs are composed of a set of rules that can be activated by inputting a query. Given a query $q(\hat{x})$, an interpreter will work backwards by matching $q(\hat{x})$ against the head of a rule. If found, the interpreter will then try to match the body of the rule, recursively, until it finds the program axioms (rules without body). If the search procedure succeeds, the interpreter finds a valid substitution for the $\hat{x}$ variables. A popular backwards-chaining programming language is Prolog [7], which has been a productive research language for executing logic programs in parallel. Researchers took advantage of Prolog's non-determinism to

PPDP '14, September 8–10, 2014, Canterbury, UK.

ACM 978-1-4503-2947-7/14/09. . . $15.00.
http://dx.doi.org/10.1145/2643135.2643150

evaluate subgoals in parallel with models such as *And-parallelism* and *Or-parallelism* [14].

In a forward-chaining logic programming language, programs start with a database of facts (filled with the program's axioms) and a set of logical rules. The database of facts is then used to fire the program's rules and derive new facts that are then added to the database. This process is repeated recursively until the database reaches *quiescence* and no more information can be derived from the program. A popular forward-chaining programming language is Datalog [31].

We have designed Linear Meld (LM), a forward-chaining logic programming language that is specially suited for concurrent programming over graph structures [9]. LM differs from Datalog-like languages because it integrates both classical logic and linear logic [11] into the language, allowing some facts to be retracted and asserted in a logical fashion. Although most Datalog and Prolog-like programming languages allow some kind of state manipulation [24], those features are extra-logical, reducing the advantages brought forward by logic programming.

The roots of LM are the P2 system [25] and the original Meld language [3, 4]. P2 is a Datalog-like language that maps a computer network to a graph, where each computer node can perform computations locally and communicate with neighbors. Meld is inspired by the P2 system but adapted to the concept of massively distributed systems made of modular robots with a dynamic topology. LM still follows the same graph model of computation of Meld, which makes LM programs naturally concurrent since the graph of nodes can be easily partitioned to be executed by different threads. As a forward-chaining linear logic programming language, LM also shares similarities with Constraint Handling Rules (CHR) [6, 22]. CHR is a concurrent committed-choice constraint language used to write constraint solvers. A CHR program is a set of rules and a set of constraints (which can be seen as facts). Constraints can be consumed or generated during the application of rules. Some optimizations used in LM such as join optimizations and the use of different data structures for indexing facts were inspired by research done in CHR [18].

In this paper, we present the design and implementation of the LM virtual machine and compiler. The LM virtual machine was designed from scratch to run LM programs on multicore machines in an efficient manner[1]. The virtual machine is multithreaded and executes byte-code that is generated by the LM compiler. To test our language and virtual machine we have implemented several graph algorithms, search algorithms and machine learning algorithms, including: belief propagation [13], belief propagation with residual splash [13], PageRank, graph coloring, N-Queens, shortest path, diameter estimation, map reduce, quick-sort, neural network training, minimax, etc.

Our results show that our virtual machine is scalable and presents some interesting execution times when compared with other competing systems. The virtual machine uses a simple, but effective, work stealing algorithm that is able to balance the load across threads, improving scalability. Another important feature of our virtual machine is the dynamic indexing algorithm. It is a runtime algorithm that decides how to index logical facts and which data structure is best to use, so that database lookup and insertion time during rule application is effectively reduced.

This remainder of the paper is organized as follows. First, we briefly introduce the LM language. Then, we present an overview of the virtual machine and describe in more detail the code organization, thread management, rule execution and database organization. Finally, we present our experiments and outline some conclusions.

[1] Source code available at `http://github.com/flavioc/meld`

2. The LM Language

Linear Meld (LM) is a logic programming language that offers a declarative and structured way to manage mutable state. A program consists of a database of facts and a set of derivation rules. The database includes persistent and linear facts. Persistent facts cannot be deleted, while linear facts can be asserted and retracted.

The dynamic (or operational) semantics of LM is identical to Datalog. Initially, we populate the database with the *program's axioms* (initial facts) and then determine which derivation rules can be applied by using the current database. Once a rule is applied, new facts can be derive, which are then added to the database. If a rule uses linear facts, they are retracted from the database. The program stops when *quiescence* is achieved, i.e., when rules no longer apply.

Each fact is a predicate on a tuple of *values*, where the type of the predicate prescribes the types of the arguments. LM rules are type-checked using the predicate declarations in the header of the program. LM has a simple type system that includes types such as *node*, *int*, *float*, *string*, *bool*. Recursive types such as *list X* and *pair X; Y* are also allowed. Each rule in LM has a defined priority that is inferred from its position in the source file. Rules at the beginning of the file have higher priority. We consider all the new facts that have been not used yet to create a set of *candidate rules*. The set of candidate rules is then applied (by priority) and updated as new facts are derived.

2.1 Example

```
type left(node, node).                                          1
type right(node, node).                                         2
type linear value(node, int, string).                           3
type linear replace(node, int, string).                         4
                                                                5
// set of rules                                                 6
replace(A, K, New),                                             7
value(A, K, Old)                                                8
   -o value(A, K, New). // we found our key                     9
                                                                10
replace(A, RKey, RValue),                                       11
value(A, Key, Value),                                           12
!left(A, B),                                                    13
RKey < Key                                                      14
   -o value(A, Key, Value),                                     15
      replace(B, RKey, RValue). // go left                      16
                                                                17
replace(A, RKey, RValue),                                       18
value(A, Key, Value),                                           19
!right(A, B),                                                   20
RKey > Key                                                      21
   -o value(A, Key, Value),                                     22
      replace(B, RKey, RValue). // go right                     23
                                                                24
// initial configuration                                        25
!left(@0, @1).     !right(@0, @2).                              26
!left(@1, @3).     !right(@1, @4).                              27
!left(@2, @5).     !right(@2, @6).                              28
                                                                29
value(@0, 3, a).   value(@1, 1, b).                             30
value(@2, 5, c).   value(@3, 0, d).                             31
value(@4, 2, e).   value(@5, 4, f).                             32
value(@6, 6, g).                                                33
                                                                34
// update key 6 to value x                                      35
replace(@0, 6, x).                                              36
```

Figure 1. LM program for replacing a key's value in a binary tree dictionary

We now present in Fig. 1, a LM program that implements the update operation for a binary tree dictionary represented as key/value pairs. We first declare the predicates (lines 1-4) which represent the facts we are going to use. Predicate `left/2` and `right/2` are persistent while predicates `value/3` and `replace/3` are linear. Predicate `value/3` assigns a key/value pair to a tree node and predicate `replace/3` represents an update operation that updates the key in the second argument to the value in the third argument.

The algorithm uses three rules for the three cases of updating a key's value: the first rule performs the update (lines 6-9); the second rule recursively picks the left branch for the update operation (lines 11-16); and the third rule picks the right branch (lines 18-23). The initial axioms are presented in lines 26-33 and they describe the initial binary tree configuration, including keys and values. Finally, with the `replace(@0, 6, x)` axiom instantiated at the root node `@0` (line 36), we intend to change the value of key 6 to value `x`. Note that when writing rules or axioms, persistent facts are preceded with a '`!`'.

Figure 2 illustrates the trace of the execution. Note that the program database is partitioned by the tree nodes using the first argument of each fact. In Fig. 2(a), we present the database filled with the program's axioms. Next, we follow the right branch using rule 3 since $6 > 3$ (Fig. 2(b)). We then use the same rule again in Fig. 2(c) where we finally reach the key 6. Here, we apply rule 1 and `value(@6, 6, g)` is updated to `value(@6, 6, x)` (Fig. 2(d)).

2.2 Syntax

Table 1 shows the abstract syntax for rules in LM. An LM program $Prog$ consists of a set of derivation rules Σ and a database D. Each derivation rule R can be written as $BE \multimap HE$ where BE is the body of a rule and HE is the head. Rules without bodies are allowed in LM and they are called *axioms*. Rules without heads are also allowed. The body of a rule, BE, may contain linear (L) and persistent (P) *fact expressions* and constraints (C). Fact expressions are template facts that instantiate variables (from facts in the database). Variables can be used again in the body for matching and also in the head when instantiating facts. Constraints are boolean expressions that must be true in order for the rule to be fired. Constraints use variables from fact expressions and are built using a small functional language that includes mathematical operations, boolean operations, external functions and literal values. The head of a rule, HE, contains linear (L) and persistent (P) *fact templates* which are uninstantiated facts to derive new facts. Head expressions may use the variables instantiated in the body. The head can also have *comprehensions* (CE) and *aggregates* (AE).

Program	$Prog$	::=	Σ, D
Set of Rules	Σ	::=	$\cdot \mid \Sigma, R$
Database	D	::=	$\Gamma; \Delta$
Rule	R	::=	$BE \multimap HE \mid \forall_x.R$
Body Expression	BE	::=	$L \mid P \mid C \mid BE, BE \mid \exists_x.BE \mid 1$
Head Expression	HE	::=	$L \mid P \mid HE, HE \mid CE \mid AE \mid 1$
Linear Fact	L	::=	$l(\hat{x})$
Persistent Fact	P	::=	$!p(\hat{x})$
Constraint	C	::=	$c(\hat{x})$
Comprehension	CE	::=	$\{\ \hat{x};\ BE;\ SH\ \}$
Aggregate	AE	::=	$[\ A\ \Rightarrow\ y;\ \hat{x};\ BE;\ SH_1;\ SH_2\ ]$
Operations	A	::=	`min` | `max` | `sum` | `count`
Sub-Head	SH	::=	$L \mid P \mid SH, SH \mid 1$
Linear Facts	Δ	::=	$\cdot \mid \Delta, l(\hat{t})$
Persistent Facts	Γ	::=	$\cdot \mid \Gamma, !p(\hat{t})$
Terms	t	::=	$\mathsf{node}(n) \mid \mathsf{int}(n) \mid \mathsf{float}(f) \mid \mathsf{string}(s) \mid \mathsf{bool}(b) \mid l \mid \mathsf{pair}(t, t)$
Lists	l	::=	$\mathsf{nil} \mid [t \mid l]$

Table 1. Abstract syntax of LM

We created the concept of comprehensions to be used when the consumption of a linear fact should generate a set of facts accordingly to the current contents of the database. In a comprehension $\{\ \hat{x};\ BE;\ SH\ \}$, $\hat{x}$ is a list of variables, BE is the body of the comprehension and SH is the head. The body BE is used to generate all possible combinations for the head SH, according to the facts in the database. The following program shows a simple example that uses comprehensions:

```
!edge(@1, @2).
!edge(@1, @3).
iterate(@1).

iterate(A)
   -o {B | !edge(A, B) | perform(B)}.
```

When the rule is fired, we consume `iterate(@1)` and then generate the comprehension. Here, we iterate through all the `edge/2` facts that match `!edge(@1, B)`, which are `!edge(@1, @2)` and `!edge(@1, @3)`. For each fact, we then derive `perform(B)`, namely `perform(@2)` and `perform(@3)` in this example.

Another useful feature is the ability to reduce several facts into a single fact. For that, we have aggregates, a special kind of sub-rule that works very similarly to comprehensions. In a aggregate $[\ A\ \Rightarrow\ y;\ \hat{x};\ BE;\ SH_1;\ SH_2\ ]$, A is the aggregate operation, $\hat{x}$ is the list of variables introduced in BE, SH_1 and SH_2 and y is the variable in the body BE that represents the values to be aggregated using A. Like comprehensions, we use $\hat{x}$ to try all the combinations of BE, but, in addition to deriving SH_1 for each combination, we aggregate the values represented by y and derive SH_2 only once using y. As an example, consider the following program:

```
price(@1, 3).
price(@1, 4).
price(@1, 5).
count-prices(@1).

count-prices(A)
   -o [sum => P | . | price(A, P) | 1 | total(A, P)].
```

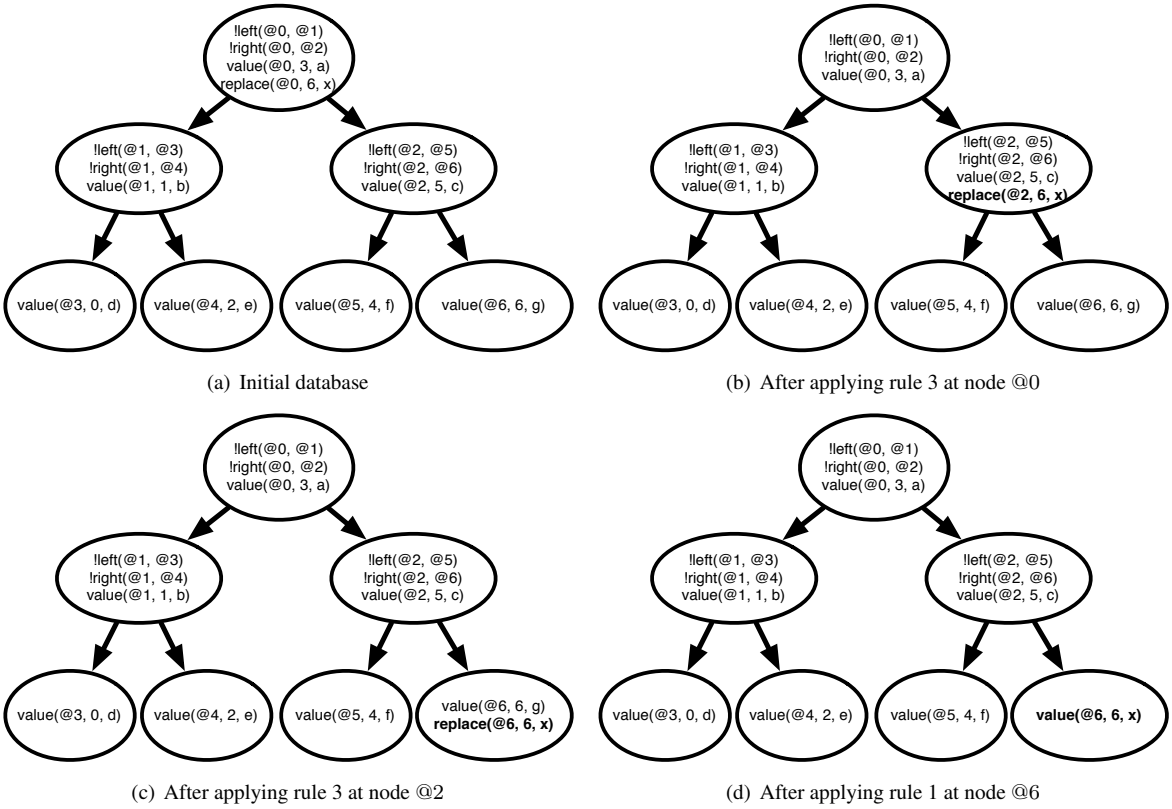
By applying the rule, we consume `count-prices(@1)` and derive the aggregate which consumes all the `price(@1, P)` linear facts. These are summed up and `total(@1, 12)` is derived. LM provides several aggregate operations, including the *minimum*, *maximum*, *sum* and *count*. Note that the `.` syntax indicates an empty list of variables and `1` is borrowed from linear logic and represents an empty derivation.

Comprehensions and aggregates are logically justified by the underlying proof system of the language. Our proof system is extended with greatest fixed points [5], which allows us to describe recursive definitions such as comprehensions or aggregates. A more detailed description is found in [9].

2.3 Concurrency

LM is at its core a concurrent programming language. The database of facts can be seen as a graph data structure where each node contains a fraction of the database. To accomplish this, we force the first argument of each predicate to be typed as a *node*. We then restrict the derivation rules to only manipulate facts belonging to a single node. However, the expressions in the head may refer to other nodes, as long as those nodes are instantiated in the body of the rule.

Due to the restrictions on LM rules, nodes are able to run rules independently without using other node's facts. Node computation follows a *don't care* or *committed choice* non-determinism since any node can be picked to run as long as it contains enough facts to fire a derivation rule. Facts coming from other nodes will arrive in order of derivation but may be considered partially and there is no particular order among the neighborhood. To improve concurrency, the programmer is encouraged to write rules that take advantage of



(a) Initial database

(b) After applying rule 3 at node @0

(c) After applying rule 3 at node @2

(d) After applying rule 1 at node @6

Figure 2. An execution trace for the binary tree dictionary program

the non-deterministic nature of execution since too much determinism will naturally imply less scalability and more synchronization when executing the programs.

3. The Virtual Machine

We have developed a compiler that compiles LM programs to byte-code and a multithreaded virtual machine (VM) using POSIX threads that runs the byte-code.

3.1 Threads

A key goal of our design is to keep the threads as busy as possible and to reduce inter-thread communication. When the VM starts, it reads the byte-code file and starts all threads. Initially, the VM will partition the application graph of N nodes into T subgraphs (the number of threads) and then each thread will work on their own subgraph. Reduction of communication between nodes in different threads is achieved by first ordering the node addresses present in the code in such a way that connected nodes are clustered together and then partitioning them to threads. During compilation, we take note of predicates that are used for communication (arguments with type *node*) and then build a graph of nodes from them. The nodes of the graph are then ordered by using a breadth-first search algorithm that changes the nodes of addresses to the domain $[0, n[$, where n is the number of nodes. Once the VM starts, we simply partition the range $[0, n[$ into T subgraphs.

Figure 3 presents the layout of our virtual machine for a program with six nodes and two running threads. Each thread space includes the nodes owned by the thread (the dotted arrows represent the edges between nodes) and a *Work Queue*, a single linked list containing *active nodes*, i.e., nodes that have new facts to process. Initially, the *Work Queue* is filled with all the nodes of the thread in order to derive the axioms.

During execution, threads can steal nodes of other threads to keep themselves busy. The load balancing aspect of the system is performed by our work scheduler that is based on a simple work stealing algorithm. The pseudo-code for the main thread loop is shown in Fig. 4. In each round, a thread inspects its *Work Queue* for active nodes with new candidate rules and, if there is any, procedure `process_node()` is called on the target node. Otherwise, if the *Work Queue* is empty, the thread attempts to steal one node from another thread. Starting from a random thread, it cycles through all the threads to find one active node. Eventually, there will be no more work to do and the threads will go idle. There is a global atomic counter, a global boolean flag and one boolean flag for each thread that are used to detect termination. Once a thread goes idle, it decrements the global counter and changes its flag to idle. If the counter reaches zero, the global flag is set to idle. Since every thread will be busy-waiting and checking the global flag, they will detect the change and stop executing.

3.2 Nodes

Figure 3 also illustrates the internal structure layout of a node, which includes: the database of linear facts (*Linear DB*); the database of persistent facts (*Persistent DB*); the rule matching structures (*Rule Engine*); and an auxiliary buffer for storing intermediate facts coming from other threads (*Fact Buffer*).

Whenever a new fact is derived through rule derivation, we need to update the data structures for the corresponding node. This is trivial if the thread that derived the fact also owns the node. If that is not the case, then we have to synchronize since multiple threads might be updating the same node's data structures. We added a

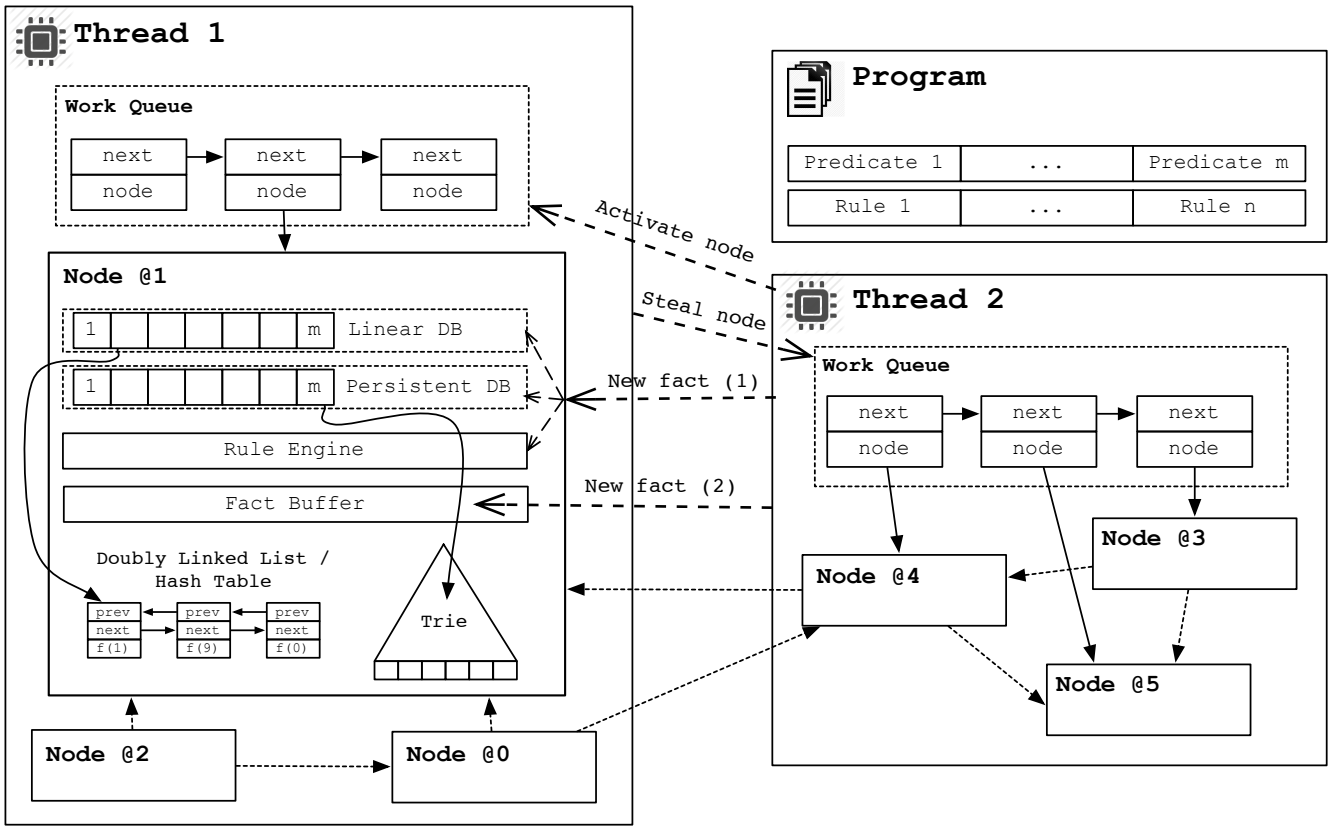


Figure 3. Layout of the virtual machine

```
void thread_work_loop(thread_id tid)
   while (true)
      node = pop_node_from_work_queue(tid)
      if (node)
         process_node(tid, node)
      else
         // need to steal a node
         target = random(NUM_THREADS)
         for (i = 0; i < NUM_THREADS && !node; i++)
            target = (target + 1)
            node = steal_node_from_thread(target)
            if (node) break
         if (!node)
            // try to terminate
            become_idle(tid)
            if (synchronize_termination(tid))
               return
            become_active(tid)
```

Figure 4. Thread work loop

lock and a boolean flag to each node to protect the access to its data structures. When the flag is activated, it means that the node is currently being executed by the owning thread. For example, in Fig. 3, if thread 2 derives a fact to node `@1` (owned by thread 1), then thread 2 checks the node's flag and if not activated, will lock node `@1` and perform the required updates (*New fact (1)*). If the flag is activated, it will not touch the main node data structures, but instead will add the new fact to *Fact Buffer* (*New fact (2)*). The facts stored in *Fact Buffer* will then be processed whenever the corresponding node's flag becomes active.

There is another thread interaction that might happen during fact derivation if the node receiving a new fact is not active. In such case, the sending thread needs to activate the node by pushing it to the *Work Queue* of the target thread. For example, consider again the situation in which thread 2 sends a new fact to node `@1`. If node `@1` is not active, then thread 2 also needs to activate it by pushing it to the *Work Queue* of thread 1. After this synchronization point, if the target thread is currently idle, it will become active and with a new node to process.

3.3 Database Data Structures

We said before that LM rules are constrained by the first argument and that each node has its own database of linear and persistent facts. Moreover, since only one thread at a time will be using a node's database, we do not need to deal with synchronization issues. Note also that the first argument of each fact is not stored.

The database must be implemented efficiently because during matching of rules we need to restrict the facts using a given *match object*, which fixes arguments of the target predicate to instantiated values. Each node's database is implemented using three kinds of data structures:

- *Trie Data Structures* are used exclusively to store persistent facts. Tries are trees where facts are indexed by common prefix arguments.
- *Doubly Linked List Data Structures* are used to store linear facts. We use a double linked list because it is a very efficient way to add and remove facts.
- *Hash Table Data Structures* are used to improve lookup when linked lists are too long and when we need to do search filtered

by a fixed argument. The virtual machine decides which arguments are best to be indexed (see Section 3.6) and then uses a hash table indexed by the appropriate argument. If we need to go through all the facts, we just iterate through all the facts in the table. For collisions, we use the above doubly linked list data structure.

Figure 5 shows an example for a hash table data structure for a `a(int,int)` predicate with 3 linear facts indexed by the second argument and stored as a doubly linked list in bucket 2. Each linear fact contains the regular list pointers, a `flags` field and the fact arguments. Those are all stored continuously to improve data locality. One use of the `flags` field is to mark that a fact is already being used. For example, consider the rule:

```
a(A,B), a(C,D) -o ...
```

When we first pick a fact for `a(A, B)` from the hash table, we mark it as being used in order to ensure that, when we retrieve facts for `a(C, D)`, the first one cannot be used twice since that would violate linearity.

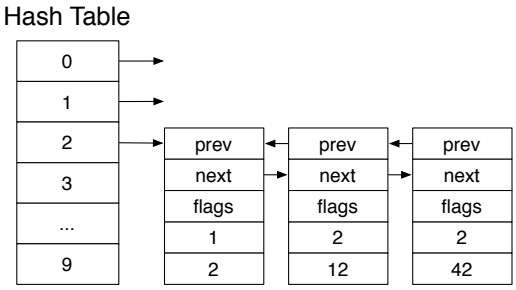


Figure 5. Hash table and doubly linked data structures for a `a(int,int)` predicate

3.4 Rule Engine

The rule engine decides which rules may need to be executed while taking into account rule priorities. Figure 6 shows the rule engine data structures in more detail. There are 4 main data structures for scheduling rule execution: `Rule Queue` is the bitmap representing the rules scheduled to run; `Active Bitmap` contains the rules that have enough facts to be fired; `Dropped Bitmap` contains the rules that must be dropped from `Rule Queue` and `Active Bitmap` if the rule being executed succeeds; and `Predicates Count` counts the number of facts per predicate. To understand how our engine works, consider the following set of facts and rules:

```
a.
e(0).

a, e(1) -o b.  // rule 1
a -o c.        // rule 2
b -o d.        // rule 3
e(0) -o f.     // rule 4
c -o e(1).     // rule 5
```

Since we have facts for predicates `a/0` and `e/1`, the `Active Bitmap` starts with rules 1, 2 and 4 marked as having enough facts to be fired. The `Rule Queue` bitmap also starts with the same three rules. In order to pick rules for execution, we take the rule corresponding to the least significant bit from the `Rule Queue` bitmap, initially the first rule `a, e(1) -o b`. However, since we don't have fact `e(1)`, this rule fails and we execute the second rule `a -o c`. Figure 6(a) shows the rule engine data structures at that point.

Because the derivation for the second rule succeeds, we will consume fact `a` and derive fact `c`. We thus update `Predicates Count` accordingly, mark the first and second rules in `Dropped Bitmap` since such rules are no longer applicable (`a` was consumed) and mark the fifth rule in `Active Bitmap` since `c` was derived. Finally, to update the `Rule Queue`, we remove the bits marked in `Dropped Bitmap` and add the active rules marked in `Active Bitmap` that use the newly derived predicates, rule 5 in this case. In the continuation, the engine will schedule the fourth and fifth rules to run. Figure 6(b) shows the rule engine data structures at that point.

Note that every node in the program has the same set of data structures presented in Fig. 6. We use 32 bits integers to implement the 3 bitmaps and an array of 16 bits integers to count facts.

We do a small optimization to reduce the number of derivations of persistent facts and, for that, we divide the program rules into two sets: *persistent rules* and *non persistent rules*. Persistent rules are rules where only persistent facts are involved. We compile such rules incrementally, i.e., we attempt to fire all rules where a persistent fact is used. This is called the *pipelined semi-naive* evaluation and it originated in the P2 system [25]. This evaluation method avoids excessing re-derivations of the same fact. The order of derivation does not matter for those rules, since only persistent facts are used.

3.5 Rule Execution

A byte-code file contains meta-data about the program's predicates, initial nodes, partitioning information, and code for each rule. Each VM thread has 32 registers that are used during rule execution. Registers can store facts, integers, floats, node addresses and pointers to runtime data structures (lists and structures). When registers store facts, we can reference fields in the fact through the register.

Consider the example in Fig. 7 that shows the LM byte-code for the rule `!a(X,Y), b(X,Z), c(X,Y) -o d(Y)`. Consider also a database with the facts `!a(1,2)`, `!a(2,3)`, `b(1,3)`, `b(5,3)`, `c(1,2)`, `c(1,3)`, `c(5,3)`. Rule execution for this rule and facts proceeds in a series of recursive loops, as follows. The first loop retrieves an iterator for the persistent facts of `!a/2` and moves the first valid fact, `!a(1,2)`, to register 0. The inner loop retrieves linear facts that match `b(1,Z)` (from the *join constraint*) and moves `b(1,3)` to register 1. The final loop moves `c(1,2)` to register 2 and the body of the rule is successfully matched. Next, we derive `d(2)`, where 2 comes from register 0.

```
PERSISTENT ITERATE a MATCHING TO reg 0
   LINEAR ITERATE b MATCHING TO reg 1
      (match).0=0.0                  // match argument X
      LINEAR ITERATE c MATCHING TO reg 2
         (match).0=0.0               // match argument X
         (match).1=0.1               // match argument Y
         ALLOC d TO reg 3
         MVFIELDFIELD 0.1 TO 3.0       // get argument Y
         ADDLINEAR reg 3                  // derive d(Y)
         REMOVE reg 2
         REMOVE reg 1
         TRY NEXT
      NEXT
   NEXT
RETURN
```

Figure 7. LM byte-code for rule `!a(X,Y), b(X,Z), c(X,Y) -o d(Y)`

In case of failure, we jump to the previous loop in order to try the next candidate fact. In case of rule success, the head is derived and we should backtrack to the inner most *valid loop*, i.e., the older loop that uses linear facts or, if there are no linear facts involved, to

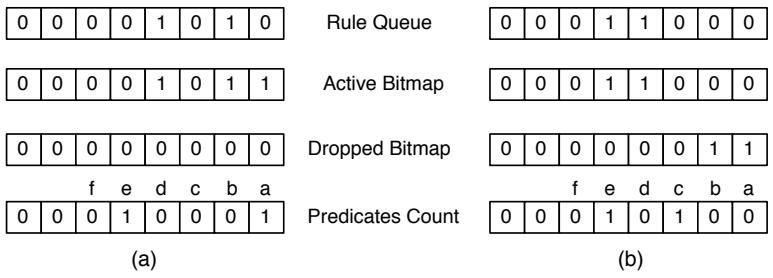


Figure 6. Rule engine data structures (a) before and (b) after applying the rule `a -o c.`

the previous loop. We need to jump to a valid loop because we may have loops with linear facts that are now invalid. In our example, we would jump to the loop of `b(X,Z)` and not `c(X,Y)`, since `b(1,3)` was consumed.

As an optimization, the compiler re-orders the fact expressions used in the body in order to make execution more efficient. For example, it forces the join constraints in rules to appear first so that matching will fail sooner rather than later. It also does the same for constraints. Note also that for every loop, the compiler adds the *match object*, which contains information about which arguments need to match, so that runtime matching be efficient.

Our compiler also detects cases where we re-derive a linear fact with new arguments. For example, as shown in Fig. 8, the LM byte-code for rule `a(N) -o a(N+1)` will compile to code that reuses/updates the old `a(N)` fact when deriving the new `a(N+1)` fact.

```
LINEAR ITERATE a MATCHING TO reg 0
   MVFIELDREG 0.0 TO reg 1           // initial argument
   MVINTREG INT 1 TO reg 2
   reg 1 INT PLUS reg 2 TO reg 3
   MVREGFIELD reg 3 TO 0.0      // reuse/update argument
   UPDATE reg 0
   TRY NEXT
RETURN
```

Figure 8. LM byte-code for rule `a(N) -o a(N+1)`

3.6 Indexing

To improve fact lookup, the VM employs a fully dynamic mechanism to decide which argument may be optimal to index. The algorithm is performed in the beginning of the execution and empirically tries to assess the argument of each predicate that more equally spreads the database across the values of the argument. A single thread performs the algorithm for all predicates.

The indexing algorithm is performed in three main steps. First, it gathers statistics of lookup data by keeping a counter for each predicate's argument. Every time a fact search is performed where arguments are fixed to a value, the counter of such arguments is incremented. This phase is performed during rule execution for a small fraction of the nodes in the program.

The second step of the algorithm then decides the candidate arguments of each predicate. If a predicate was not searched with any fixed arguments, then it will be not indexed. If only one argument was fixed, then such argument is set as the indexing argument. Otherwise, the top 2 arguments are selected for the third phase, where *entropy statistics* are collected dynamically.

During the third phase, each candidate argument has an entropy score. Before a node is executed, the facts of the target predicate are used in the following formula applied for the two arguments:

$$Entropy(A,F) = -\sum_{v \in values(F,A)} \frac{count(F, A=v)}{total(F)} \log_2 \frac{count(F, A=v)}{total(F)}$$

where A is the target argument, F is the set of linear facts for the target predicate, $values(F, A)$ is set of values of the argument A, $count(F, A = v)$ counts the number of linear facts where argument A is equal to v and $total(F)$ counts the number of linear facts in F. The entropy value is a good metric because it tells us how much information is needed to describe an argument. If more information is needed, then that must be the best argument to index.

For one of the arguments to score, $Entropy(A, F)$ multiplied by the number of times it has been used for lookup must be larger than the other argument. The argument with the best score is selected and then a global variable called `indexing_epoch` is updated. In order to convert the node's linked lists into hash tables, each node also has a local variable called `indexing_epoch` that is compared to the global variable in order to rebuild the node database according to the new indexing information.

Our VM also dynamically resizes the hash table if necessary. When the hash table becomes too dense, it is resized to the double. When it becomes too sparse, it is reduced in half or simply transformed back into a doubly linked list. This is done once in a while, before a node executes.

We have seen very good results with this scheme. For example, for the all-pairs shortest paths program, we obtained a 2 to 5-fold improvement in sequential execution time. The overhead of dynamic indexing is negligible since programs run almost as fast as if the indices have been added from the start.

3.7 Runtime Data Structures

LM also supports recursive types such as lists and pairs. These complex data structures are stored in the heap of the VM and are managed through reference counting. For instance, each list is a *cons cell* with 3 fields: `tail`, the pointer to the next element of the list; `head`, the element stored by this element of the list; and `refs` that counts the number of pointers to this list element in the VM. The list is deleted from the heap whenever `refs` is decremented to zero.

We avoid garbage collection schemes since objects are created and discarded in very specific points of the virtual machine and our objects cannot contain circular references. A reference counting mechanism is thus more appropriate than a parallel garbage collector which would entail pausing the execution of the program to garbage collect all the unused objects.

4. Experimental Results

This section presents initial results for our VM. First, we present a comparison with similar programs written in other programming languages in order to show evidence that our VM is viable. Then,

we present scalability results in order to show that LM programs can take advantage of multicore architectures.

For our experimental setup, we used a machine with 32 (2x16) Core AMD Opteron (tm) Processor 6274 @ 2.2 GHz with 32 GBytes of RAM memory and running the Linux kernel 3.8.3-1.fc17.x86_64. We compiled our VM using GCC 4.7.2 (g++) with the flags `-O3 -std=c+0x -march=x86-64`. We run all experiments 3 times and averaged the execution time.

4.1 Absolute Execution Time

To put our VM in perspective, we first compare it in terms of absolute execution time with other competing systems using a single thread.

In Table 2, we compare LM's version of the classic N-Queens puzzle against 3 other versions: a straightforward sequential program implemented in C using backtracking; a sequential Python implementation [33]; and a Prolog implementation executed in YAP Prolog [8], an efficient implementation of Prolog. Numbers less than 1 mean that LM is faster and larger than 1 mean that LM is slower. We can observe that LM easily beats Python, but is 5 to 10 times slower than YAP Prolog and around 15 times slower than C. Note however that, as we will see next, if we use at least 16 threads in LM, we can beat the sequential implementation written in C.

Problem Size	System		
	C	Python	YAP Prolog
10x10	16.92	0.62	5.42
11x11	21.59	0.64	6.47
12x12	10.32	0.73	7.61
13x13	14.35	0.88	10.38

Table 2. Comparing the absolute execution times (LM/System) for the N-Queens program

In Table 3, we compare LM's Belief Propagation (BP) program, a machine learning algorithm to denoise images, against a sequential C, Python and GraphLab [27] version of the algorithm. GraphLab is a parallel C++ library used to solve graph-based problems in machine learning. C and GraphLab perform about the same since they are both compiled to machine code, although the GraphLab version is highly optimized to run on multicore architectures. Python runs very slowly since it is a dynamic programming language and BP has many mathematical computations. We should note, however, that LM's version uses some external functions written in C++ in order to improve execution time, therefore the comparison is not totally fair.

Problem Size	System		
	C	Python	GraphLab
10	1.00	0.03	1.00
50	1.77	0.04	1.73
200	1.99	0.05	1.79
400	2.00	0.04	1.80

Table 3. Comparing the absolute execution times (LM/System) for the Belief Propagation program

We also compared a LM's version of the PageRank program against a similar GraphLab version and LM showed to be around 4 to 6 times slower. Our worse results were obtained for the all-pairs shortest distance algorithm where a LM's version of the problem was around 50 times slower than a C sequential implementation of the Dijkstra algorithm, but almost twice as fast when compared to the same implementation in Python.

4.2 Scalability

In this section we measure the scalability of the VM along with the performance gains due to work stealing and dynamic indexing. For this purpose, we used 4 configurations for the VM: **WI**, the full configuration that includes work stealing and dynamic indexing; **WN**, with work stealing but without dynamic indexing; **NI**, with indexing but without work stealing; and **NN**, without work stealing and without dynamic indexing. We ran each configuration using 1, 2, 4, 6, 8, 10, 12, 14 and 16 threads and compared the run time against the sequential execution (1 thread) of **WI**. We used the following set of programs:

- PageRank implements a PageRank algorithm without synchronization between iterations. Every time a node sends a new rank to its neighbors and the change was significant, the neighbors are scheduled to recompute their ranks.
- Greedy Graph Coloring (GGC) colors nodes in a graph so that no two adjacent nodes have the same color. We start with a small number of colors and then we expand the number of colors when we cannot color the graph.
- Shortest Distance (SD) computes the shortest distance of all nodes to all nodes.
- MiniMax, the AI algorithm for selecting the best player move in a game of Tic-Tac-Toe.
- N-Queens, the classic puzzle for a 13x13 board.
- Belief Propagation, a machine learning algorithm to denoise images.

The PageRank results are shown in Fig. 9(a). We used a search engine graph of 12,000 webpages[2]. Since this dataset follows the power law, that is, there is a small number of pages with a lots of links (1% of the nodes have 75% of the edges), it can be difficult to parallelize. Our results show that the VM is able to scale the program with up to 14 threads. We also notice the huge performance drop when we run the VM without work stealing. Dynamic indexing is also an advantage, since it detects that the facts for the pagerank of neighboring nodes need to be indexed efficiently.

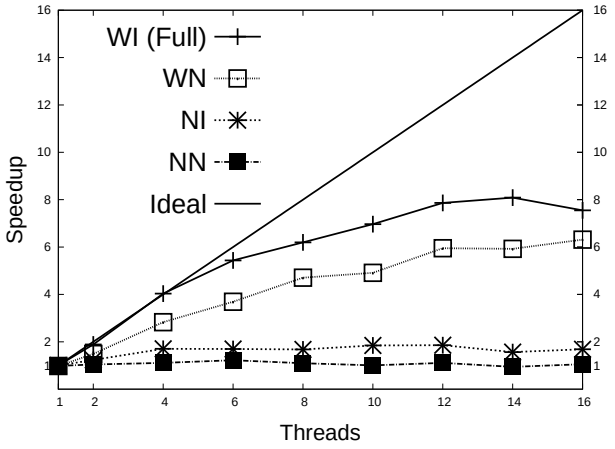
Figure 9(b) presents the results for the GGC program with a random dataset of 2,000 nodes with an uniform distribution of edges. There is a slight drop in scalability as the number of threads goes up, but the VM is still capable of reducing the run time. We note that in this program, the work available is reduced as the graph becomes increasingly colored.

In Fig. 10(a) we show the results for the Shortest Distance program. We attain a 13-fold speedup for 16 threads with both work stealing and dynamic indexing (**WI**). We note that indexing is more advantageous than work stealing because indexing the distance facts according to the source node is more crucial than improved load balancing.

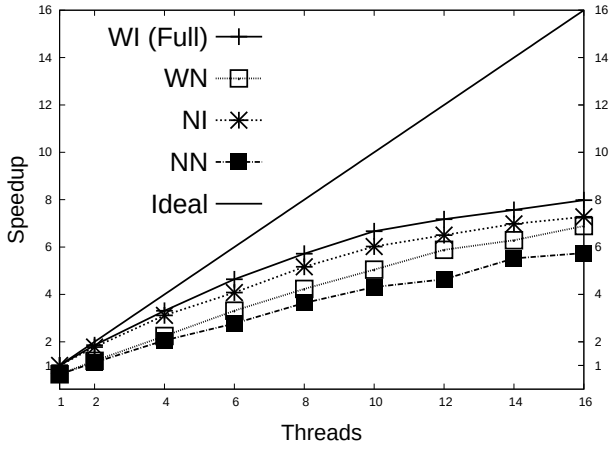
The results for the MiniMax algorithm are presented in Fig. 10(b). MiniMax is very different than the other algorithms because the graph of nodes is dynamic and is created during program execution. The load balancing is also problematic since there is little work to do in the initial and final phases of the algorithm. Still, our VM has decent performance, with almost a 7-fold speedup for 14 threads. The scalability drops with 16 threads but we think that is due to the simplicity of the current work stealing algorithm. Dynamic indexing has no effects in this program.

The results for the N-Queens program are shown in Fig. 11(a). The program is not regular since computation starts at the top of

[2] Available from `http://www.cs.toronto.edu/~tsap/experiments/download/download.html`

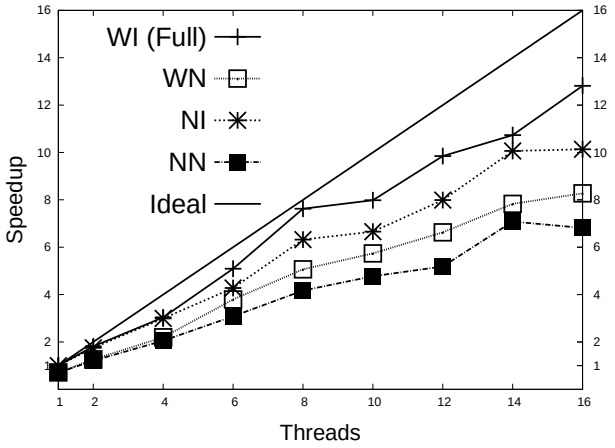


(a) PageRank using a graph of web pages with around 12,000 nodes and 292,000 edges

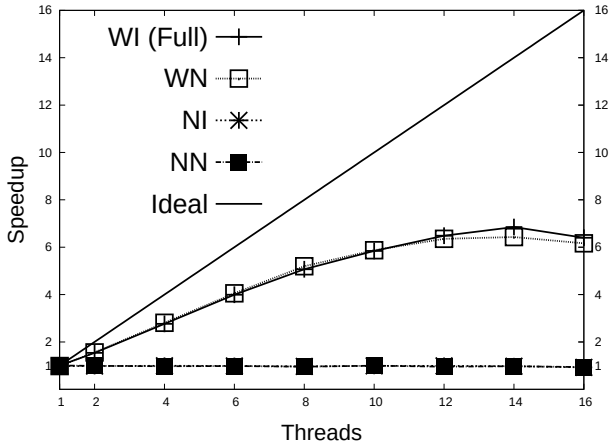


(b) GGC using a random graph with 2,000 nodes and 600,000 edges

Figure 9. Experimental results for the PageRank and GGC algorithms



(a) Shortest Distance for a graph with around 5,000 nodes and 13,000 edges



(b) MiniMax algorithm for the Tic-Tac-Toe game (complete tree)

Figure 10. Experimental results for the Shortest Distance and MiniMax algorithm

the grid and then rolls down, until only the last row be doing computation. Because the number of valid states for the nodes in the upper rows is much less than the nodes in the lower rows, this may potentially lead to load balancing problems. The results show that our system is able to scale well. When work stealing is left out (**NI** and **NN**), we see a serious drop in performance with 16 threads.

Finally, we shown the results for the Belief Propagation (BP) program in Fig. 11(b). BP is a regular and asynchronous program and benefits (as expected) from having multiple threads executing since the belief values of each node will converge faster. The super-linear results prove this assertion.

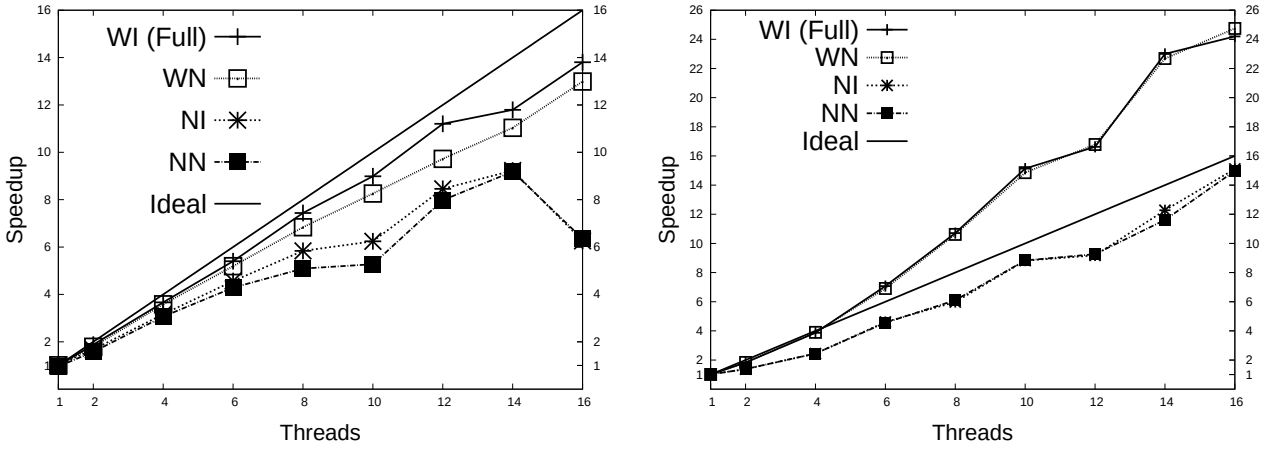
Program	Facts	Memory	Per Fact
PageRank	1,180,603	203,832 KB	176 bytes
GGC	2,363,536	292,682 KB	127 bytes
SD	502,347	45,387 KB	92 bytes
MiniMax	3	886,443 KB	295,481 KB
N-Queens	74,557	55,408 KB	760 bytes
BP	2,235,200	617,417 KB	283 bytes
Average (without MiniMax)			288 bytes

Table 4. Memory usage of programs after completion using 1 thread.

4.3 Memory Statistics

In this section, we present the amount of memory used by the VM after the programs presented in the previous section complete. We also count the number of facts stored in the database. This serves as an indication of how much memory is needed in terms of the number of facts stored in the database. Results are shown in Table 4.

The most unexpected result in Table 4 is that of the MiniMax program. There is a very low number of facts (that indicate the final player decision) and a huge amount of memory. How can this be explained? We note that MiniMax builds a huge decision tree of nodes with almost 9! leaves. Although these nodes do not participate in the computation after the MiniMax result is computed for them, the VM does not garbage collect nodes that do not contain facts and that do not have external references from other nodes. This is a potential improvement that can be done in the future.



(a) N-Queens program (13x13 board)

(b) Belief Propagation (400x400 image)

Figure 11. Experimental results for the N-Queens and Belief Propagation programs

PageRank, GGC and SD show a low memory usage per fact. This makes sense since the predicates used in those programs are relatively simple with only a few arguments. This changes with the N-Queens problem where the predicates have lists representing the valid positions of the queens in the board. Storing lists is far more expensive than storing integral values such as integers or floating point numbers. The same happens with BP because the belief values are stored as lists of floating point numbers.

5. Related Work

Virtual machines are a popular technique for implementing interpreters for high level programming languages. Due to the increased availability of parallel machines and distributed architectures, several machine models have been developed with parallelism in mind [21]. One example is the Parallel Virtual Machine (PVM) [32], which serves as an abstraction to program heterogeneous computers as a single machine. Another important machine is the Threaded Abstract Machine (TAM) [10, 12], which defines a self-scheduled machine language of parallel threads where a program is represented as conventional control flow.

Prolog, the most prominent logic programming language, has a rich history of virtual machine research centered around the Warren Abstract Machine (WAM) [34]. Prolog is naturally parallel because several clauses for the same goal (AND-parallelism) or all goals in a clause (OR-parallelism) can be tried in parallel. Different abstract machines for AND-parallelism has been developed for the WAM [16, 23]. For OR-parallelism we have several models such as: the SRI model [35], the MUSE model [1] and the BC machine [2]. Although these models are built on top of the WAM, some machines use different approaches, such as the PPAM [20] with its data-flow model.

Several linear logic programming languages have been developed in the past [29]. Lolli, a programming language based on a fragment of intuitionistic linear logic [17], proves goals by lazily managing the context of linear resources during backward-chaining proof search. Forum [30] is an extension of Lolli that is based on a classical proof system. While Lolli uses an intuitionistic proof system that supports only one conclusion, Forum allows multiple conclusions. Forum is therefore better adapted to represent concurrent computations than Lolli, since many multiple goals need to be proved at once. Another extension of Lolli is LolliMon [26], a concurrent linear logic programming language that integrates both forward and backward-chaining search, where the backward-chaining phase is done sequentially but the forward-chaining is done concurrently inside a monad. The backward-chaining phase is suspended between the forward-chaining phases, where a fix-point is computed. LolliMon is derived from the logical framework called CLF [36].

Lygon [15] is a backward-chaining linear logic programming language that extends Prolog with linear resources. Lygon is not only able to run Prolog programs but also allows the body of clauses to be linear that can be used exactly once in each query. In contrast to Prolog, where there is no clause selection, Lygon needs to use heuristics to decide which clause to prove first, since that will help finding a proof and finding it efficiently. Notably, Lygon is very suitable to solve graph-based problems due to the mutable state provided by linear clauses.

6. Conclusions

We have presented a parallel virtual machine for executing forward-chaining linear logic programs, with particular focus on thread management, code organization, fact indexing, rule execution and database organization for fast insertion, lookup, and deletion of linear facts. Experimental results show that our VM is able to scale the execution of programs when run with up to 16 threads. Our results also show the importance of having an efficient indexing mechanisms for facts. With our dynamic indexing, the VM automatically detects which predicates need to be indexed in order to improve performance. Due to these and other optimizations, the VM fairs relatively well against other programming languages, including compiled languages. Moreover, since LM programs are concurrent by default, we can easily get better performance from the start by executing them with multiple threads. As further work, we want to improve parallel scalability and take advantage of linear logic to perform whole-program optimizations, including computing program invariants, loop detection in rules and bypass of rule priorities.

We think that our virtual machine is a promising starting point to make logic programming more desirable in the data-mining, machine learning and distributed/parallel programming community. Moreover, our virtual machine can be easily extended to execute over computer networks or to execute programs on really big datasets. In a nutshell, LM provides a concise way to describe graph-based algorithms that can be more easily reasoned about, a clear advantage over competing systems.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions that helped us improve the quality of the paper.

This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme; by FCT (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program and project SIBILA (NORTE-07-0124-FEDER-000059); and by the Qatar National Research Fund under grant NPRP 09-667-1-100. Flavio Cruz is funded by the FCT grant SFRH/BD/51566/2011.

References

[1] K. Ali and R. Karlsson. Full prolog and scheduling or-parallelism in muse. *International Journal of Parallel Programming*, 19(6):445–475, 1990. ISSN 0885-7458. . URL `http://dx.doi.org/10.1007/BF01397627`.

[2] K. A. M. Ali. Or-parallel execution of prolog on bc-machine. In R. A. Kowalski and K. A. Bowen, editors, *ICLP/SLP*, pages 1531–1545. MIT Press, 1988. ISBN 0-262-61056-6. URL `http://dblp.uni-trier.de/db/conf/iclp/iclp88.html#Ali88`.

[3] M. P. Ashley-Rollman, M. D. Rosa, S. S. Srinivasa, P. Pillai, S. C. Goldstein, and J. D. Campbell. Declarative programming for modular robots. In *Workshop on Self-Reconfigurable Robots/Systems and Applications at IROS 2007*, 2007.

[4] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell. A language for large ensembles of independently executing nodes. In *International Conference on Logic Programming (ICLP)*, 2009.

[5] D. Baelde. Least and greatest fixed points in linear logic. *ACM Transactions on Computational Logic*, 13(1):1–44, 2012.

[6] H. Betz and T. Frühwirth. A linear-logic semantics for constraint handling rules. In *Principles and Practice of Constraint Programming - CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 137–151. 2005.

[7] A. Colmerauer and P. Roussel. The birth of prolog. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, pages 37–52, New York, NY, USA, 1993.

[8] V. S. Costa, L. Damas, and R. Rocha. The yap prolog system. *CoRR*, abs/1102.3896, 2011.

[9] F. Cruz, R. Rocha, S. Goldstein, and F. Pfenning. A Linear Logic Programming Language for Concurrent Programming over Graph Structures. *Journal of Theory and Practice of Logic Programming, 30th International Conference on Logic Programming (ICLP 2014), Special Issue*, 14(4 & 5):493–507, July 2014.

[10] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993. URL `http://www.cs.cmu.edu/~seth/papers/CullerGSvE93.pdf`.

[11] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[12] S. C. Goldstein. The implementation of a threaded abstract machine. Technical Report UCB/CSD-94-818, EECS Department, University of California, Berkeley, 1994. URL `http://www.cs.cmu.edu/~seth/papers/goldstein-tr94.pdf`.

[13] J. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. In *Artificial Intelligence and Statistics (AISTATS)*, 2009.

[14] G. Gupta, E. Pontelli, K. A. M. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel execution of prolog programs: A survey. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23 (4):472–602, 2001.

[15] J. Harland, D. Pym and M. Winikoff Programming in Lygon: An Overview In *International Conference on Algebraic Methodology and Software Technology (AMAST)*, 391-405, 1996.

[16] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, 1986. AAI8700203.

[17] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110:32–42, 1994.

[18] C. Holzbaur, M. J. G. de la Banda, P. J. Stuckey, and G. J. Duck. Optimizing compilation of constraint handling rules in hal. *CoRR*, cs.PL/0408025, 2004.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, pages 59–72, 2007.

[20] P. Kacsuk. *Execution Models of PROLOG for Parallel Computers*. MIT Press, Cambridge, MA, USA, 1990. ISBN 0262111497.

[21] M. Kara, J. R. Davy, D. Goodeve, and J. Nash, editors. *Abstract Machine Models for Parallel and Distributed Computing*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 1997. ISBN 90-5199-267-X.

[22] E. S. L. Lam and M. Sulzmann. Concurrent goal-based execution of constraint handling rules. *CoRR*, abs/1006.3039, 2010.

[23] Y. Lin and V. Kumar. And-parallel execution of logic programs on a shared memory multiprocessor:a summary of results. Technical report, Austin, TX, USA, 1988.

[24] M. Liu. Extending datalog with declarative updates. In *International Conference on Database and Expert System Applications (DEXA)*, volume 1873, pages 752–763, 1998.

[25] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, and J. M. Hellerstein. Declarative networking: Language, execution and optimization. In *International Conference on Management of Data (SIGMOD)*, pages 97–108, 2006.

[26] P. López, F. Pfenning, J. Polakow, and K. Watkins. Monadic concurrent linear logic programming. In *International Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 35–46, New York, NY, USA, 2005.

[27] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 340–349, 2010.

[28] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *International Conference on Management of Data (SIGMOD)*, pages 135–146, 2010.

[29] D. Miller. An overview of linear logic programming. In *in Computational Logic*, pages 1–5, 1985.

[30] D. Miller. A multiple-conclusion meta logic In *Logic in Computer Science (LICS)*, pages 272–281, 1994.

[31] R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23:125–149, 1993.

[32] V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2:315–339, 1990.

[33] G. van Rossum. Python reference manual. Report CS-R9525, Centrum voor Wiskunde en Informatica, Amsterdam, the Netherlands, Apr. 1995. URL `http://www.python.org/doc/ref/ref-1.html`.

[34] D. H. D. Warren. An abstract prolog instruction set. Technical Report 309, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Oct 1983.

[35] D. H. D. Warren. Or-parallel execution models of prolog. In *II and Colloquium on Functional and Logic Programming and Specifications (CFLP) on TAPSOFT '87: Advanced Seminar on Foundations of Innovative Software Development*, pages 243–259, New York, NY, USA, 1987. Springer-Verlag New York, Inc. ISBN 0-387-17611-X. URL `http://dl.acm.org/citation.cfm?id=67683.67699`.

[36] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. In *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. 2004. .