The Logic Virtual Machine Specification

Xining Li
Department of Computer Science
Lakehead University
Thunder Bay, Canada

©XINING LI 2000
1 Introduction

Prolog is a manifestation of logic programming. It separates the logic and control aspects of an algorithm, hides control details from programming, and allows a very high-level description of relationships among values. Prolog has been used in a wide spectrum of applications: symbolic processing, artificial intelligence, expert systems, simulation, planning and deductive databases, etc.

The first Prolog interpreter was designed with no concern for memory efficiency. Memory allocated during execution is not reclaimed before backtracking occurs [2]. The Edinburgh Prolog implementation [1] is the first to benefit from memory management efforts. Memory space is divided into two blocks, the local stack and the heap. The local stack contains the control part of execution. Hence some memory can be reclaimed before backtracking. The heap contains the data part. In general, space in the heap cannot be reclaimed before backtracking.

By 1983, Warren had developed the WAM (Warren Abstract Machine), a structure copying execution model for Prolog. The WAM is an efficient execution model consisting of a set of high-level instructions and a memory architecture for handling control and unification [3][4]. The WAM has been accepted as the standard basis for implementing Prolog for more than ten years. Most of the high performance Prolog systems, such as Aquarius, BIM, Quintus and SICStus, are based on the WAM and extend it for further efficiency [5][6].

The WAM simulates the conventional procedure call to control Prolog program execution. It adopts structure copying [8] to represent Prolog terms and defines a set of operations to deal with special cases of the general unification. The parameter passing in a procedure call consists of two phases: the put-phase and the get-phase. During the put-phase the arguments of the caller are loaded into argument registers and during the get-phase the values in the argument registers are unified with the arguments of the head of the callee.

Traditional Prolog implementations are based on the stack/heap memory architecture: stack holds local variables and control information, whereas the heap stores data objects which outlive procedure activations. A stack frame can be deallocated when an activation ends while heap space can only be reclaimed on backtracking or by garbage collection. Conventional garbage collection methods may yield poor performance. A reason for using stack/heap architecture is that deallocating stack frames is, in fact, cheap, incremental garbage collection.

In this document, I present the specification of a new Prolog execution model - the Logic Virtual Machine (LVM). The most significant difference from the WAM is that the LVM adopts a novel memory management approach. This scheme uses a single stack for all dynamical memory requirements, embedding an efficient garbage collection algorithm, the Chronological Garbage Collection (CGC), to reclaim useless memory cells. This approach can also be used in various logical/functional
programming language implementations. An experimental interpreter of LVM has been implemented. Our experimental results show that the proposed approach has low runtime overhead, good virtual memory and cache performance, and very short, evenly distributed pause times. Most benchmarks even revealed that the CGC not only improves the program's cache performance by more than enough to pay its own cost, but also improves the program execution performance which is competitive with the SICStus fast-code.

1.1 Terminology

To illustrate the major ideas of the LVM, let us consider first how Prolog program terminologies are used in this document. Other terminologies related with execution models and various implementation techniques will be introduced along the evolution of our discussion.

Term: Logic terms include constants, variables, and structures.

Variable: A logical variable stands for an unspecified individual and is used accordingly.

Ground term: A ground term is one in which variables do not occur.

Structure (compound term): A structure (or a compound term) comprises a functor and a sequence of one or more terms as its arguments.

Occurrence: A literal occurrence of a variable in a program clause. All the occurrences of a variable inside a clause refer to the same individual.

Flat occurrence: A variable occurrence which is not embedded in a structure.

Fact clause: A program clause without body.

Rule clause: A program clause with at least one goal in its body.

Chain clause: A rule clause whose body consists of the last and only user-defined goal.

Procedure: A collection of clause definitions with the same head functor and arity.

Unbound variable: A variable which has not been assigned to represent any other term. The convention for representing an unbound variable is to set the variable to contain its own address.

Bind: A binary operation with two memory addresses as its operands, at least one of which is an unbound variable. Its effect is to assign the unbound variable to hold the other's address.
Dereference: Deferring a variable is the process of starting from the variable and following a chain of pointers to access the variable's value.

1.2 Summary of Chapters

This document is organized as follows. In chapter 2, we will briefly review the principles of the WAM, its variants, and its existing problems. In chapter 3, I discuss issues of single stack model, the LVM architecture, and the stack allocation strategy. In chapter 4, I introduces the LVM Instructions - an intermediate language to the LVM byte-code. I proceed in chapter 5 to discuss implementation issues such as two stream code generation and memory expanding/contracting considerations. Chapter 6 will describe the idea of chronological garbage collection. In chapter 7, I will present the performance analysis: the LVM emulator will be compared with the SICStus fast code (binary code) and compact code (byte-code) respectively. Finally, I present the conclusion and outline of future work. Proposed implementation of instructions will be presented as Appendices.
2 The WAM and Beyond the WAM

For more than fifteen years, the WAM has been the paradigm for implementing Prolog and other logic languages. It serves performance by special unification instructions (get, put, unify) and control dispatching instructions (switch); and it features portability in the sense that its instruction set is coarse-grained, and its architecture is generic and simple. Most of the high performance Prolog systems, such as Aquarius, Parma, BIM, Quintus, SICStus and waMCC, are based on the WAM or the WAM data structure with refined instructions for further efficiency [5][14][6][7].

2.1 An Overview of the WAM

Generally speaking, the flow of control in a program corresponds to a depth-first traversal of the activation tree. A control stack is used to keep track of the chronological order of procedure calls and returns. Unlike the LIFO discipline of a stack, data structures in a heap may be allocated in any order. A heap object may outlive the procedure that created it. However, most objects live a very short time, while a small percentage of them live much longer.

The WAM is based on such a stack/heap model. It stores execution environments and choice-points in a stack, saves all dynamically created data objects in a heap, and uses two supplementary stacks, a trail and a pushdown list, to support backtracking and full unification. Stack frames can be deallocated on the return of procedure calls. The chronological order of procedure activations also makes tail recursion optimization and environment trimming possible. However, heap is allocated on-the-fly. Useless heap space can only be reclaimed on backtracking or by garbage collection.

The WAM supports the tail-recursion optimization: the effect of which is to turn some recursive procedures into equivalent iterative forms. It is also called Last Call Optimization (LCO), as it is applied systematically with or without recursion. It is more general as a stack frame recovery process.

The essence of LCO resides in the fact that local variables allocated to a rule should no longer be needed by the time all the put instructions preceding the last call in the body are passed. Hence, it is safe to discard the current environment before calling the last procedure of the rule’s body.

Another memory management scheme proposed by the WAM is the environment trimming: the intended effect of such a process is to make the current environment frame’s size on the stack shrink gradually, until it eventually vanishes altogether by LCO.

This gradual environment trimming can be made to work automatically by carefully ordering the variables in their environment so as to reflect the ordering of their associated last occurrence goals. Namely, the later a local variable’s last occurrence’s
goal is in the body, the lower its offset in the current environment frame is.
During compilation, the WAM needs to analysis and manipulate three kinds of variables:

**register (temporary) variables**: mostly used for passing arguments from a caller to a callee. They are directly accessible even if they are in fact simulated by memory cells. Their life time is scoped from caller-callee unification to the first goal in the callee’s body.

**local (permanent) variables**: used for holding intermediate values during the execution of a procedure. Any variable which occurs in more than one body goal should be a local variable. Such a variable is frame_base + index accessible. Their life time is restricted to the scope from entering the callee to its return.

**heap variables**: used for holding values useful across procedure calls. They are accessed through the current heap pointer or a local reference indirectly. They are long lived unless being discarded by a garbage collector or by backtracking.

For the purpose of utilizing memory efficiently, memory is allocated with the preference order of registers, local variables and finally heap variables. By the variable classification, three cases of variable-variable bindings are possible: (1) heap-heap, (2) stack-stack, and (3) heap-stack. The WAM enforces strict rules for the bind operation:

**Binding Rule 1**: always make the variable of higher address reference that of lower address.

**Binding Rule 2**: heap variables must never be set to a reference into the stack.

**Binding Rule 3**: the stack must be allocated at higher addresses than the heap, in the same global address space.

Even if the above binding rules are followed, there still exists some violation because of the use of LCO. For example, a stack variable is discarded before calling the goal in which it last occurs although it may still be unbound or bound to another unbound stack variable in the same frame. If the LCO disposes this stack frame, the danger is then that the call may refer to the discarded variable. A variable which bares this danger is called an *unsafe* variable. Hence, the WAM introduces a special instruction to globalize an unsafe variable.

Prolog is a nondeterministic programming language. In order to cope with non-deterministic resolution, the WAM adopts a trail stack to save locations of bound variables that have to be unbound on backtracking. Saving variables is called *trailing*, and restoring them to unbound is called *detrailing*. Not all variables that are
bound have to be trailed. A variable must only be trailed if it continues to exist on backtracking. A trail condition is checked for each variable on trail.

The WAM instruction set consists of three subsets: unification, control and nondeterministic manipulation. Control instructions mimic conventional procedure call and return. Nondeterministic manipulation instructions set up choice points and implement backtracking. Perhaps the most important feature of the WAM is the way it handles unification. The WAM uses structure copying model to represent logic terms. Terms of different types fit in the size of a machine word and are discriminated by explicit tags. Non-structure terms can be handled quite efficiently in most cases. Their unification operations could be simplified into matches and assignments. When a variable comes to stand for a structure, however, a concrete instance of the structure must be created on the heap, which includes copying the ground description of the structure and allocating a heap cell for each argument in the structure.

The WAM uses a set of general purpose registers as an interface and a two-stage operation (put-get) to pass arguments through the interface from a caller to a callee. Unification is done for each pair of actual-formal argument. In particular, unification of an actual argument with a compound data term (formal argument) known at compile-time is decomposed into instructions (get and unify) to handle the functor and arguments separately. The same instruction sequence is used to take part an existing structure (a selector) in read mode or to built a new instance of the structure (a constructor) in write mode. The set of get instructions are used to unify with the head arguments (formal arguments) and determine the mode flag. The unify instructions unify with the arguments of structures and have different behavior in read and write mode.

Based on the WAM, there are two main approaches to efficient Prolog implementation: emulated (byte) code and native (binary) code. Bytecode consists of abstract machine instructions and is executed by a virtual machine emulator. Binary code compiles to the target machine and is executed directly. (In general, a byte code program is 3-5 times slower than its binary code counterpart.) The original WAM was designed with an emulated implementation in mind. Its unification instructions are more suited to emulated code.

2.2 Evolutionary Developments

The WAM was the first step towards the execution efficiency of Prolog. However, the coarse-grained WAM instructions and the requirement of read/write mode have several problems:

1. Write mode is not propagated to subterms.
2. Instructions have modes. A mode flag is set in get_list and get_structure instructions and is tested in all unify instructions.

3. Poor translation to native code. Interleaved read/write code results in many jumps.

In order to solve the above problems, modern native code compilers adopt the so-called two-stream unification: an elegant scheme for compiling unification that is much more efficient than the WAM for native code implementation (speedup factor of 2 to 3). The key insight is that unification should be done in two instruction streams, one for read mode and one for write mode, with conditional jumps between them. In this way, one avoids superfluous operations while keeping a linear code size. The practical problems that remain are how to configure the instructions so they work correctly despite being jumped to from different places and how to minimize bookkeeping overhead for the jumps.

Generating two stream code looks to be quite complicated. The compiler needs to arrange code segments being properly labeled and conditional jumps being properly inserted. As [15] pointed out, some bookkeeping overhead is necessary. Detailed examples were given in [28] which showed the idea of treating the deeper nested structures later than the less deeply nested ones. For example, to collapse the most jumps, the compiler should reorder the arguments of all subterms to unify the most complex subterms last, or adjust the structure pointer to avoid jump backs from a left-nested structure. The reordering of the arguments and adjusting structure pointer complicates the code generation. For instance, the write stream must depend on the read stream code for a proper unification order.

Another example is the SICStus Prolog [25]. In this scheme, a Prolog program is translated first to WAM code, then to SAM code, and finally binary code. Translator _pl_wam generates get and unify largely as in the original WAM, and translator _wam_sam expands these to realize a version of two-stream algorithm. The compound term, regarded as a tree with a node per compound subterm, is traversed, unifying with the term that was represented by the variable at compile time. As long as this is nonvariable, execution proceeds in the read mode stream. When a variable is encountered where a compound subterm is to be, execution branches to the write mode stream, which builds the compound subterm, Afterward, execution returns to the read mode stream. Each subterm corresponds to exactly one sequence of instructions in each stream - code size is linear in term size. Write mode is propagated to subterms. The order of traversal is the traditional depth-first - subterms are not reordered. Given this order, an economical branch apparatus is generated.

Since the development of the WAM in 1983, there have been several abstract machines which either based on novel models of execution very different from the WAM or based on the WAM but introducing great modifications.
An alternative to the WAM is the Vienna Abstract Machine (VAM)[13]. The VAM eliminates the parameter passing bottleneck of the WAM by unifying caller and callee’s arguments in one step. In the VAM, temporary (register) variables cannot be shared between the head and the first subgoal. They have to be stored as permanent (local) in stack. Caller’s arguments are passed in stack (the use of registers has some restrictions).

Naive LCO is performed by updating the references and copying the new stack frame over the old stack frame. Further improvement implements two versions of LCO. The first variant (post-optimization) is as follows: if the determinacy of a clause can be determined during run time, the registers containing the head variables are stored in the caller’s frame. Head variables which resides in the stack frame due to the lack of registers are copied from the callee’s frame to the caller’s frame. The second version, if the determinacy of a clause can be detected during compile time, the caller’s and the callee’s stack frame are the same. Now all unifications between variables with the same offset can be eliminated. If not all head variables are held in registers reading and writing variables must be done in the right order. This is called pre-optimization.

The LAM$^{1\frac{1}{2}}$ is a Prolog abstract machine[26] developed by the author. It separates control and unification code and cooperates two engines: an one-program-counter (1P) control engine and a two-program-counter (2P) unification engine. The 1P engine executes control instructions which are similar to the WAM’s counterpart, which include stack allocation, initialization, execution control, nondeterministic control, and environment manipulation. Unification instructions, however, are executed like the VAM$_{2P}$ which eliminates the register interface by unifying goal and head arguments in one step. Although the 2P engine looks like the VAM$_{2P}$ in the sense that they both use the merged caller-callee unification, yet an essential difference is that the VAM$_{2P}$ adopts a lazy structure copying strategy, whereas the LAM$^{1\frac{1}{2}}$ works on the program sharing model. To enable fast decoding of a pair of instructions, the VAM$_{2P}$ defines separate sets of instructions for head and goal arguments and the sum of a pair of op-code must be unique. However, when two terms to be unified are dynamic structure instances, a full unification procedure must be invoked. On the other hand, the LAM$^{1\frac{1}{2}}$ specifies a set of neutral unification instructions with the same format (a single word). This instruction set is used not only for encoding (static) program terms but also for representing dynamic term instances. The advantage is that the 2P engine can execute any static/dynamic instruction combination. However, the costs of instruction decoding and operand calculation in the LAM$^{1\frac{1}{2}}$ emulator are more expensive. The VAM$_{2P}$ and LAM$^{1\frac{1}{2}}$ are well-suited for an intermediate code interpreter or a Prolog virtual machine. Unfortunately, using two program counters almost precludes native code compilation. Whether the PS model can be used to generate highly optimized, native code, such as the work done by Aquarius Prolog, remains to be investigated.
The ATOAM [27] is a variant of the WAM in the sense that it shares the same term representation and the major part of execution model of the WAM. It differs from the WAM mainly in that 1) arguments are passed directly into stack frames; 2) only one frame is used for each procedure call, and 3) procedures are translated into matching trees if possible, and clauses in each procedure are indexed on all input arguments.

A stack frame (if necessary) is allocated in three steps: 1) the caller prepares arguments on the top of the stack, 2) before issuing a call, the caller allocates and initializes two slots (continuation pointer and old activation frame pointer), and 3), depending on the compiled property of the called procedure, the callee allocates and initializes additional slots.

A stack frame is deallocated by return instruction (the WAM uses the pre-deallocate while the ATOAM uses the post-deallocate). The ATOAM does not support general LCO, instead it provides instructions to support TRO (Tail Recursive Optimization). A disadvantage of the ATOAM is that its non-recursive call chain is returned in the reverse order, whereas the WAM implements long-return, i.e., bypass the middle last calls and returns to an ancestor directly.

BinProlog [29] is based on a simplified WAM that is efficient for executing binary programs - programs with clauses having only one literal in the body. Binarization by continuation passing is used as a preprocessing step, working on a clause by clause basis. A clause like c(A) :- a(A), e(A, B), b(B) becomes c(A, Ct) :- a(A, e(A, B, b(B, Ct))) where Ct is a new variable representing the continuation that is recursively passed between calls.

The result of binarization is that it gives up WAM’s environment and puts on the heap continuation, recursively embedded in the last arguments of binary programs. As a consequence, the heap consumption of the program goes up.

The WAM features such as register interface and LCO create a bottleneck in parameter passing and unsafe variables. The above abstract machines try to attack this problem by different means. Most of the optimizations that extends the WAM are intended to reduce memory use and implement fast unification.

Yet another possible scheme is to allocate variables occurred in the last goal as heap variables. Thus arguments are passed through stacks or heap frames, and pre-deallocate of stack frame can be easily implemented because arguments used in the last call are entirely stored in heap and the local stack frame is free to be reused. This will increase the amount of heap consumption, however, if we have a fast GC to collect heap garbage, it is a considerable paradigm.

One step further, a more interesting scheme is to use the stack only and applying the incremental garbage collection. This scheme reclaims space not as a deliberate garbage-collection operation but as a natural way of collecting useful results. This scheme will be fully investigated in this document.
3 The LVM Architecture

Although the stack/heap architecture has been widely accepted, several problems merit further investigation. First, separating memory space into a heap and a stack may degrade the scale of program's data locality. In the past 10 years, peak processor speeds and memory sizes increased by nearly three orders of magnitude. “Yet another trend portends more difficulty in achieving much higher application performance in the coming years - the disparity between speed increases in processors (60% per year) and in DRAM memories (7% per year). This trend, coupled with physically distributed memory architectures, is leading to very nonuniform memory access time, with latencies ranging from a couple of processor cycles for data in cache to hundreds of thousands of cycles.” [30] Therefore, locality becomes ever more important to cache and virtual memory performance, poor locality may yield more cache misses and page faults. Second, the stack/heap architecture requires different management strategies and system resources (such as registers). This may complicate the language implementation. For example, the WAM needs to check the binding direction to avoid the case that a heap variable is bound to a pointer to a stack variable. Third, stack frames cannot always be deallocated (unless a separate choice stack is used) because some of them might be frozen by choice points. This will cause extra overhead in stack allocation/deallocation. For example, in stack allocation, the WAM has to compare the current environment register and the latest choice point register to determine which is the latest frame on the stack. As well, a deallocation action becomes meaningless when the frame has been frozen.

Can we combine the stack and the heap into a single memory block for use? The answer is certain because the first Prolog as well as the Algol 60 were built on this paradigm. In the pure stack discipline, all data values produced by the computation are kept in a single memory block. The beauty of this discipline is that every point of allocation is paired with a point of deallocation, and these points are easily recorded by the chronological order of program execution. The success of this paradigm is based on that the actual size of a value must be known at the point of memory allocation. This requirement causes a severe problem that recursively defined data types, such as list, tree, etc., are ruled out, or otherwise, stack frames cannot be deallocated upon completion of procedure calls because they might contain useful values constructed incrementally. The first Prolog interpreter was designed with no concern for memory efficiency. Memory allocated during execution is reclaimed only on backtracking. Hence, if we followed the trace of the first Prolog implementation, we would timewarped to the Stone Age. My hypothesis is that

if a high efficiency garbage collection operation is periodically invoked to reclaim useless memory space, the single memory block paradigm might breathe a new life into Prolog implementation.
I call this single memory block as stack because later readers will find that my garbage collection algorithm maintains the LIFO discipline of a stack. Under this hypothesis, we can find several advantages of the single stack scheme:

1. Single stack has better data locality than the heap/stack scheme.
2. Execution environment management can be greatly simplified.
3. There is no need to check binding directions.
4. Data objects are allocated dynamically in the natural (chronological) generation order of procedure calls. This facilitates both backtracking and garbage collection.
5. Two-stream unification can be easily implemented at the course-grain instruction level.
6. This paradigm facilitates the implementation of multiple threading and code migration in Mobile Agents systems.

In the following sections, I will presents the LVM memory organization, system registers, and stack allocation strategy. For the moment, we assume that memory is infinite and leave the garbage collection issues to Chapter 6.

3.1 Memory Organization and System Registers

The LVM allocates a block of storage (per thread for a multi-threaded virtual machine) to hold execution information. In this specification, we concentrate on the single thread virtual machine, however, issues to facilitate multi-threading will be considered as well. Fig. 1 gives the memory organization. Conceptually, the whole block is an addressable array of memory cells. The highest address is pointed by the trail base whereas the lowest address is given by the symbol base. The whole block is divided into four segments:

- **The symbol table** holds symbolic names of atoms, functors and predicates.
- **The code area** holds the executable code of a program.
- **The stack** holds all kinds of data, such as variables, structures, control information, choice points, etc., allocated during execution.
- **The trail** holds two types of addresses: addresses of bound variables which must be set unbound upon backtracking and addresses of roots which must be collected upon garbage collection.
Figure 1: Memory Organization
The memory regions for these segments does not need to be contiguous. The LVM does not provide the programmer or the user control over the initial sizes of these segments, instead, they are dynamically expanded or possibly contracted as required by the computation. To facilitate implementation, we assume that two separate blocks are allocated for each newly created thread: a *program block* for the symbol table and program code, and an *environment block* for the stack and the *trail*. The reason is that these two memory area might be expanded/contracted at different times, frequencies and strategies.

The LVM defines ten system registers (see Table 3.1) to represent the internal state of a (thread) computation. These registers will be used and maintained by the LVM run time system. Like their roles in the WAM, most registers have a straightforward purpose.

| PP: program pointer |
| AF: active frame pointer |
| CF: continuation frame pointer |
| ST: stack top pointer (successive put pointer) |
| GP: successive get/set pointer |
| BB: backtrack frame pointer |
| B0: cut register |
| GL: generation line |
| TT: trail pointer |
| NL: two stream unification branch flag |

**Table 3.1: System Registers**

In loading a program, symbol table and executable code are saved in the program block. A symbol hash table is also built in this block which will be used for symbolic *resolution* - a unique entity (offset) produced by the initial resolution of an atom entry will be shared by all the later resolutions of the same atom entry.

During execution, the environment block is used for all dynamical information. The stack is allocated upwards whereas the trail downwards, *i.e.*, they grow towards the opposite directions in the environment block. If the stack top pointer ST and the trail top pointer TT meet, the environment block is exhausted, and the LVM will dynamically expands the environment block through the underline OS system calls (such as *realloc*). The AF register always points to the top of the current activation frame, and is used as the base to determine the address of an indexed variable through the (*AF - index*) operation. The CF register is used to implement long return (from a chain of calls). The GP register is used for efficiency purpose: a sequence of unification (*get/set*) instructions can use GP to access their operands if these operands are located in contiguous memory cells. The BB register is for backtracking. It holds the root of a linked list of choices. The B0 is used for an
efficient implementation of *cut*. The GL register is novel and plays a key role in the LVM. It holds the *generation line* which divides the stack into two *dynamic* areas: a *generational space* and a *nursery space*. It is the key factor to determine whether a new generation should be created or a round of garbage collection should be invoked. The NL register is used for two stream unification, and will be discussed later.

### 3.2 Memory Allocation

The LVM does not classify local variables and global (heap) objects, nor does it use soft registers for passing arguments. All variables and structure instances are called *dynamic objects*. A single stack is used to hold activation information and dynamic objects. For a given clause, the total number of its dynamic objects is completely determined during compilation. When a procedure is called, an integral stack frame is allocated for the matching clause. This frame must be big enough to hold all dynamic objects required by the matching clause. In other words, the matching clause will never ask for more dynamic memory during its execution.

The LVM defines three kinds of stack frames: choice frame, data frame and activation frame (see Fig. 2). A choice frame is allocated when a choice point is met. The LVM handles backtracking the same way as the WAM does. A data frame, if necessary, is created upon a call to the type of clauses which do not have a continuation point (such as facts and chain clauses). An activation frame consists of a data portion and a three-cell expansion for control. Control information includes

![Figure 2: Stack Frames](image-url)
the continuation frame pointer CF, the continuation point CP and the current trail top TT. Frame allocation follows the chronological order of procedure calls. When a procedure returns, it leaves its frame deallocated.

Stack allocation is simple and straightforward. It is a push-only stack. The whole stack is divided by a stack top register ST: the space above ST is the free area and below is the occupied area. Frames can be allocated linearly simply by incrementing the ST pointer by the size of the frame to be allocated. The unit of stack allocation is based on 32-bit machine word, a slight modification is needed for porting the LVM to a 64-bit machine. Figure 3 shows the memory division and terminologies which will be used in our subsequent discussion.

The LVM does not do last call optimization, nor does it do environment trimming. There is no need to verify binding direction of two variables. There are no
unsafe variables. During execution, the value of the ST register might be changed by six possible operations:

**Frame allocation:** the ST register will be increased by a certain amount of size upon a frame allocation.

**Put operation:** a put instruction will push an argument on the top of the stack, and therefore increase ST by one.

**Garbage collection:** the ST register will be decreased after a round of garbage collection.

**Backtracking:** upon backtracking, ST will be reset to point to the latest choice point.

**Floating point operation:** constructing a floating point constant or computing a floating point expression will set up the value on the stack top, the therefore ST is increased by 2.

**Stack expanding/contracting:** after a round of environment block expanding or contracting, ST will be adjusted accordingly, so do other related registers.

Even though the full unification will use ST as the pointer of a temporary push-down stack to save pairs of terms to be unified. The original value of ST will be restored upon the completion of unification.
4 The LVM Instructions

A Prolog program is first compiled into the LVM assembly code. Producing an assembly-language program as output makes the process of code generation somewhat easier and the resulting code more readable.

4.1 Conventions

The general format of instructions is given in Table 4.1 where the opcode is a mnemonic name, and followed with zero or more operands. Each field in an instruction occupies a memory cell (a 32-bit machine word).

<table>
<thead>
<tr>
<th>opcode</th>
<th>operand1</th>
<th>operand2</th>
<th>...</th>
<th>operandn</th>
</tr>
</thead>
</table>

Table 4.1: Instruction Format

The reason of using a memory cell instead of bytecode for an opcode is to facilitate both case-switch engine and threaded-code engine. For a threaded-code engine, each opcode can be resolved into its corresponding execution entry during loading. The LVM assembly language uses the operand conventions in Table 4.2 to indicate the properties of the operands. If several operands share the same property in an instruction, they may be indexed with a sequence of number, such as e1, e2, ...

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Examples</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>sort/3.1</td>
<td>a program label</td>
</tr>
<tr>
<td>n</td>
<td>256</td>
<td>an integer or an index</td>
</tr>
<tr>
<td>c</td>
<td>abc</td>
<td>an atom</td>
</tr>
<tr>
<td>f</td>
<td>f/2</td>
<td>a functor with arity</td>
</tr>
<tr>
<td>r</td>
<td>3.1416</td>
<td>a floating point number</td>
</tr>
</tbody>
</table>

Table 4.2: Operand Conventions

It is worth to note that the operand conventions in this specification aim on easy to read and understand of an assembly code. In fact, a LVM compiler might modify this convention by resolving some operands further to facilitate the assembler. For example, a floating value might be broken into two machine words representing the high and low portion respectively; an atom might be replaced by the offset to symbol table; integer, atom, and functor operands have already tagged, etc.

During compilation, code segments must be labeled systematically. We use procedure name/arity to denote a procedure, and call this label as a pid (procedure identifier); and pid.i to indicate the ith clause of the procedure, and call this label as a cid (clause identifier); and finally, cid.(r/w flag).j’s to indicate various read/write stream entries inside that clause.
4.2 Internal Term Representations

As same as the WAM, the LVM represents a Prolog term as a tagged word: a memory cell which contains a tag field and a value field. The LVM uses the last three bits of a cell for the tag field, and defines different types and their fields in Table 4.3.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Tag</th>
<th>Meaning</th>
<th>Value Range</th>
<th>Value Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>REF</td>
<td>000</td>
<td>reference</td>
<td>30 bits</td>
<td>address to a term</td>
</tr>
<tr>
<td>LIS</td>
<td>001</td>
<td>list</td>
<td>29 bits</td>
<td>address to a list node</td>
</tr>
<tr>
<td>STR</td>
<td>010</td>
<td>structure</td>
<td>29 bits</td>
<td>address to a structure node</td>
</tr>
<tr>
<td>FLP</td>
<td>011</td>
<td>float ref</td>
<td>29 bits</td>
<td>address to a float value</td>
</tr>
<tr>
<td>REF1</td>
<td>100</td>
<td>reference</td>
<td>30 bits</td>
<td>address to a term</td>
</tr>
<tr>
<td>INT</td>
<td>101</td>
<td>integer</td>
<td>29 bits</td>
<td>integer value</td>
</tr>
<tr>
<td>CON</td>
<td>110</td>
<td>atom</td>
<td>9 + 20 bits</td>
<td>arity and symbol table index</td>
</tr>
<tr>
<td>FVL</td>
<td>111</td>
<td>float value</td>
<td>61 bits</td>
<td>floating point value</td>
</tr>
</tbody>
</table>

Table 4.3: Term Representations

Generally speaking, internal term representations can be classified into two groups: tag-on-value and tag-on-pointer. CON, INT and FVL belong to the first group, whereas REF, REF1, LIS, STR and FLP the second group.

An atom term uses an index to the symbol table as part of its value. This index is not a pointer, instead, it is a static offset throughout the computation. Each entry in the symbol table is stored by following the align-4 convention. As a consequence, the index of an entry is the word offset from the symbol table base. The symbol table might be expanded and moved to a different memory space during execution, but the original existing atoms and their offsets never change. The LVM guarantees that identical atoms must refer the same instance (offset) in the symbol table. Thus unification of two atoms becomes a simple comparison operation. In order to use CON term to cope with functors and procedure names, the LVM divides the value field into two parts: the first 9 bits to indicate the arity and the next 20 bits for the symbol table offset of an atom. Thus the maximum size of the symbol table is 4 Megabyte, and the maximum arity of a functor is 511.

An integer term carries its direct value, and integer arithmetic operations are performed in 29 bits precision, i.e., its value domain is $-2^{28}$ to $2^{28} - 1$.

Although floating point numbers are not very heavily used in typical Prolog programming (the argument is that Prolog is primarily a language for symbolic, non-numeric computation), we still need to consider efficient implementation of floating point arithmetic for a broad range of Prolog applications. Like most Prolog implementations, the LVM defines floating point values in double precision (IEEE 754 format), i.e., each floating value occupies two successive machine words (64 bits).
A minor change to the IEEE 754 format is that the last three bits are used by the LVM as the tag of a floating value, thus the *significant* is represented by 49 bits which should be adequate for most floating point computations. In the LVM, floating point values, no matter they are constants or dynamically generated, are stored in the stack. Next we should consider alignment. Alignment requirements imply that the address of any piece of dynamically allocated memory space must be able to serve as the address of any kind of data (terms). Different alignment might be required for different platforms, for example, align-8 for SPARC and align-4 for Intel x86 machines. Furthermore, the internal order of a double value differs on different platforms. Let \( w1 \) and \( w2 \) be two successive machine words, the SPARC family stores the first 32 bits of a double value in \( w1 \) and the next 32 bits in \( w2 \). The Intel x86 family, however, takes a reverse order: the last 32 bits in \( w1 \) and the first 32 bits in \( w2 \).

Here we simply follow the Intel x86’s alignment and storage order. The LVM defines that each floating point value is properly aligned to the 32-bit boundary, *i.e.*, align-4 as default, and each value is stored in the reverse word-wise order, *i.e.*, the first word holds the lower portion (part of significant plus FVL tag), and the next word holds the high portion (sign, exponent, and part of significant). One advantage is that no special alignment is required. In fact, align-4 applies to all LVM components, such as symbolic names, instructions, as well as terms. An Intel x86 LVM emulator can efficiently implement floating point arithmetic. However, an emulator on SPARC might involve some overhead dealing with floating point computations.

As a LVM term (one machine word) is not able to hold a floating value, we introduce floating reference term (with tag FLP) whose value field gives the absolute address of its floating point value. Thus, constructing a floating point term generally involves two steps: constructing the floating reference term by ST (the stack top pointer) in conjunction with the FLP tag and then pushing the floating value onto the stack (increasing ST by 2). Dynamically created floating point values are constructed in the same way. All floating values are subject to garbage collection. Furthermore, care should be taken when the stack is expanded to a new memory block, in this case, pointers in floating terms must be adjusted (relocated) accordingly.

The reason of creating floating constants onto the stack rather than putting them into the symbol table is that we want two memory blocks - the program block and the environment block - being manipulated as independently as possible. From the stack point of view, a tag-on-pointer term always refers to a stack address, and a tag-on-value term either refers to a direct value or a static index to the symbol table. On the other hand, from the program instruction point of view, a tag-on-pointer
operand is always an offset to the current activation frame, and a tag-on-value operand is always a direct value. As there are no mutual absolute term references across these two blocks, expanding/contracting the environment block has no impact on the program block. However, as the stack not only holds dynamically created terms but also control information such as CP’s and BP’s, moving the program block will cause some adjustment work on the stack. How to relocate pointers upon a block moving will be discussed in Chapter 5.

Like the WAM, an unbound variable is represented by a self-referential reference. Even through two tags, REF and REF1, are allocated to variable terms, the effective tag bits are the last two and the address portion is composed by the first 30 bits. Thus, a reference term is always a direct address to its binding.

Other tag-on-pointer terms, such as list term, structure term, or floating reference terms, use the first 29 bits as the node pointer. The actual address of the pointed node is calculated by masking away the tag and right-shifting 1 bit.

Tag-on-value terms can be constructed in compilation, whereas tag-on-pointer terms are created during execution. If the environment block is moved upon expansion, contraction or migration (for a Mobile Agents application), only tag-on-pointer terms need to be relocated.

4.3 Control Instruction Set

Control instructions are used to simulate different kinds of procedure invocations. Generally speaking, the control instruction set includes instructions for stack allocation, procedure invocation, control dispatching, and nondeterministic manipulation. They are given in Table 4.4.

Now, we discuss the basic formats of clause translation. We will use regular expressions to show the syntactical form of the LVM code generation where Kleene star * indicates zero or more occurrences, + represents at least one, and [...] denotes optional. The symbol \(\rightarrow\) is a relation of followed by.

1. Fact clause:
   \[\text{[allod]} \rightarrow \text{get} * \rightarrow \text{proceed}\]

2. Chain clause:
   \[\text{[allod]} \rightarrow \text{get} * \rightarrow \text{put} * \rightarrow \text{chaincall}\]

3. Rule clause: last goal is a user defined goal
   \[\text{alloca} \rightarrow \text{get} * \rightarrow \{ \text{put} * \rightarrow \text{call} \} + \rightarrow \text{put} * \rightarrow \text{lastcall}\]

4. Rule clause: last goal is a system predicate or an arithmetic operation
   \[\text{alloca} \rightarrow \text{get} * \rightarrow \{ \text{put} * \rightarrow \text{call} \} + \rightarrow \text{return}\]
5. Query:

\[ \{ \text{put} \ * \rightarrow \text{call} \} + \rightarrow \text{finish} \]

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operand</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| alloa    | n1, n2  | allocate an activation frame  
|          |         | n1: # of arguments           
|          |         | n2: # of additional cells    |
| allod    | n1, n2  | allocate a data frame         
|          |         | n1: # of arguments           
|          |         | n2: # of additional cells    |
| call     | e       | procedure call                
|          |         | e: callee’s procedure entry   |
| lastcall | e       | last call                     
|          |         | e: callee’s procedure entry   |
| chaincall| e       | chain call                    
|          |         | e: callee’s procedure entry   |
| proceed  |         | proceed to an ancestor       |
| return   |         | return to parent              |
| finish   |         | execution terminate           |
| try      | n, e    | allocate a choice frame and try this clause  
|          |         | n: # of arguments            
|          |         | e: next alternative entry    |
| retry    | n, e    | retry this clause             
|          |         | n: # of arguments            
|          |         | e: next alternative entry    |
| trust    |         | trust this clause             |
| switch   | n, e1, e2, e3, e4 | switch wrt the nth argument  
|          |         | n: index of the argument     
|          |         | e1: variable entry           
|          |         | e2: constant entry           
|          |         | e3: list entry               
|          |         | e4: structure entry          |
| hashing  | n, e    | branch wrt the nth argument’s hash value  
|          |         | n: index of the argument     
|          |         | e: hash table entry          |

Table 4.4: Control Instructions

Readers who are familiar with the WAM might find that the clause translation formats are striking resemblances between the WAM and the LVM. From the control aspect, it is true that the LVM mimics the WAM with a very minor difference:

22
the stack allocation. First, the WAM passes calling arguments through a set of soft registers, whereas the LVM passes the calling arguments directly to the callee’s data (or activation) frame. This scheme was first adopted by the NTOAM [27]. The advantage of this scheme is to eliminate the bottle neck of the procedure invocation. The disadvantage, however, is that we have to allocate a stack frame for each invocation whereas the WAM can invoke a clause without allocating an environment if the clause can be executed using only register operations. Secondly, the LVM should allocate memory space for all dynamic objects of the matching clause. The number of dynamic objects is the amount of flattened terms of that clause.

Here we define a flattened term as an equation of the form:

\[ X_i = \textit{a constant or a variable} \]

or

\[ X_i = f(X_{i1}, ..., X_{in}), (n \geq 0) \]

where the \( X_i \)'s are all distinct new variable names.

Frame allocation instructions require two operands: the number of arguments and the number of additional cells. For a data frame, the amount of flattened terms contributes the number of additional cells. However, for an activation frame, we need to add three more cells to hold control information. The total size of an allocated frame is the sum of the two operands given by a frame allocation instruction. Let us consider Example 4.1:
Example 4.1: \( \text{eq}(\text{if}(X, \text{if}(Y, t, f), f), \text{and}(X, Y)) \).

In Example 4.1, terms are flattened to eleven distinct variables which is the exact number of additional cells required in executing \( \text{eq}/2 \). Since the clause is a fact with two arguments, an \textit{alloca 2, 11} will be generated. The frame to be allocated is given by Fig. 4, where \( X_0 \) to \( X_{10} \) represent flattened terms, \( X_{11} \) and \( X_{12} \) are two calling arguments. After frame allocation, register AF will point to the top cell, and it serves as the base register to locate an indexed terms by (AF - index).

The next example gives a more precise analysis of memory allocation.

Example 4.2:

\[
p(f(X, Y), a) :-
\begin{align*}
  q(h(b, g(Y)), c), \\
  r(123, i(a, b, c)).
\end{align*}
\]

Let \( A_i \) represent an input argument, \( O_i \) an output argument, \( T_i \) a memory cell holding either a function or a term, and symbol \( @T_i \) indicate an object at \( T_i \)'s location. Terms in Example 4.2 are flattened and counted in depth-first order as follows:

\[
\begin{array}{lcl}
  \text{Example 4.2: Memory allocation} \\
  A_1 = @T_1(T_2, T_3) & \% f(X, Y) \\
  T_1 = f/2 \\
  T_2 = X \\
  T_3 = Y \\
  A_2 = a & \% a \\
  O_1 = @T_4(T_5, T_6) & \% h(b, g(Y)) \\
  T_4 = h/2 \\
  T_5 = b \\
  T_6 = @T_7(T_8) \\
  T_7 = g/1 \\
  T_8 = @T_3 \\
  O_2 = c & \% c \\
  O_3 = 123 & \% 123 \\
  O_4 = @T_9(T_{10}, T_{11}, T_{12}) & \% i(a, b, c) \\
  T_9 = i/3 \\
  T_{10} = a \\
  T_{11} = b \\
  T_{12} = c
\end{array}
\]

After this analysis, we could easily find that an \textit{alloca 2, 15} should be coded as this clause requires two input arguments \( A_1 \) and \( A_2 \), 12 additional memory cells.
for holding its internal term nodes, i.e., cells $T_1$ to $T_{12}$, plus three cells for control information. No memory is reserved for an output argument. Corresponding $O_i$'s will be pushed on to the stack top when a procedure is called. In next section reader will find that each $A_i$ will be coded to a get instruction and its subterms (if any) will be handled by two streams of get/set instructions; each $O_i$ will be coded to a put instruction and its subterms (if any) will be created by a sequence of set instructions.

A rule clause must create an activation frame as its invocation environment. In the third and the fourth regular expressions, the first instruction alloa could not be omitted because a rule clause needs at least three cells for holding control information. If the last goal is a user defined goal, lastcall is used to set up a continuation environment. However, this instruction does not implement LCO (last call optimization). Another possible case is that the last several goals are builtin predicates or arithmetic operations. If this happens, a return is used to pass control back to its caller.

Instructions switch and hash are used to dispatch control based on the type/value of a specified term. Variants of these instructions are lastswitch and lasthash, i.e., they make a control jump depending on the last argument. The reason of doing so is straightforward: the last argument is on the top of the stack when a procedure is called. However, this constraints that dispatching instructions depending on the last argument must be used before any frame allocation activities, because the stack top will be changed by a frame allocation instruction.

Like the WAM, the LVM generates a

\[
\text{try} \rightarrow \text{retry}^* \rightarrow \text{trust}
\]

sequence to handle a nondeterministic procedure. Note, this sequence is equivalent to the WAM's

\[
\text{try_me_else} \rightarrow \text{retry_me_else}^* \rightarrow \text{trust_me}
\]

A choice frame is allocated by the try instruction. This frame sits on the top of a set of arguments passed to this procedure. Hence, different from the WAM which has to save argument registers, the LVM has to reput these arguments on the top of the stack in order to invoke the trying clause. A trust instruction will deallocate the choice frame. As a consequence, the arguments are exposed to the last alternative clause. Appendix A gives the implementation details of control instructions.

4.4 Unification Instruction Set

Most unification operations could be simplified into matches and assignments. Thus, each unification instruction is annotated with a special type. Generally speaking, there are four types of instructions dedicated to unification and term construction:
**put:** used by the caller to set up calling arguments before issuing a procedure call. They must be arranged to follow the occurrence order of arguments, *i.e.*, the 1st argument must be put first, then the 2nd argument, ..., and finally the last argument. The operand (if any) of a put instruction is called the *annotated term*. The behaviour of a put is to push its annotated term onto the top of the execution stack.

**get:** used by the callee to initiate unification. The first operand of a get instruction is the *source term*, and the rest operands specify the *annotated term*. A get operation triggers backtracking if the source term has failed in unifying with the annotated term. They form the read stream in the so-called two stream unification. Some get operations, namely getlist and getstr, might transfer from a read stream to a write stream if the source term is an unbound variable.

**set:** used by the caller to construct new instances of compound terms, or used by the callee to implement write stream unification. The first operand of a set instruction is the *destination term*, and the second operand (if any) specify the *annotated term*. They are efficient because the annotated term is constructed on the destination term by destructive assignment.

**branch:** used to transfer control from a write stream back to a read stream.

Let CON(c) be an atom term, INT(n) an integer term, and FVL(r) a double precision value. In order to distinguish functors from constants, we use FUN(f) to denote a functor. Furthermore, let Cell(n) represent a memory cell at nth position of the current frame, REF(n) be a reference term, FLP(n) a floating reference term, LIS(n) a list term, and STR(n) a structure term, such that each term is formed by the address of Cell(n) in conjunction of the annotated tag, we define unification instructions with each a brief description in Table 4.5.

For the sake of efficiency, the LVM expands the basic set to include more variants. For example, a sequence of putval instructions can be folded into a single putval instruction. In addition, get and set instructions require the first operand as the source/destination term, *i.e.*, an index indicating where to get the term for unification or where to construct the annotated term. However, it is common that a sequence of get or set instructions specify these indices in continuous, decreasing order. Having such a term in every instruction is thus a wasting. The LVM introduces a register GP to keep track of the first operand address of the most recent get or set instruction, and the subsequent instructions will be replaced by cget or cset instructions which have their source/destination indices omitted, provided that their omitted indices obey the continuous, decreasing order, and very importantly, there is no break-in point labeled to these instructions.
<table>
<thead>
<tr>
<th>Operator</th>
<th>Operand</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>putvoid</td>
<td></td>
<td>push REF(ST) - a void variable</td>
</tr>
<tr>
<td>putvar</td>
<td>n</td>
<td>push REF(n) and make Cell(n) unbound</td>
</tr>
<tr>
<td>putval</td>
<td>n</td>
<td>push Cell(n)'s value</td>
</tr>
<tr>
<td>putcon</td>
<td>c</td>
<td>push CON(c)</td>
</tr>
<tr>
<td>putint</td>
<td>n</td>
<td>push INT(n)</td>
</tr>
<tr>
<td>putlist</td>
<td>n</td>
<td>push LIS(n)</td>
</tr>
<tr>
<td>putstr</td>
<td>n</td>
<td>push STR(n)</td>
</tr>
<tr>
<td>getval</td>
<td>n1, n2</td>
<td>unify Cell(n1) and Cell(n2)</td>
</tr>
<tr>
<td>getcon</td>
<td>n, c</td>
<td>unify Cell(n) and CON(c)</td>
</tr>
<tr>
<td>getint</td>
<td>n1, n2</td>
<td>unify Cell(n1) and INT(n2)</td>
</tr>
<tr>
<td>getfloat</td>
<td>n, r</td>
<td>unify Cell(n) and FVL(r) or push FVL(r) onto the stack</td>
</tr>
<tr>
<td>getlist</td>
<td>n1, n2, n3, e</td>
<td>unify Cell(n1) with LIS(n2) n3: nesting level e: branch if Cell(n1) is an unbound</td>
</tr>
<tr>
<td>getvlist</td>
<td>n1, n2</td>
<td>unify Cell(n1) with LIS(n2) or set Cell(n1) to LIS(n2) and set Cell(n2)-Cell(n2+1) to unbound</td>
</tr>
<tr>
<td>getstr</td>
<td>n1, n2, n3, e, f</td>
<td>unify Cell(n1) with STR(n2) n3: nesting level e: branch if Cell(n1) is an unbound CON(f): functor of the structure</td>
</tr>
<tr>
<td>setvar</td>
<td>n</td>
<td>set Cell(n) to REF(n)</td>
</tr>
<tr>
<td>setval</td>
<td>n1, n2</td>
<td>set Cell(n1) to REF(n2)</td>
</tr>
<tr>
<td>setcon</td>
<td>n, c</td>
<td>set Cell(n) to CON(c)</td>
</tr>
<tr>
<td>setint</td>
<td>n1, n2</td>
<td>set Cell(n1) to INT(n2)</td>
</tr>
<tr>
<td>setfloat</td>
<td>n, r</td>
<td>set Cell(n) to FLP(ST) and push FVL(r) onto the stack (trailing)</td>
</tr>
<tr>
<td>setlist</td>
<td>n1, n2</td>
<td>set Cell(n1) to LIS(n2)</td>
</tr>
<tr>
<td>setstr</td>
<td>n1, n2</td>
<td>set Cell(n1) to STR(n2)</td>
</tr>
<tr>
<td>setfun</td>
<td>n, f</td>
<td>set Cell(n) to FUN(f)</td>
</tr>
<tr>
<td>branch</td>
<td>e1, e2, e3, e4</td>
<td>branch wrt NL's value</td>
</tr>
</tbody>
</table>

Table 4.5: Unification Instructions

Example 4.3 shows the translation of a simple Prolog clause. The activation frame layout is given as the comments on the LVM code. In this example, four successive set instructions following the getcon 7, a could have their destination terms indexed in the order of 6, 5, 4, 3, and thus they are substituted by corresponding cset instructions.
Example 4.3:

\[ p(Y, a) :\]
\[ q(X, Y, f(b, X, Y)), \]
\[ r(X, Y). \]

---

**Example 4.3: the LVM code**

```
procedure p/2  % STACK:
p/2:       alloa 2, 7  % 0: CF
          getcon 7, a  % 1: CP
csetfun  f/3  % 2: TT
csetcon  b     % 3: REF(@8)
csetvar  % 4: REF(@4)
csetval  8     % 5: CON(b)
put2val  4, 8  % 6: FUN(f/3)
putstr   6     % 7: A2
call     q/3  % 8: A1
put2val  4, 8
lastcall r/2
```

As having mentioned before, arguments to a procedure are passed through a sequence of `put` instructions. These arguments constitute the lower portion of the callee’s activation frame and can be accessed by the callee directly through the \((AF - index)\) addressing mode. It looks like that a sequence of `get_var` instructions (such as in the WAM) has been executed. Hence, there is no need to have a `getvar` instruction in the LVM instruction set. To illustrate this, let us examine Example 4.4. This example is translated to both LVM code and WAM code. To be easily understood, I use symbolic names instead of memory indices for variables, and \(R_i\) for registers.

Example 4.4:

\[ p(X, Y) :\]
\[ q(Y), \]
\[ r(Y, X). \]
Example 4.4: LVM vs. WAM code

<table>
<thead>
<tr>
<th>LVM code</th>
<th>WAM code</th>
</tr>
</thead>
<tbody>
<tr>
<td>p/2: alloa</td>
<td>p/2: alloc</td>
</tr>
<tr>
<td>putval Y</td>
<td>get_var R1, X</td>
</tr>
<tr>
<td>call q/1</td>
<td>get_var R2, Y</td>
</tr>
<tr>
<td>putval Y</td>
<td>put_val R1, Y</td>
</tr>
<tr>
<td>putval X</td>
<td>call q/1</td>
</tr>
<tr>
<td>lastcall r/2</td>
<td>put_val R2, X</td>
</tr>
<tr>
<td>dealloc</td>
<td>execute r/2</td>
</tr>
</tbody>
</table>

Before passing a tag-on-pointer argument (such as STR, LIS or FLP) to a call, the node referred by that argument must be constructed first. As illustrated by Example 4.3, the structure node f(b, X, Y) has been set in the caller’s stack before issuing the putstr instruction which pushes the third argument (the structure term) to the procedure q/3.

On the other hand, if a structure term occurs in the head of a procedure, unification operation takes place. In general, unification of a pair of structure terms could be simplified into matches (read-mode) and assignments (write-mode). When the input argument carries a structure instance (node), its functor will be matched against the functor of the head structure, and a sequence of unification operations will be followed to unify every pair of nested arguments if two functors are identical. When the input argument comes to stand for a free variable, however, a concrete instance of the head structure must be constructed. To cope with these cases, the WAM unification instructions have two modes of execution. The current mode is stored in a global mode register, which is set by get_list and get_structure, and tested in all unify instructions. As the write mode is not propagated to subterms, unification of subterms may involve superfluous dereferences, trail checks and bindings.

On the other hand, a so called two stream unification algorithm separates works in read mode and write mode into two streams of instructions - one stream for selection and simple unification, and another stream for compound term construction - with conditional jumps between them. Demoen[28] shown how to improve WAM instructions to facilitate two stream unification. Roy[15] gives a good survey of this algorithm and Haygood[25] presented an example of two stream code generated by the SICStus compiler. They both pointed out that this scheme is elegant for compiling unification and much more efficient than the WAM for native code implementation.
With the idea of selective execution in mind, I designed the LVM instructions which fully support two stream unification at the coarse-grain level. To illustrate this, let us examine how `getstr` is implemented.

```c
getstr(n1, n2, n3, e, f) {
    t = dereference(Cell(n1));
    if (is_ref(t)) {
        GP = Cell(n2); // assign continuing set pointer
        *t = STR(GP);
        trail(t);
        NL = n3; // assign nesting level to register NL
        goto e; // jump to write stream
    } else if (is_str(t) && (t->functor == CON(f))) {
        for (i = 1; i <= arity(f); i++) {
            Cell(n2 + i) = t->argument[i];
        }
        GP = Cell(n1); // assign continuing get pointer
        goto next_instruction; // remain in read stream
    } else backtrack;
}
```

The `getstr` instruction first checks the tag of the dereferenced source term. If it is an unbound, it is bound to the annotated structure, then operand `n3` is assigned to register NL, annotated address is assigned to GP, and control is transferred to the write stream labeled by `e` where a sequence of `cset` instructions will construct the annotated structure instance. Operand `n3` is an integer representing the nesting level of the annotated term. On the other hand, if the source term is a structure and its functor agrees with the annotated functor `f`, arguments of the source term are copied to the annotated frame space. After copying, execution continues to the next instruction and arguments of the matching structure are ready for further selection and unification, i.e., execution remains in the read stream. This is fundamentally different from the WAM’s `get_structure` instruction. In the WAM, even if the unification continues in read mode, a sequence of `unify_variable` instructions must be coded in order to set register variables to arguments of the matching structure for subsequent unification. In contrast with the WAM, the LVM’s `getstr` instruction sets up arguments of the matching structure as part of its own function. Therefore, for a given compound term, the code size of its read stream is in general less than number of its flattened subterms. Furthermore, the LVM has no `unify`-like instructions at all. The set of `get` instructions is sufficient for coding read stream
programs. A shortcoming of this scheme is that the LVM must allocate frame cells for each structure instance no matter the instance will be selected (read-mode) or constructed (write-mode), therefore increasing the consumption of the stack.

Now going back to our Example 4.1:

**Example 4.1:** eq(if(X, if(Y, t, f), f), and(X, Y)).

The two stream unification code looks thus:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Eq/2</th>
<th>% Directive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allod</td>
<td>2, 11</td>
<td>% Eq/2 entry</td>
</tr>
<tr>
<td>Getstr</td>
<td>12, 10, 0, if/3, eq/2.w.0</td>
<td>% Unify A1 with if/3</td>
</tr>
<tr>
<td>Getstr</td>
<td>8, 6, 1, if/3, eq/2.w.1</td>
<td>% Unify nesting if/3</td>
</tr>
<tr>
<td>Getcon</td>
<td>4, t</td>
<td>% Unify t in nesting if/3</td>
</tr>
<tr>
<td>Cgetcon</td>
<td>f</td>
<td>% Unify f in nesting if/3</td>
</tr>
<tr>
<td>Eq/2.r.0: Getcon</td>
<td>7, f</td>
<td>% Unify f in if/3</td>
</tr>
<tr>
<td>Eq/2.r.1: Getstr</td>
<td>11, 2, 0, and/2, eq/2.w.2</td>
<td>% Unify A2 with and/2</td>
</tr>
<tr>
<td>Getval</td>
<td>1, 9</td>
<td>% Unify X in and/2</td>
</tr>
<tr>
<td>Cgetval</td>
<td>5</td>
<td>% Unify Y in and/2</td>
</tr>
<tr>
<td>Proceed</td>
<td>1</td>
<td>% Write stream</td>
</tr>
<tr>
<td>Eq/2.w.0: Csetfun</td>
<td>if/3</td>
<td>% Set if/3 functor</td>
</tr>
<tr>
<td>Csetvar</td>
<td></td>
<td>% Set X in if/3</td>
</tr>
<tr>
<td>Csetstr</td>
<td>6</td>
<td>% Set nesting if/3 in if/3</td>
</tr>
<tr>
<td>Csetcon</td>
<td>f</td>
<td>% Set f in if/3</td>
</tr>
<tr>
<td>Eq/2.w.1: Csetfun</td>
<td>if/3</td>
<td>% Set nesting if/3 functor</td>
</tr>
<tr>
<td>Csetvar</td>
<td></td>
<td>% Set Y in nesting if/3</td>
</tr>
<tr>
<td>Csetcon</td>
<td>t</td>
<td>% Set t in nesting if/3</td>
</tr>
<tr>
<td>Csetcon</td>
<td>f</td>
<td>% Set f in nesting if/3</td>
</tr>
<tr>
<td>Branch</td>
<td>eq/2.r.1, eq/2.r.0, fail, fail</td>
<td>% If NL = 0 to eq/2.r.1</td>
</tr>
<tr>
<td>Eq/2.w.2: Csetfun</td>
<td>and/2</td>
<td>% Set and/2 functor</td>
</tr>
<tr>
<td>Csetval</td>
<td>9</td>
<td>% Set X in and/2</td>
</tr>
<tr>
<td>Csetval</td>
<td>5</td>
<td>% Set Y in and/2</td>
</tr>
<tr>
<td>Proceed</td>
<td>1</td>
<td>% Write stream</td>
</tr>
</tbody>
</table>

At some stage a write stream needs to transfer control back to a certain point of a read stream. This is done by the `branch` instruction. One `branch` instruction
may cope with dispatching points up to four nesting levels. A sequence of \texttt{branch}
instructions is able to cope with dispatching points of arbitrary nesting levels.

\texttt{branch(e1, e2, e3, e4)\{}
\begin{verbatim}
if (NL < 4) goto e_{NL}; // back to read stream
else {
    NL = NL - 4;
    goto next_instruction; // next branch
}
\end{verbatim}

In the original WAM, there are no \texttt{set} instructions. Ait-Kaci introduced \texttt{set}
instructions in his WAM introductory book\cite{4}. He indicates that all \texttt{set} instructions
are equivalent to \texttt{unify} instructions in write mode, and more efficient to construct
terms after \texttt{put} instructions. In other words, their \texttt{set} instructions are used \textit{before}
unification taking place. The LVM goes one step forward: \texttt{set} instructions are not
only used in the data preparing phase of unification before matching work comes
into play, but also used in the data constructing phase when unification falls into
write mode.

Generating two stream code looks to be quite complicated. The compiler needs to
arrange code segments being properly labeled and conditional jumps being properly
inserted. Detailed examples were given in \cite{28} which shown the idea of treating the
deeper nested structures later than the less deeply nested ones. For example, to
collapse the most jumps, the compiler should reorder the arguments of all subterms
to unify the most complex subterms last, or adjust the structure pointer to avoid
jump backs from a left-nested structure. The reordering of the arguments and
adjusting structure pointer complicates the code generation. For instance, the write
stream must depend on the read stream code for a proper unification order. On the
other hand, the LVM generates two stream code in a natural, simple way. There
are no structure pointer adjustment and argument reordering. Two stream code are
generated independently from the parse tree of the clause. Briefly, the compiler visits
term nodes in depth-first manner for both streams. It generates a \texttt{get} instruction
for each node along the traversing to form the read stream. In generating write
stream, however, \texttt{set} instructions are coded in breadth-first only if the visited node
is a compound term. This algorithm not only sets up labels and jumps, but also
generates instructions with respect to properties of terms, such as whether a variable
term is the first occurrence, or whether a \texttt{get} operation can be replaced by a \texttt{cget},
\texttt{etc.} It does not do argument reordering. Most jumps can be folded into a single
\texttt{branch} instruction. The code size is linear in term size. More precisely, for a
compound term composed by N flattened terms, its two stream code size will not
exceed $2 * N$. See Chapter 5 for a detailed discussion.
As we mentioned before, constructing a floating point literal requires two steps: one for set up the double precision value and another for set up the float reference term. This looks like the constructing process of a structure/list term in which one instruction is used to set up the term and a set of instructions are coded for setting up the structure/list node. However, there is a slight difference: a floating point term has no pre-allocated cells for holding its node. The LVM compiler only allocates one FLP-type cell (if necessary) for a floating point literal, and this value will be created dynamically (on the top of the current stack) during execution. As the constructing process of a floating point value involves stack-related operation, we cannot have a putfloat instruction. The reason is straightforward: a sequence of put operations must maintain the argument occurrence order and follow the one term per cell convention, that is, no other stack-related operations are allowed during the argument setup process. Thus, we must use setfloat first, and then putval to deliver a floating point argument. Example 4.5 shows how floating constants are treated when used in the clause head and goals.

Example 4.5:

\[ p(3.579, X) :- \]
\[ \q (\text{foo}(2.468, \text{bar}), X), \]
\[ \text{r}(X, 312.4). \]

---

**Example 4.5: the LVM code**

<table>
<thead>
<tr>
<th>Procedure</th>
<th>p/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p/2 )</td>
<td></td>
</tr>
<tr>
<td>\text{alloca}</td>
<td>2, 7</td>
</tr>
<tr>
<td>( % 0: \text{CF} )</td>
<td></td>
</tr>
<tr>
<td>\text{getfloat}</td>
<td>8, 3.579</td>
</tr>
<tr>
<td>( % 1: \text{CP} )</td>
<td></td>
</tr>
<tr>
<td>\text{setfun}</td>
<td>6, foo/2</td>
</tr>
<tr>
<td>( % 2: \text{TT} )</td>
<td></td>
</tr>
<tr>
<td>\text{csetfloat}</td>
<td>2.468</td>
</tr>
<tr>
<td>( % 3: \text{FLP}(312.4) )</td>
<td></td>
</tr>
<tr>
<td>\text{csetcon}</td>
<td>bar</td>
</tr>
<tr>
<td>( % 4: \text{CON}(\text{bar}) )</td>
<td></td>
</tr>
<tr>
<td>\text{putstr}</td>
<td>6</td>
</tr>
<tr>
<td>( % 5: \text{FLP}(2.468) )</td>
<td></td>
</tr>
<tr>
<td>\text{putval}</td>
<td>7</td>
</tr>
<tr>
<td>( % 6: \text{FUN}\text{foo}/2 )</td>
<td></td>
</tr>
<tr>
<td>\text{call}</td>
<td>q/2</td>
</tr>
<tr>
<td>( % 7: \text{A2} )</td>
<td></td>
</tr>
<tr>
<td>\text{setfloat}</td>
<td>3, 312.4</td>
</tr>
<tr>
<td>( % 8: \text{A1} )</td>
<td></td>
</tr>
<tr>
<td>\text{put2val}</td>
<td>7, 3</td>
</tr>
<tr>
<td>\lastcall</td>
<td>r/2</td>
</tr>
</tbody>
</table>
4.5 Arithmetic Operations and Builtins

The LVM adopts three-address scheme for arithmetic operations. For an instruction of the form:

\[ \text{op n1, n2, n3} \]

the corresponding operation is defined by:

\[ \text{VAL(n1) op VAL(n2) } \rightarrow \text{Cell(n3)} \]

where \( \text{VAL(n)} \) represents either an integer value or a floating point value dereferenced from a floating reference term at the \( n \)th position of the current stack frame, and \( \text{Cell(n3)} \) must be unbound, because the result will be assigned destructively. A brief description of arithmetic instructions is given in Table 4.6.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operands</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>n1, n2, n3</td>
<td>VAL(n1) + VAL(n2) → Cell(n3)</td>
</tr>
<tr>
<td>sub</td>
<td>n1, n2, n3</td>
<td>VAL(n1) - VAL(n2) → Cell(n3)</td>
</tr>
<tr>
<td>mul</td>
<td>n1, n2, n3</td>
<td>VAL(n1) × VAL(n2) → Cell(n3)</td>
</tr>
<tr>
<td>div</td>
<td>n1, n2, n3</td>
<td>VAL(n1) ÷ VAL(n2) → Cell(n3)</td>
</tr>
<tr>
<td>minus</td>
<td>n1, n2</td>
<td>-VAL(n1) → Cell(n2)</td>
</tr>
<tr>
<td>lshift</td>
<td>n1, n2, n3</td>
<td>VAL(n1) &gt;&gt; VAL(n2) → Cell(n3)</td>
</tr>
<tr>
<td>rshift</td>
<td>n1, n2, n3</td>
<td>VAL(n1) &lt;&lt; VAL(n2) → Cell(n3)</td>
</tr>
<tr>
<td>and</td>
<td>n1, n2, n3</td>
<td>VAL(n1) &amp; VAL(n2) → Cell(n3)</td>
</tr>
<tr>
<td>or</td>
<td>n1, n2, n3</td>
<td>VAL(n1)</td>
</tr>
<tr>
<td>eor</td>
<td>n1, n2, n3</td>
<td>VAL(n1) ∧ VAL(n2) → Cell(n3)</td>
</tr>
<tr>
<td>inc</td>
<td>n1, n2</td>
<td>VAL(n1) + 1 → Cell(n2)</td>
</tr>
<tr>
<td>dec</td>
<td>n1, n2</td>
<td>VAL(n1) - 1 → Cell(n2)</td>
</tr>
<tr>
<td>mod</td>
<td>n1, n2, n3</td>
<td>VAL(n1) % VAL(n2) → Cell(n3)</td>
</tr>
<tr>
<td>int</td>
<td>n1, n2</td>
<td>integer(VA(n1)) → Cell(n2)</td>
</tr>
<tr>
<td>float</td>
<td>n1, n2</td>
<td>float(VA(n1)) → Cell(n2)</td>
</tr>
<tr>
<td>is</td>
<td>n1, n2</td>
<td>VAL(n1) = VAL(Cell(n2)) or VAL(n1) → Cell(n2)</td>
</tr>
</tbody>
</table>

Table 4.6: Arithmetic Instructions

Three-address scheme is less efficient than the scheme which uses soft registers. In addition, it requires that the LVM compiler allocate enough cells to hold intermediate results. An intermediate cell can be reused if its value is no longer being accessed in the future computation of the clause. Readers might ask how the compiler allocate cells for intermediate floating point values, as sometimes the compiler even could not determine whether an expression will generate floating point
values or not. The LVM takes a simple solution: the compiler allocates one cell for an intermediate result. If the result is an integer, then the cell will hold the integer term; if the result is a floating point value, then the cell will hold the floating reference term which points to the floating value constructed on the top of the current stack. In other words, the LVM compiler does not allocate memory cells for floating point values at all, they are dynamically formed during execution. This strategy facilitates mixed-type arithmetic computation and makes the implementation of the compiler simpler. Example 4.6 shows the translation of a clause with mixed-type calculation.

**Example 4.6:**

\[ p(X, Y, Z) :\]

\[ X \text{ is } 3.579 + (Y 	imes 2 - Z). \]

<table>
<thead>
<tr>
<th><strong>Example 4.6: the LVM code</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>procedure</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>p/3: alloid</td>
</tr>
<tr>
<td>setint</td>
</tr>
<tr>
<td>mul</td>
</tr>
<tr>
<td>sub</td>
</tr>
<tr>
<td>setfloat</td>
</tr>
<tr>
<td>add</td>
</tr>
<tr>
<td>is</td>
</tr>
<tr>
<td>proceed</td>
</tr>
</tbody>
</table>

To execute this clause, one additional frame cell is required, *i.e.*, Cell(0) for holding intermediate values. How many floating point values will be created on the top of the current stack depends on the type of input arguments Y and Z. For example, three floating point values will be constructed if Y is a floating point reference. During calculation, Cell(0) is reused to refer these intermediate values and Cell(1) is reused to refer the floating constant 3.579. However, memory cells occupied by dynamically created floating point values can not be reused, they remain in the stack until a round of garbage collection.

Some arithmetic instructions apply to integer-type operands only, such as bitwise operations, mode operation, and incremental/decremental operations. In addition, programmer may explicitly convert operand type. Type conversion is implemented by \textbf{int} and \textbf{float} instructions.
The LVM provides a set of branch instructions based on arithmetic comparison. For an instruction of the form:

\[
\text{op n1, n2, e}
\]

the corresponding operation is defined by:

\[
\begin{align*}
\text{if (VAL(n1) op VAL(n2)) goto e} \\
\text{else goto next_instruction}
\end{align*}
\]

Table 4.7 gives a brief description of arithmetic comparison instructions.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operands</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifeq</td>
<td>n1, n2, e</td>
<td>succeeds iff VAL(n1) =:= VAL(n2)</td>
</tr>
<tr>
<td>ifne</td>
<td>n1, n2, e</td>
<td>succeeds iff VAL(n1) =:= VAL(n2)</td>
</tr>
<tr>
<td>iflt</td>
<td>n1, n2, e</td>
<td>succeeds iff VAL(n1) &lt; VAL(n2)</td>
</tr>
<tr>
<td>ifle</td>
<td>n1, n2, e</td>
<td>succeeds iff VAL(n1) &lt;= VAL(n2)</td>
</tr>
<tr>
<td>ifgt</td>
<td>n1, n2, e</td>
<td>succeeds iff VAL(n1) &gt; VAL(n2)</td>
</tr>
<tr>
<td>ifge</td>
<td>n1, n2, e</td>
<td>succeeds iff VAL(n1) &gt;= VAL(n2)</td>
</tr>
<tr>
<td>ifeq0</td>
<td>n, e</td>
<td>succeeds iff VAL(n) =:= 0</td>
</tr>
<tr>
<td>ifne0</td>
<td>n, e</td>
<td>succeeds iff VAL(n) =:= 0</td>
</tr>
<tr>
<td>iflt0</td>
<td>n, e</td>
<td>succeeds iff VAL(n) &lt; 0</td>
</tr>
<tr>
<td>ifle0</td>
<td>n, e</td>
<td>succeeds iff VAL(n) &lt;= 0</td>
</tr>
<tr>
<td>ifgt0</td>
<td>n, e</td>
<td>succeeds iff VAL(n) &gt; 0</td>
</tr>
<tr>
<td>ifge0</td>
<td>n, e</td>
<td>succeeds iff VAL(n) &gt;= 0</td>
</tr>
</tbody>
</table>

Table 4.7: Arithmetic Comparison Instructions

Appendix C shows implementation of current arithmetic instructions. Efficient arithmetic is left open to a later version.

Built-in predicates extend the expressive power of Prolog. Some built-in predicates are outside the scope of first-order-logic because they retrieve system state or change the states of objects. From the implementation point of view, built-in predicates are divided into three classes:

**Emulated builtins**: implemented by the emulator directly for efficiency.

**Compiled builtins**: predefined in Prolog and initialized when the LVM system starts to run.

**System builtins**: implemented by stand-alone functions. The reason is flexibility and simplicity.

Meta-logical predicates are emulated directly. Those include term type testing and term comparison. Table 4.8 gives instructions for emulated builtins.
Table 4.8: Term Testing and Comparison Instructions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operands</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>isatom</td>
<td>n, e</td>
<td>succeeds if Cell(n) is an atom</td>
</tr>
<tr>
<td>isatomic</td>
<td>n, e</td>
<td>succeeds if Cell(n) is atomic</td>
</tr>
<tr>
<td>isfloat</td>
<td>n, e</td>
<td>succeeds if Cell(n) is a float reference</td>
</tr>
<tr>
<td>isinteger</td>
<td>n, e</td>
<td>succeeds if Cell(n) is an integer</td>
</tr>
<tr>
<td>isnumber</td>
<td>n, e</td>
<td>succeeds if Cell(n) is a number</td>
</tr>
<tr>
<td>isnonvar</td>
<td>n, e</td>
<td>succeeds if Cell(n) is not a variable</td>
</tr>
<tr>
<td>isvar</td>
<td>n, e</td>
<td>succeeds if Cell(n) is a variable</td>
</tr>
<tr>
<td>islist</td>
<td>n, e</td>
<td>succeeds if Cell(n) is a list</td>
</tr>
<tr>
<td>isstr</td>
<td>n, e</td>
<td>succeeds if Cell(n) is a structure</td>
</tr>
<tr>
<td>iscompound</td>
<td>n, e</td>
<td>succeeds if Cell(n) is a list/structure</td>
</tr>
<tr>
<td>isground</td>
<td>n, e</td>
<td>succeeds if Cell(n) is a ground term</td>
</tr>
<tr>
<td>isnil</td>
<td>n, e</td>
<td>succeeds if Cell(n) is a nil list</td>
</tr>
<tr>
<td>tifeq</td>
<td>n1, n2, e</td>
<td>succeeds if Cell(n1) == Cell(n2)</td>
</tr>
<tr>
<td>tifeq</td>
<td>n1, n2, e</td>
<td>succeeds if Cell(n1) == Cell(n2)</td>
</tr>
<tr>
<td>tiflt</td>
<td>n1, n2, e</td>
<td>succeeds if Cell(n1) &lt; Cell(n2)</td>
</tr>
<tr>
<td>tiflt</td>
<td>n1, n2, e</td>
<td>succeeds if Cell(n1) &lt;= Cell(n2)</td>
</tr>
<tr>
<td>tifgt</td>
<td>n1, n2, e</td>
<td>succeeds if Cell(n1) &gt; Cell(n2)</td>
</tr>
<tr>
<td>tifge</td>
<td>n1, n2, e</td>
<td>succeeds if Cell(n1) @&gt; Cell(n2)</td>
</tr>
</tbody>
</table>

Predicates such as **repeat, not, call, etc**, are treated as compiled predicates. They may be compiled first and stored in code area (CLP), or be loaded from a precompiled code file (ATOAM). The LVM leaves this open for later revision.

System builtins are implemented as stand-alone functions, i.e., they are invoked by standard calls (SICStus implements some system builtins, such as **arg/3** and **functor/3**, as kernel routines).

Like a normal procedure call, a system builtin call is translated as:

... **put**\* \to **builtin** ... 

that is, arguments (if any) to the builtin predicate must be put on the top of the stack, and then execute the **builtin** instruction. The **builtin** instruction takes one operand, the builtin’s name, as its operand. In execution, it invokes the named function which accesses the stack for input and carries out the corresponding activity. Appendix D gives a detailed discussion of builtin predicates.

5 Implementation Issues

In this chapter, we will discuss some implementation issues and some possible optimizations. As the LVM is virtually a WAM-like machine, most WAM compilers
can be easily transformed to a LVM compiler, and most optimization techniques except memory optimizations can be adopted. The major issue related with the design of a LVM compiler is the two stream code generation. We will present the two stream code generation algorithm is this chapter. In addition, we will discuss how to expand/contract memory blocks dynamically. This is closely related to extend the LVM to a multithreaded virtual machine for Mobile Agents applications.

5.1 Register AF and Argument Manipulation

Register AF plays an important role in both control and unification. It is the base (the pointer to the top) of the current activation frame. It will be set by two kinds of instructions: procedure calls, such as call, lastcall, ..., and memory allocations, such as alloa, alld, ... Thus, at the entry of a procedure, AF points to the last argument to this procedure. If this procedure does not require any extra memory space, the argument portion becomes its activation frame. If this procedure dispatches control based upon certain specified argument, then the argument can be indexed as the offset to the last argument. Hence, last-argument dispatching is economic because the last argument can be accessed through dereferencing AF without offset calculation. On the other hand, if a matching clause requires more memory space, then AF will point to the top of the newly allocated frame. Arguments to this clause must be indexed accordingly.

A possible optimization to manipulate arguments is given by Example 5.1. In which, only one putcon instruction is needed to bridge a call to foo/3 to a call to bar/4.

Example 5.1: foo(A, B, C) :- bar(A, B, C, 100).

\begin{center}
\begin{tabular}{lc}
\textbf{Example 5.1: the LVM code} \\
\hline
procedure & foo/3 \\
foo/3: & putcon 100 \\
& chaincall bar/4 \\
\hline
\end{tabular}
\end{center}

However, if the arguments are rearranged randomly in a chain call, such as presented in Example 5.2, a sequence of put instructions are required to build up calling arguments.

Example 5.2: foo(A, B, C) :- bar(B, A, 100, C).
5.2 Two Stream Coding Algorithm

In this section, I present the two stream code generation algorithm adopted in the LVM compiler. Before going to details of the algorithm, we need to discuss a special case in processing list terms.

List is the mostly used data structure in Prolog. A common case is that lists are used in the flat [X|Y] form, such as

Example 5.3: partition(Y, [X|L1], L2, [X|L]) :- ...

The general two stream translation of the partition/4 is:

```
Example 5.3: Case(1) (Y, [X|L1], L2, [X|L])

% READ STREAM
getlist 6, 3, 0, W.0 %STACK:
R.0: getlist 4, 1, 0, W.1 % 0: L
getval 1, 3 % 1: @X
% WRITE STREAM
getlist 4, 1, 0, W.1 % 2: L1
W.0: csetvar % 3: X
csetvar % 4: ↑ 1 (A4)
jump 1, R.0 % 5: L2 (A3)
jump 0, R.0 % 6: ↑ 3 (A2)
W.1: csetval 3 % 7: Y (A1)
csetvar
```

Examining the above two stream code, we find that a possible improvement could be done. The first getlist instruction jumps to W.0 if the input is an unbound variable. The write stream W.0 returns back to the read stream labeled by R.0, i.e., the next instruction of the read stream, immediately after two list cells having been initialized. As this is a common usage in most incremental list construction predicates, efficiency must be considered. Hence, we introduce a new instruction getvlist which will handle both read and write cases.
getvlist(n1, n2) {
    GP = V(n1);
    t = dereference(*GP);
    if (tag(t) == REF) {
        *t = make_list(n2);
        trail(t);
        *V(n2) = V(n2); // make two self-referential cells
        *V(n2+1) = V(n2+1);
        goto next_instruction; // remain in read stream
    } else if (tag(t) == LIS) {
        *V(n2) = t->argument[0];
        *V(n2+1) = t->argument[1];
        goto next_instruction; // remain in read stream
    } else backtrack;
}

A constraint of using getvlist is that the list must be at nesting level 0 and its two arguments must be first-occurred variables. Thus we have the following improved code:

Example 5.3: Case(2): (Y, [X][L1], L2, [X][L])

<table>
<thead>
<tr>
<th>% READ STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>getvlist 6, 3</td>
</tr>
<tr>
<td>getlist 4, 1, 0, W.0</td>
</tr>
<tr>
<td>getval 1, 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>% WRITE STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>W.0: csetval 3</td>
</tr>
<tr>
<td>csetvar</td>
</tr>
</tbody>
</table>

The algorithm presented below is a working version based upon the following auxiliary functions and data preparation:

1. A pushdown stack and two basic operations: push and pop, where the pop function will return a nil pointer if the stack is empty.

2. Arguments of a procedure have been tokenized such that multiple occurrences of a name share a single node, see the NODE type declaration.
3. Argument terms of the procedure have been flattened and inserted to a multi-
branch term tree. Each tree node is a term (see the TERM type declaration),
with one pointer to the eldest (leftmost) child and another pointer to the
younger (leftmost) brother. During insertion, each term is initialized with its
annotated type, nesting level, source term index, and annotated index.

4. Code being generated is enclosed in square brackets which can be replaced by
printf or fprintf with certain format.

5. Two stream coding takes two steps: a call to read_stream(arg_root) which
generates a sequence of get instructions, and a call write_stream(arg_root)
to produce a sequence of set and jump instructions. Note that these two
procedures are independent of each other. With a slight modification, jump
instructions can be folded into a few branch instructions.

    /* Two Stream Coding Algorithm */

typedef struct {
    char* name;
    int first;
} NODE;

typedef struct term {
    int type; // (REF, CON/INT, FLP, LIS, STR, LVS)
    int level; // nesting level
    int index; // source term index
    int annot; // annotated term index
    NODE* node;
    struct term* kids;
    struct term* next;
} TERM;

int coding_get(TERM* t, int c, int wlab){
    switch (t->type){
    case REF:
        if (t->index != t->annot){
            if (c)[cgetval t->annot];
            else [getval t->index, t->annot];
            return 1;
        }
    else return 0; // bypass getvar
    case CON:
if (c) [cgetcon t→node→name];
else [getcon t→index, t→node→name];
return 1;
case FLP:
    if (c) [cgetfloat t→node→name];
else [getfloat t→index, t→node→name];
return 1;
case LIS:
    if (c) [cgetlist t→annot, t→level, W.wlab];
else [getlist t→index, t→annot, t→level, W.wlab];
return 1;
case STR:
    if (c) [cgetstr t→annot, t→level, t→node→name, W.wlab];
else [getstr t→index, t→annot, t→level, t→node→name, W.wlab];
return 1;
case LVS:
    if (c) [cgetvlist t→annot];
else [getvlist t→index, t→annot];
return 1;
}

void read_stream(TERM* t){
int level, cget, rlab, wlab;
    level = t→level;
    wlab = rlab = cget = 0;
    do {
        if (t→level < level){
            [R.rlab];
            rlab++;
        }
        cget = coding_get(t, (level == t→level) && cget, wlab);
        if (t→type == LIS || t→type == STR) wlab++;
        level = t→level;
        if (t→next) push(t→next);
        if (t→kids) push(t→kids);
    } while (t = pop());
}

void coding_set(TERM* t){
    switch (t→type){
case REF:
    if (t->index != t->annot) [csetval t->annot];
else [csetvar ];
    return ;
case CON:
    [csetcon t->node->name];
    return ;
case FLP:
    [csetfloat t->node->name];
    return ;
case LIS:
    [csetlist t->annot];
    return ;
case STR:
    [csetstr t->annot];
    return ;
}

void write_stream(TERM* t){
    int level, wlab, rlab;
    TERM* sib;
    rlab = wlab = 0;
    level = t->level;
    do {
        for (; level > t->level, level-){
            [jump level-1, rlab];
            rlab++;
        }
    }
    level = t->level;
    if (t->type == STR){
        [W.wlab: setfun t->index, t->node->name];
        wlab++;
        sib = t->kids;
        while (sib){
            coding_set(sib);
            sib = sib->next;
        }
    }
else if (t->type == LIS){
    [W.wlab: ];

43
wlab++;  
sib = t→kids;  
while (sib){  
coding_set(sib);  
sib = sib→next;  
}  
}  
if (t→next) push(t→next);  
if (t→type != LVM && t→kids) push(t→kids);  
} while (t = pop());  
}

Although I mentioned earlier that a term must be fully initialized when it is inserted to a term tree, in fact we only need to initialize fields of type, nesting level, node, and two tree links. The source term index and the annotated index can be assigned systematically by calling the following program before two stream code generation.

void assign_cells(TERM* t, int size){  
TERM* sib;  
do {  
    if (t→type == STR){  
        t→annot = size--;  
        sib = t→kids;  
        while (sib){  
            index = size--;  
            if (sib→type == REF){  
                if (sib→node→first == -1){  
                    sib→node→first = sib→index;  
                    sib→annot = sib→index;  
                }  
                else {  
                    sib→annot = sib→node→first;  
                }  
            }  
            sib = sib→next;  
        }  
    }  
    else if (t→type == LIS){  
        t→annot = size; // no functor cell  
    }  
} while (t = pop());  
}
sib = t→kids;
while (sib){
    sib→index = size--;
    if (sib→type == REF){
        if (sib→node→first == -1){
            sib→node→first = sib→index;
            sib→annot = sib→index;
        }
        else {
            t→type = LIS;
            sib→annot = sib→node→first;
        }
    }
    else t→type = LIS;
    sib = sib→next;
}
if (t→next) push(t→next);
if (t→kids) push(t→kids);
} while (t = pop());

Parameter size is the sum of the number of flattened terms and the number of arguments. This function allocates a frame cell to each term and checks the first occurrence if the term is a variable. Thus, the first occurrence of a variable can later be identified by its source index being equivalent to its annotated index. Here we only discuss the coding method of head arguments. As a matter of fact, coding process of goal’s arguments is even simpler, because only one stream of code needs to be generated.

Example 5.4: m(f(t, g(f, h(t, X), t), f), i(j(Y, X), k(l(t, X), f), t, f), X, Y)

Example 5.4 shows a complex compound term and its two stream code generation. Its term tree is given by Fig. 5. It should be noted that each structure term in the term tree occupies two memory cells: one source and one annotated functor; on the other hand, a list term does not need a functor cell, see the assign_cells function. Hence, 31 cells are required in memory allocation where 1 cell for input argument (the source term of m/4) and 30 cells for flattened subterms. The following two stream code is generated by the given algorithm.
Figure 5: Example 5.4: Term Tree

*Example 5.4: the LVM code*

```
% READ STREAM
getstr 30, 29, 0, m/4, W.0
getstr 28, 24, 1, f/3, W.1
getcon 23, t
getstr 20, 2, g/3, W.2
getcon 19, f
cgetstr 16, 3, h/2, W.3
getcon 15, t
cgetval 26
R.0: getcon 17, t
R.1: getcon 21, f
R.2: getstr 27, 13, 1, i/4, W.4
getstr 12, 8, 2, j/2, W.5
getval 7, 25
cgetval 26
R.3: getstr 11, 5, 2, k/2, W.6
getstr 4, 2, 3, 1/2, W.7
getcon 1, t
cgetval 26
R.4: getcon 3, f
R.5: getcon 10, t
cgetcon f
R.6:
```

46
Example 5.4: the LVM code (Cont.)

% WRITE STREAM

W.0: csetfun  m/4
     csetstr  24
     csetstr  13
     csetvar
     csetvar

W.1: csetfun  f/3
     csetcon  t
     csetstr  20
     csetcon  f

W.2: csetfun  g/3
     csetcon  f
     csetstr  16
     csetcon  t

W.3: csetfun  h/2
     csetcon  t
     csetval  26
     jump  3, R.0
     jump  2, R.1
     jump  1, R.2

W.4: csetfun  i/4
     csetstr  8
     csetstr  5
     csetcon  t
     csetcon  f

W.5: csetfun  j/2
     csetval  25
     csetval  26
     jump  2, R.3

W.6: csetfun  k/2
     csetstr  2
     csetcon  f

W.7: csetfun  l/2
     csetcon  t
     csetval  26
     jump  3, R.4
     jump  2, R.5
     jump  1, R.6

47
Example 5.5 gives a three-argument head where the first argument is a list with nested arguments. Instruction `getvlist` can not be used in this case. A general two-stream code must be generated.

**Example 5.5: ([X, [Y, 3.14, b|L]], X, a)**

<table>
<thead>
<tr>
<th>Example 5.5: The LVM code</th>
</tr>
</thead>
<tbody>
<tr>
<td>% READ STREAM</td>
</tr>
<tr>
<td>getlist 12, 9, 0, W.0</td>
</tr>
<tr>
<td>getval 9, 11</td>
</tr>
<tr>
<td>cgetlist 7, 1, W.1</td>
</tr>
<tr>
<td>getlist 7, 5, 2, W.2</td>
</tr>
<tr>
<td>getlist 4, 3, 3, W.3</td>
</tr>
<tr>
<td>getfloat 3, 3.14</td>
</tr>
<tr>
<td>cgetlist 1, 4, W.4</td>
</tr>
<tr>
<td>getcon 1, b</td>
</tr>
<tr>
<td>R.0: getcon 6, []</td>
</tr>
<tr>
<td>R.1: getcon 10, a</td>
</tr>
<tr>
<td>% WRITE STREAM</td>
</tr>
<tr>
<td>W.0: csetval 11</td>
</tr>
<tr>
<td>csetlist 7</td>
</tr>
<tr>
<td>W.1: csetlist 5</td>
</tr>
<tr>
<td>csetcon []</td>
</tr>
<tr>
<td>W.2: csetvar</td>
</tr>
<tr>
<td>csetlist 3</td>
</tr>
<tr>
<td>W.3: csetfloat 3.14</td>
</tr>
<tr>
<td>csetlist 1</td>
</tr>
<tr>
<td>W.4: csetcon b</td>
</tr>
<tr>
<td>csetvar jump 4, R.0</td>
</tr>
<tr>
<td>jump 3, R.0</td>
</tr>
<tr>
<td>jump 2, R.0</td>
</tr>
<tr>
<td>jump 1, R.1</td>
</tr>
<tr>
<td>jump 0, R.1</td>
</tr>
</tbody>
</table>

Looking back at the `read_stream` function, it is easy to find that the algorithm adopts the `depth-first search with depth-first code generation` strategy. An instruction is coded for each term during the depth-first traversal of the term tree. An exception occurs when the visited term is a first-occurred variable. In this case, the `getvar` instruction is omitted (in fact, the LVM does not have this instruction) because the variable has already been in correct position. A read stream label is created.
whenever the traversal jumps back to a higher level term. Similarly, a write stream label is created if the visited term is a compound term. There is no need to remember either read labels or write labels because the write stream function will use the same traversal order to label each jump entry or code segment.

On the other hand, the write stream uses a different strategy, namely, the depth-first search with breadth-first code generation. In this strategy, terms are traversed in the depth-first manner. When a visited term is a compound term, the algorithm outputs its write label, a setfun instruction (if necessary), and then generates code for its arguments in the breath-first manner. A sequence of jump instructions will be coded if the traversal jumps back to a higher level term. Those instructions can be folded into fewer branch instructions. For example 5.4, the first three jumps in the write stream code can be replaced by one instruction, namely,

```
branch W.4, R.2, R.1, R.0.
```

In example 5.5, the last five jump's can be folded to

```
branch R.1, R.1, R.0, R.0
branch R.0, fail, fail, fail
```

5.3 Memory Expanding and Contracting

Memory management is a central issue in the design of the LVM. A good memory management scheme should be efficient all the time and should use little extra space for its own bookkeeping. Unfortunately no such scheme is known for the general problem of dynamic memory allocation. We compromise and use simple strategies to handle memory expanding/contracting as well as garbage collection, hoping that would be efficient enough in most cases. In this section, we focus on the memory expanding/contracting issues, and leave garbage collection issue to later discussion.

The LVM allocates two memory blocks for each thread: the program block and the environment block. During execution, a thread might need more program space to cope with dynamically generated clauses or symbolic constants, or more environment space to hold dynamic data. Further, for a Mobile Agents application, a thread might move to a remote host and continue execution from the moving point. Thus, both blocks should be able to be expanded (or possibly contracted) during run time, or to be moved together to another virtual machine and then be restored at different block spaces. The expansibility and mobility of a thread require efficient algorithms to relocate addresses from old to new.

First, we discuss the algorithms and strategies associated with the environment block. The environment block is controlled by two crucial pointers: TT and ST which represent the current trail top and the current stack top respectively. The environment is exhausted if these two pointers threaten to cross. Clearly, monitoring
the growth of both pointers must be very expensive. We only check the *threaten to cross* condition at a few points - stack allocation instructions. In the LVM, three instructions are responsible for stack frame allocation, namely, _alloca_, _alldod_ and _try_. When one of these is executed, it will first check if a round of garbage collection can be invoked and thus come up with the following cases:

1. no GC, TT and ST are not threaten to cross: allocate the frame and continue.
2. no GC, TT and ST are threaten to cross: expand the environment block.
3. GC and have enough copying area: do garbage collection.
4. GC but no enough copying area: expand the environment block and do GC.

The environment block needs to be expanded in cases 2 and 4. Generally speaking, there are two common ways of dynamic memory expanding: Unix dynamic memory allocation functions, such as _malloc_, _realloc_ and _free_, and Unix memory mapping functions, such as _mmap_, _mremap_ and _munmap_. The former scheme is adopted by XSB [43] - a Prolog emulator, and the latter is used in SableVM [44] - a Java virtual machine. The latter scheme has some advantages over the former. For example, using _mremap_ to replace _realloc_ may save many copying operations when the memory block needs to be relocated somewhere to grow. Unfortunately, _mremap_ is not supported by some OS or platforms. Thus we borrow ideas from XSB for environment expansion. In brief, a small environment block is allocated through _malloc_ to a new born thread, the block might be expanded by _realloc_ dynamically to meet the needs of the thread, and finally, the block will be returned to the system pool at the thread death.

Different strategies can be used to determine the growth size of the environment block. For example, XSB defines a new size calculation (abstracted) macro as follows:

```c
#define resize_stack(size, min) (size) < (min) ? (size) + (min) : 2 * (size);
```

This macro selects a binary exponential growth function or a linear growth function depending upon the given condition. The LVM intends to use the similar method. More specifically, let _n_ be the number of expansions, _k_ the turning point from binary exponential to linear, we have the following formula (_new_size_ is in the unit of Mega Byte):

```latex
\begin{align*}
\text{new}_\text{size} &= 2^n & \text{if } (n \leq k) \\
\text{new}_\text{size} &= 2^k + 4 \times (n - k) & \text{if } (n > k)
\end{align*}
```

50
The turning point $k$ depends on the memory size of the underline machine and the maximum number of threads allowed. If we choose 5 as the turning point, then an environment block can be expanded to 2M, 4M, 8M, ..., up to 32M, and then shift to the linear expansion manner, that is, 36M, 40M, ..., up to the maximum size.

Figure 6 shows the memory layout before and after expanding an environment block. The `realloc` function changes the size of an old block and returns a pointer to a (possibly moved) new block. The contents will be unchanged up to the lesser of the new and old sizes. Now, we need to adjust contents of LVM registers as well as all tag-on-pointer terms in the stack. We also need to move trailed pointers to the top area of the new block and make the corresponding pointer relocation. This algorithm is straightforward and simple: adjusting each tag-on-pointer term through a linear scan of the stack. However, it is worth to note that a special kind of pointers in the stack, i.e., program pointers such as CP’s and BPs, should remain unchanged, because they are addresses to the program block. One way is to rule out these pointers during linear scanning of the stack, which might involve some overhead because each reference must be checked. Another way is to adjust all the reference terms without testing, and then restore all the program pointers by a cost effective scanning following the CF and BB chains.

The program block of a thread contains program code and a (hashing) symbol
The LVM separates the program block from the environment block because it exhibits different expansion and management patterns. First, this block is not subject to garbage collection. Secondly, it is unlikely to be expanded as often as the environment block. Figure 7 gives a possible layout of the program block.

The code portion might be expanded only if the directive `dynamic/1` occurs in the program text. How to handle dynamic procedures will be investigated during implementation. The symbol table and its associated hash table might be expanded if new atoms are created during execution (through input or atom constructing predicates). The basic rule is that all existing atoms will remain in their original offsets to the `symbol_base` after expansion. However, they must be rehashed with respect to an expanded hash table. In order to process characters and strings efficiently and effectively, we define that the index field of an atom term be arranged as follows:

\[
\begin{align*}
\text{ASCII code of a character} & \quad \text{if} \ (0 \leq \text{index} \leq 255) \\
\text{offset} = \text{index} - 256 & \quad \text{if} \ (\text{index} > 255)
\end{align*}
\]

Contracting only applies to the environment block. A possible point to reduce the size of the environment block is a big chunk of memory being reclaimed after a round of garbage collection. This issue remains for further investigation.
6 Chronological Garbage Collection

So far, our discussion is based upon that we have an infinite memory to use, namely, stack frames are allocated on the fly with no concern for memory efficiency. Unfortunately, the assumption of an infinite memory is not true in the real world. As application programs have grown enormously in size and complexity, especially programs written in Prolog or Lisp which typically manipulate large data structures with complex inter-dependencies, automatic storage reclamation is essential for practical implementations. Moreover, the LVM’s merged stack/heap paradigm makes the memory consumption even worse. A simple recursive program, such as Tak, could quickly run out of the stack space. Thus, a high-efficiency, frequently-run garbage collector is the core of the LVM model.

Many garbage collection algorithms have been studied and used over the years. A brief summary of typical algorithms is presented below:

Reference Counting: Each dynamic object has an additional field, the reference count which will be updated whenever a pointer to this object is created or deleted. The object is garbage if its count drops to zero.

Mark-Sweep: Mark-sweep relies on a global traversal of all live objects to determine which cells are available for reclamation. It is performed in two phases. The marking phase identifies all active cells. The sweep phase scans the heap linearly from bottom to top and returns garbage cells (unmarked cells) to the free pool.

Copying: Copying collector divides the heap equally into two semi-spaces, one of which contains current data and the other obsolete data. The collector starts by flipping the roles of the two spaces. It traverses the active objects in the old space and copies each live cell into the new space.

Mark-Compact: The main drawback of mark-sweep is its tendency to fragment the heap if required to handle a variety of objects of different sizes. By compaction we mean that, at the end of a compacting phase, the heap will be divided into two contiguous area. One area will hold all active data whereas all free cells will be held in the other area.

Generational: Many researchers have gathered considerable evidence to support the weak generational hypothesis that most objects die young. Though this varies from language to language and program to program, a commonly accepted estimate is that 80 to 98 percent of all newly created objects die within a few million instructions, or before another megabyte has been allocated. The insight behind generational garbage collection is that storage
reclamation can be made more efficient and less obstructive by concentrating effort on reclaiming those objects most likely to be garbage. i.e., young objects.

What make a good garbage collection algorithm? Generally speaking, we have the following criteria:

- it must be safe: live objects must never be erroneously reclaimed;
- it should be comprehensive: garbage should not be allowed to float unreclaimed;
- it should be cost-effective: the overhead of both time and auxiliary space used by the algorithm should not greatly influence overall performance;
- it should minimize the pause times: fine-gained incremental collection facilitates the interactive systems;
- and finally, it should take locality into account: good locality improves cache and virtual memory performance.

Strictly speaking, none of the above GC algorithms offers an optimal solution meeting these criteria. In general, conventional garbage collection may yield a performance penalty ranging from a few percent to around 20 percent. In a worst case, garbage collection may take more than 80 percent of a program's total execution time. Another factor is the extent of pauses during garbage collection. It is important not only for interactive programs, but also for multiuser application servers which handle huge heaps. For example, by using mark-and-sweep algorithm which suspends all user threads while it is running, with Java object heaps of 100-500 MB, the common pause time will range from 5 to 20 seconds. From [17], a worst case for a multiuser server with a 256 MB heap will be stopped cold for minutes, and this phenomenon is called garbage collector hell.

Some of GC algorithms have been adopted in various Prolog implementations. For example, SICStus Prolog uses a mark-compact algorithm embedded with variable shunting during collection [18]. In addition, researchers have suggested different ways to improve existing GC algorithms. Bruynooghe [19] proposed an algorithm to detect which of the references on the stack will be used in the remaining computations, and mark only these references. Le Huitouze [20] presented a new data structure, attributed variable, combined with a memory management machine, MALI, which encapsulates a garbage collector. Taki [21] proposed to combine an incremental collector with a subset of reference-counting and a stop-and-collect copying collector for PIM memory management. Bevemyr and Lindgren [22] exploited a copying algorithm and concluded that copying collection is a viable alternative

However, applying traditional GC algorithms to Prolog implementation exposes various problems:

**Mark-Sweep:** This method preserves the chronological order of heap and stack segments as required for backtracking. However, the costs of garbage collection are high. Every active cell is visited in the marking phase, and all cells are examined by the sweep phase. Thus the asymptotical time complexity is proportional to the size of the entire heap $H$, *i.e.*, $O(H)$.

**Copying:** Although Copying GC offers good time complexity $O(R)$, where $R$ is the size of useful data, a frequent claim among implementors is that backtracking is incompatible with this paradigm. The reason is that Copying GC usually re-order data in the heap regardless of the structure of choice frames. [22] presented a top-down copying method and shown that their method is simpler and more efficient than the standard mark-sweep method. However, as the top-down copying does not preserve the order on the heap, it is undesirable because instant reclaiming by backtracking becomes impossible. To solve this problem, [35] proposed a segment order preserving copying algorithm. It is a bottom-up collector which retains the heap order so that space can be reclaimed on backtracking. However, both proposals require some extra bit(s) per heap cell for garbage collection and involve two rounds of traversing of the live data: one for marking and another for copying.

**Generational:** The main idea is that two generations are delimited by some choice point which serves as the *write boundary*. Garbage collection is done in the new generation only. When references are created from the old generation to the new one, these references are considered as roots. A Prolog run-time system records such roots in the trail, and the collector simply has to scan the trail to find them. The problem related with the generational behaviour is that the write boundary may leave in such a state that old generation is empty (or almost empty). For example, deterministic programs would never be in a position to benefit from the advantage of generational collection. A solution is to create artificial choice points. However, no desirable algorithm has been developed yet.
A very interesting idea similar to what we are working on is proposed by Tarau [31]: "... we suggest to compare the space consumption of the WAM on the naive reverse benchmark, $O(N^2)$, with the size of useful data that is produced, $O(N)$. ... For most of the programs, however, when compared with their theoretical lower limit (the size of the computed answer) one must agree that the space complexity of WAM computations is almost always higher. Obviously, an easy way to restore the equilibrium at some stage is to copy the (possibly partial) answers and discard the space used for computations. ... An ideal memory manager is ecological. We want it to have a self-purifying engine that recuperates space not as a deliberate garbage-collection operation but as a natural way of life, i.e., something done inside the normal, useful activities the engine performs." The only example presented in his paper is the heap lifting technique for implementing findall, however, Tarau believed that the concept of ecological approach to memory management is probably more important than the actual implementation. Another interesting idea came from Barrett and Zorn [32]. They noticed that all generational algorithms occasionally promote objects that later become garbage, resulting in an accumulation of garbage in older generations. Thus they extended existing generational techniques with a mechanism that can dynamically adjust the boundary between old and young generation either forward or backward in time, essentially allowing data to become untied.

Incremental collectors aim to diminish the disruptiveness of GC by spreading out the GC work into more uniformly distributed parcels of smaller and bounded size. Because incremental GC requires extra coordination between mutator and collector and higher conservatism, it is more expensive than stop-and-collect GC. One of the aims of generational garbage collection is to reduce pause times. Therefore it may be worth considering when to schedule garbage collection. Garbage collection will be made more efficient if it is run at time when the volume of live data is low. The ends of compute-bound periods or procedure call-return points may be good times to collect, since they are often dispatching points between major computations. Local minima of stack height are opportune moments to trigger collections. Detecting true local minima of the stack height is problematic. One approximate solution is to trigger a collection whenever the stack height drops below a certain point [16].

Based on the study of existing GC algorithms, I exploit a new policy in the spectrum of generational garbage collection, and I name this new policy as Chronological Garbage Collection. CGC is mainly a generational copying collector. However, it has new features different from the traditional generational collectors:

**Generation Organisation:** There are no predivided generations, instead, CGC introduces a concept of chronological generation - a dynamical way to divide generations. The size of a generation is bound by the machine cache size, and the number of generations vary from program to program.
Scheduling Garbage Collections: CGC controls the frequency of collector invocations by capturing the continuation point (an approximation of local minima of stack) and a factor of cache size. Therefore it collects garbage (in most cases) incrementally with a trivial pause time.

Promotion Policy: Most generational collectors promote survivors from the youngest generation into old generations. On the other hand, CGC discards the young generations, and makes the survivors temporarily tenured as a part of the generation which becomes the youngest after a round of collection.

6.1 Chronological Generations

Before we go into the details of what exactly constitutes a chronological generation, we must understand the continuation mechanism of the LVM. The LVM adopts the same strategy as the WAM to handle execution environment. Namely, the stack is organized as a linked list through a Continuation Frame slot. Different from the WAM which discards the current stack frame before issuing the last call, the LVM only resets the continuation environment (CF) upon the last call and leaves the current frame unclaimed. Such a frame is called a finished frame because no control will ever return to this frame. But it can not be discarded as the WAM did because it may hold long lived data objects.

Let P be a procedure, and CF a pointer to a continuation frame. When P is invoked by a caller, there are two possible cases:

- P is invoked by a middle call, in this case CF is the caller’s activation frame; or
- P is invoked by a last call, then CF is the continuation frame of a sequence finished procedure invocations (except the last call).

In other words, at the entry point of P, its continuation environment could be its immediate parent, or could be its ancestor. This observation gives us Property 1.

Property 1: At the entry point of a procedure, all non-choice frames (if any) in between CF and AF (excluding CF and AF) are finished frames.

The proof is trivial and directly comes from the nature of the LVM stack allocation and the way of implementing last calls. If we use CF as the delimiter to divide the stack into two generations, a consequence of Property 1 is that a generational garbage collection process can be safely applied to the young generation (finished frames) provided that:

- finished frames are not frozen by some choice point;
- all cross CF references have been recorded as initial roots; and

57
• arguments of the call have been properly reserved and restored.

Unfortunately, such kind of garbage collection does not work in practice. First, the cost of running such an aggressive collector must be significant, because it seems that we are using garbage collection to replace LCO for every finished frame. Secondly, it is too expensive to record all the forward bindings (roots) in execution. To solve these problems, we need a set of new measurements and rules to group stack frames into dynamic generations.

Certainly, collection frequency can always be reduced by increasing the size of the region being collected. The most important measurements to determine generations are generation-gap and cache-limit. Generation-gap is defined as the distance from the stack top to some old generation delimiter. Cache-limit is a machine-dependent constant and is used as the measurement for creating a new generation and invoking garbage collection. From my experiments, $1/3$ to half of the size of data cache would be a proper range to select. The underlying assumption is that data accesses are typically concentrated on a small portion of the address space of the program - the working set. It is further assumed that a cache window is moving forward/backward to hold the working set of the program as well as the space required for garbage collection. Thus, whenever one or more finished generations are recognized, CGC will be invoked to reduce the size of the current working set.

The effectiveness of the CGC rests upon the size of the cache, not the size of the main memory. At first sight, it seems that such frequent garbage collections must involve a big system overhead. However, a direct consequence of CGC is that the cache miss rate can be sharply reduced. Related problems of cache performance have been widely studied [41, 39, 42, 38]. We have simulated the cache performance of CGC based on an old experimental version of LVM[36]. An emulator was developed to do the trace-driven cache simulation. Direct-mapped cache and set-associative cache with different cache sizes, block sizes and set associativities were simulated and measured. The results shown that on a range of benchmarks, their cache misses on a machine with an infinite memory are reduced up to 90% when they are cooperated with the CGC algorithm. Although the simulation results do not directly apply to this new LVM design, we believe that our new version remains the similar cache performance because the idea of CGC and its basic algorithm does not change very much. In other words, the cost of CGC can be (almost) paid back by the improved cache performance.

As it turns out, a simple comparison

$$((ST – an\_old\_generation\_line) \geq CACHE\_LIMIT)$$

plays a very important role in the CGC algorithm: if the generation-gap is greater than or equal to the cache-limit, it either creates a new generation or triggers the collector; otherwise, it does nothing. This comparison has served as a double-edged
sword for controlling garbage collection throughout execution. On one side, we want to control collection frequency such that the collector is not invoked unless there is a reasonable amount of accumulated garbage. On the other side, we want to collect useful objects more frequently than ordinary copying/generational collectors so that most working objects are kept in the cache.

Now, we are ready to continue with implementation details. The LVM defines three memory allocation instructions, i.e., alloa, allod and try. In fact, these instructions not only allocate the specified stack frame, but also determine whether a new generation should be created, or a round of garbage collection should be triggered. Suppose that the make_X_frame(...) function is replaced according to the type of frame allocation, namely, alloa makes an A-frame, allod a D-frame, and try a C-frame, then these instructions share the following generic algorithm.

/* Generic Memory Allocation Algorithm */
generic_allocation(int n1, int n2) { // n1: # of arg, n2: # of additional cells
  if (CF > GL) {
    // CF is in the nursery space
    if ((ST - GL) ≥ CACHE_LIMIT) {
      GL = ST;
      new_generation();
    }
    make_X_frame(n2);
    goto next_instruction;
  }
  else {
    if (CF < BB) {
      // CF belongs to a nondeterministic generation
      temp = get_gline(BB→TT);
    }
    else {
      // CF belongs to a deterministic generation
      temp = get_gline(CF→TT);
    }
    if ((ST - untag(*temp)) ≥ CACHE_LIMIT) {
      GL = untag(*temp);
      gc(n1, temp);
    }
    make_X_frame(n2);
    goto next_instruction;
  }
}

59
Figure 8: Stack Division and Views

As we focus on the garbage collection issues in this chapter, environment expansion function is not addressed in the above generic memory allocation algorithm. This function can be easily added to our algorithm in the implementation of an LVM emulator.

Traditional generational garbage collection algorithms divide the heap into two or more (static) generations, segregating objects by age. Objects are first allocated in the youngest generation, but are promoted into an older generation if they survive long enough. On the contrary, CGC does not pre-divide its working space. When a program starts, there is no generation. Both GL and ST point to the stack bottom. The whole stack is the nursery space which is the area used for new allocation. As soon as the size of allocated stack reaches the CACHE\_LIMIT, a new generation is born. A special case is that a choice frame is allocated. If so, a new generation must
be created by the `make_choice_frame()` function, regardless of the CACHE\_LIMIT. The reason is to let each nondeterministic segment coincide with a new generation (segment preserving technique is discussed in [35]). Therefore, we have a single generation view in GC. In both cases, register GL will be set to the stack top which represents the most recent generation line separating the nursery space and the generational space. Fig. 8 shows the stack division and terminologies used in our discussion with respect to different views, where $g_1$, $g_2$ and $g_3$ (pointed by register GL) represent dynamic generations from old to young.

Hence, when the program continues, a memory allocation instruction will face two possible situations:

**(CF > GL):** It also implies that (CF > BB) because (GL ≥ BB). No garbage collection can be done in this case. The reason is straightforward: CF is still in the nursery space and no finished generation exists. Therefore, we just allocate a new frame and set a generation line if necessary. If a new generation line is set, then all frames from the previous line to the new line belong to this new born generation.

**(CF ≤ GL):** This condition means that CF belongs to an old generation. We shall first search the actual generation to which CF belongs. It should be noted that CF could belong to an arbitrary generation under GL but above BB (a deterministic generation), or it could belong to a generation under BB (a nondeterministic generation). A different search pointer is required for these two cases. After having found CF’s generation, we check if there exists enough space worth to invoke garbage collection. If the generation-gap is greater than the specified limit, CGC is called. Here I shall point out that different standards might be used for two types of instructions. For example, `alloca` and `allod` can use the CACHE\_LIMIT to control the invocation of CGC whereas `try` may start CGC even if there exists a small amount of garbage. This is because we do not want to freeze uncollected space by a choice point. The limit used by `try` depends on experiments.
6.2 Generation Line and Remembered Set

Garbage collection starts by tracing from an initial root set. Traditional generational GC defines its initial root set to include two parts:

- Roots from scanning the heap for cross-generation pointers, and
- Roots from scanning registers and the stack.

Scanning the whole old generation to discover the pointers that go into the young generation is an expensive operation. There are two commonly used solutions for this problem: remembered set and card marking. In general, pointers going forward in time (from one generation to a younger one) are usually rare. A remembered set is a technique for remembering these pointers. This set is maintained by the write barrier during the normal execution of an application. A write barrier is used to trap and filter each eligible pointer assignment and insert forward-in-time pointers into a remembered set. Maintaining the remembered set implies a costly overhead. Card marking partitions the heap into cards of equal size, and reduces runtime overhead by recording only dirty cards when an application modifies objects in these cards. However, it must scan all dirty cards for cross-generation pointers when the collector is invoked.

In the case of Prolog, we found that trailing and detrailing operation bear a strong resemblance to the write barrier mechanism. These operations were introduced by the WAM to cope with nondeterministic computation. Trailing is used to record variable bindings if they were unbound before creation of the current choice, and detrailing is needed to reset these variables to unbound upon backtracking. Similarly, the write barrier should record variable bindings if they were unbound before creation of the current generation, and garbage collection will transitively traverse these remembered pointers to gather all reachable objects and return useless memory. Thus, extending the trailing operation to be the write barrier makes the scheme of remembered set a natural choice for CGC.

Scanning for roots in registers can be done in constant time. However, scanning for roots in stack requires time linear to the size of the stack. It could be quite expensive when a highly recursive calls leading to very deep stack. This happens a lot in logic and functional programming languages. One solution is to apply the write barrier to local variables, but this would considerably increase the cost of the barrier. For instance, if we remember all local roots, we not only increase the size of the remembered set, but also need a special function to get rid of these local roots when their stack frame is popped. Another solution is to link the generation information (water mark) with stack frames, such that only frames allocated after the last collection need to be scanned for pointers into the young generation. However, some extra overhead is involved with this approach to overcome the difficulty that
the high water mark frame might be popped between collections [16]. On the other hand, applying CGC to a merged stack/heap architecture rules out this problem. There is no distinction of stack roots and heap roots, all cross-generation roots are properly maintained by the write barrier and the remembered set paradigm.

From the previous discussion about generation creation, it has become clear that a frame is always allocated before its generation is established. Thus, an immediate question is how to link a frame with its later born generation. Recall that the LVM defines three kinds of stack frames: activation frame, data frame, and choice frame. As the data frame does not have control information, i.e., it can not be a continuation frame, we only need to consider the activation frame and the choice frame.

First, a special slot is defined in both frame structures which is used to save the register TT, the trail top pointer, when a frame is allocated. The value in the TT slot is thus served as the generation search pointer (note that the TT slot in a choice frame is also used in backtracking). Secondly, we introduce a new (GEN) tag to represent a generation line. Function new_generation() is substituted by a simple assignment statement:

\[*TT = = GL | GEN;\]

As a consequence, the trail is segmented by generation lines, see Fig. 9. Each generation line saved in the trail is used as the delimiter and the boundary to separate two generations. The reason of introducing the GEN tag is to distinguish between generation delimiters and roots. In implementation, this tag could be replaced by any non-REF tag without ambiguity. With this setup, we have the following property:

**Property 2:** Let F be an arbitrary continuation/choice frame in generational space. The generation-line associated with F is the first (tagged) generation-line being found by a downward search of the trail starting from F→TT.

The proof is straightforward. Let \( g_i \) be a generation-line saved at trail position \( t_i \), and \( g_j \) be the next generation-line at \( t_j \). Frames allocated in between \( g_i \) and \( g_j \) belong to generation \( g_j \). Let \( t \) be the generation search pointer of an arbitrary frame which belongs to generation \( g_j \), then \( t \) must be in the range:

\[ t_i < t \leq t_j \]

Generally speaking, for an arbitrary continuation/choice frame F which is not in the nursery space, its associated generation-line can be found by a call get_gline(F→TT) to the following function:

```c
int** get_gline(int** t){
    while (tag(*t) != GEN) t--;
    return t;
}
```
Since the CGC deals with multiple dynamic generations, a remembered set is associated with each generation and the write barrier is implemented by a generalized trail mechanism. As mentioned before, register GL is always greater than or equal to BB. Hence, a generalized trailing operation fulfills the requirement of both backtracking and garbage collection:

```c
void trailing(int* t) {
    if (t <= GL) *tt-- = t;
}
```

Recording cross generation pointers may be the major cost of generational garbage collection, because several machine instructions must be executed for every potential pointer assignment. However, it is worth to note that a trailing operation has to be done for each variable binding even without the presence of the CGC. The extra overhead caused by CGC is that the trail stack is expanded to hold remembered sets as well. With this setup, Fig. 9 shows a possible layout of the trail stack. Now, let us assume that Fig. 8 and Fig. 9 together give a snapshot of execution, it is easy to see that any currently trailed root goes to \( g_3 \)'s remembered set (note that \( g_3 \) is pointed by GL in Fig. 8).

A refinement that I introduce next is a special trailing operation dedicated to backtracking only. From backtracking point of view, trailed variables must be set to unbound. From garbage collection point of view, however, remembered variables must be collected. Clearly, if a remembered variable was bound to a non-structure term, the collection work is completely wasted. Thus, we need a filter to the write
barrier which gets rid of recording non-structure bindings. This special trailing operation is almost the WAM-like trailing operation, except that the original WAM model requires several pointer comparisons to determine if trailing is necessary:

\[
\text{void btrailing(int* t)\
  \quad \text{if} \ (t \leq BB) \ *\text{TT}-- = t; \]
\]

Since variable bindings are performed mainly in get instructions, btrailing() is used by the subset of get’s annotated to non-structure terms, whereas trailing() is used by the one annotated to structure terms. This refinement greatly reduces the number of roots in remembered sets with virtual no extra cost.

6.3 An Overview of CGC

Now, we have an embryonic form of the CGC algorithm, refer to Fig. 10. At some stage of execution, the collector is invoked. Four dynamic areas are divided by GL, ST and TT:

- the space below GL holds old generations;
- the space between GL and ST contains finished frames - young generations, and calling arguments;
- the space between ST to TT is the free area which will be used to save a temporary copy of useful data; and
- the space above TT stores remembered sets.

The algorithm consists of three phases: initialization, collection and compact.

The initialization phase prepares the initial root set which is used as the starting point in determining all reachable objects during garbage collection. Let \( r_j \) be the remembered set of \( g_j \), \( r_{arg} \) the set of argument roots of the last call. For a given generation \( g_i \), its initial root set is defined by:

\[
\text{root\_set}(g_i) = \bigcup_{j \geq i} r_j \bigcup r_{arg}
\]

In other words, \( g_i \)'s initial root set consists of all the remembered sets which belong to generations younger than or the same age as \( g_i \), plus the argument roots for the last call. In fact, if \( g_i \) is found at the trail position \( t_i \) (the search result of get\_glimpse(...)), then remembered sets saved from TT to \( t_i \) give the first portion of \( \text{root\_set}(g_i) \). Hence, the initialization phase only needs to push \( r_{arg} \) onto the trail, and the initial root set, namely, roots from the new TT to \( t_i \), is ready for collection.

65
Figure 10: CGC Overview
The collection phase transitively traverses from the initial root set and creates copies of all reachable objects located in the young area. Most copying collectors use Cheney’s breadth-first algorithm [33] to collect useful data: repeatedly copying live objects to the copy area, and then scanning these copied replicas for pointers to further objects that have not been copied. A simple Cheney-style collector, the scans each live object just once, is inadequate since an object might be shared between different roots, each of which accesses a different set of live cells. An improved algorithm [22] involves two rounds of traversing of the live data: 1) marking the live data, and 2) scanning backward/forward to find a marked block and then copying the whole block and setting up forward-pointers. The advantage of marking-copying is that no duplication of cells could occur. As each copied cell has been forwarded, a subsequent visit to a copied block will stop copying by referring to the forward pointer directly. A problem in this scheme is that it requires two extra bits per heap cell: one for marking a copied as forwarded, and another for indicating a live internal cell (a cell appears inside a live structure).

On the other hand, CGC takes a different copying strategy. It does not have a marking phase, instead, it uses the root stack (the trail) to simulate a recursive algorithm [37], i.e., transitively traverses each root, creates a copy along the traversing and set up a forward pointer to avoid duplication. As a result, CGC does not need any extra bits dedicated to garbage collection. During copying, new roots might be added to the root set. This phase stops when the root set becomes empty. Live objects are temporarily placed in the copy area, however, references in these copies have already been calculated to final destination addresses.

A so called variable shunting is done during collection. “Variable shunting consists in finding variables which are only seen in their bounded state, and then replacing pointers to such variables with their binding values. Such variables are those for which no choice points have been created (or they have been destroyed by cut operation) between creation time and binding time” [24]. A simple fact is that any intermediate reference in young area fulfills the above mentioned condition, therefore, CGC implements a cheap (partial) variable shunting by simply dropping all intermediate references which reside above GL. Namely, a collecting root will be bound to a constant, or a copied structure instance, or a reference below GL. The reason of saying that CGC does partial variable shunting is that this technique is not applied to references in old generation.

Finally, the compact phase will move copied objects, by a simple word-wise loop, back to the young area without any pointer adjustment (this scheme is also used in [35]). The remaining space in young area will be returned to the free pool as a whole, and the new stack top is assigned to ST. To protect the collected data from repeated collections, GL is promoted to this new stack top which makes collected data temporarily tenured. As Barrett and Zorn [32] pointed out that those promoted objects might later become garbage, resulting in an accumulation of garbage in older
generations. To solve this problem, they proposed a dynamic generation adjustment mechanism to allow part of collected data untenured. Fortunately, such a mechanism is natural in the CGC algorithm: after a collection, temporarily tenured data belong to the youngest generation and will become untenured when a subsequent garbage collection is triggered from an older generation. However, a shortcoming is that some collected objects might survive through many collections. Another problem is that CGC is not comprehensive: garbage in old generations are still float unreclaimed. Whether the LVM requires a major collector to clean up the whole stack from time to time is left to be investigated.

This brief discussion gives a general view of the CGC algorithm. The whole algorithm (including initialization, collection and compact) is implemented in less than 100 lines of C code and is embedded in the LVM engine as an inline function (detailed algorithm is given in the next section). It is completely transparent to Prolog programmers. It is also transparent to the LVM compiler, i.e., a LVM compiler can be designed with no knowledge of the existence of the CGC.
6.4 Collector Algorithm

In this section, I present the details of the CGC collector. This algorithm is conditionally invoked by memory-allocation instructions, namely **alloca, alldod and try**. Before entering this procedure, four system registers, *i.e.*, GL, TT, ST and BB have been properly set. In addition, we need five working registers and two input arguments. Table 6.1 gives a brief description of their roles in the algorithm.

<table>
<thead>
<tr>
<th>System registers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int* GL</td>
</tr>
<tr>
<td>int** TT</td>
</tr>
<tr>
<td>int* ST</td>
</tr>
<tr>
<td>int* BB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Working registers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int* AX</td>
</tr>
<tr>
<td>int BX</td>
</tr>
<tr>
<td>int CX</td>
</tr>
<tr>
<td>int* DX</td>
</tr>
<tr>
<td>int** AF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input arguments:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int n</td>
</tr>
<tr>
<td>int* bottom</td>
</tr>
</tbody>
</table>

**Table 6.1: Working Registers and Inputs**

Next, we define a set of macros in order to make the algorithm easier to understand. Some of the macros will be explained further along our discussion. Further, Fig. 11 shows the memory configuration at the collector’s entry point.

**Macro** | **Definition**
---|---
CTOP | (**int**) (ST + CX) // top of copy area
DTOP | (**int**) (GL + CX) // top of destination area
gc_copy(X) | *(ST + CX++) = (**int**) (X)
gc_pop() | (TT == AF) ? 0 : *(++TT)
gc_push(X) | *TT -- = (**int**) (X)
hit_old(X) | (**int**) (X) ≤ GL
hit_mid(X) | (((**int**) (X) ≤ ST) && (**int**) (X) > GL))
hit_choice(X) | (**int**) (X) < BB
hit_copied_var(X) | (**int**) (X) > ST
hit_copied_list(X) | (is_var(X) && (**int**) (X) > ST) && !is_var(*(**int**) X))
hit_copied_structure(X) | (is_var(X) && (**int**) (X) > ST))
destination(X) | (**int**) (GL + (**int**) (X) − ST))
Figure 11: Memory Configuration
/* Collector Algorithm */
cgc(int n, int** bottom){

GC_INITIAL:

    AF = bottom;               // set bottom of root stack
    CX = n;                    // set number of arguments
    while (CX) {               // check each argument
        if (is_var(*ST) && *ST == (int)ST)
            *ST = DTOP;       // make a new void
        else if (!is_con(*ST))
            gc_push(ST);      // push an argument root
            ST--;              
            CX--;             
    }
    CX = n + 1;                // set initial copy count

GC_FETCH:

    AX = gc_pop();            // pop a root
    if (AX == 0) {            // root stack empty
        for (BX = 1; BX <= CX; BX++)   // COMPACTING
            GL[BX] = ST[BX];
        ST = GL + (CX - 1);    // CX is one more ahead
        CX = n;               // get number of arguments
        for (BX = 1; BX <= CX; BX++)  // mirror arguments
            ST[BX] = GL[BX];
        GL = (ST += CX);      // promote generation line
        new_generation();     // *tt-- = make_str(ST);
        return;
    }
    else if (hit_mid(AX))     // bypass intermediate root
        goto GC_FETCH;
    else if (hit_choice(AX)){  // AX is also used for backtracking
        gc_push(*AF );        // move a root to top
        *AF -- = AX;          // use that empty slot to save AX
    }
}

71
GC_SHUNTING:

BX = *AX; // init a chasing pointer
while (is_var(BX)){
  if (hit_copied_var(BX)){ // an already copied instance
    *AX = destination(BX);
    goto GC_FETCH;
  }
  else if (hit_old(BX)){ // an old generation instance
    *AX = BX;
    goto GC_FETCH;
  }
  else if (* (int*)BX == BX){ // a self-ref instance
    *(int*)BX = (int)AX; // root reversing
    if (hit_copied_var(AX)) // self-ref a young root
      *AX = destination(AX);
    else *AX = (int)AX; // self-ref an old root
    goto GC_FETCH;
  }
  else BX = *(int*) BX; // shunting intermediate refs
}

GC_COPY:

if (is_con(BX)){ // an atom/integer
  *AX = BX;
  goto GC_FETCH;
}
else if (is_flp(BX)){ // a floating reference
  DX = addr(BX); // get address
  if (hit_old(DX)){ // an old/ground instance
    *AX = BX;
    goto GC_FETCH;
  }
  *AX = make_flp(DTOP); // make a new FLP instance
  AX = (int*)DX; // get word1
  gc_copy(AX); // copy word1
  AX = (int*)(DX + 1); // get word2
  gc_copy(AX); // copy word2
  goto GC_FETCH;
}
else {
    // a list/str
    DX = addr(BX); // get address
    if (hit_old(DX)){ // an old/ground instance
        *AX = BX;
        goto GC_FETCH;
    }
    if (is_list(BX)){ // a list
        BX = *(DX + 1); // get the tail
        if (hit_copied_list(BX)){ // a copied list
            *AX = make_list(destination((int*)BX - 1));
            goto GC_FETCH;
        }
        *AX = make_list(DTOP); // make a new instance
        AX = (int*)DX; // get head
        if (!is_con(AX)) gc_push(CTOP); // push a head root
        gc_copy(AX); // copy the head
        *(DX + 1) = CTOP; // set tail forward
        AX = (int*)CTOP; // set the tail root
        gc_copy(BX); // copy the tail
        goto GC_SHUNTING; // last root optimization
    }
    else { // a structure
        BX = *DX; // get functor
        if (hit_copied_structure(BX)){ // a copied structure
            *AX = make_str(destination(BX));
            goto GC_FETCH;
        }
        *AX = make_str(DTOP); // make a new instance
        *DX = CTOP; // set functor forward
        gc_copy(BX); // copy the functor
        BX = arity(BX); // get # of args
        while (BX--){
            AX = (int*)(++DX); // get an argument
            if (is_con(AX))
                gc_push(CTOP); // push an arg root
            gc_copy(AX); // copy an argument
        }
        goto GC_FETCH;
    }
}
}
At the entry point of the collector, the initial root set, i.e., the trail stack from TT to AF, only contains roots across the given generation delimiter (pointed by register GL). As the collector is triggered just before executing a new call, that is, the calling arguments (if any) have already been put on the top of the stack, we should carefully handle these arguments during garbage collection. This involves two steps: collecting them during GC and restoring them after GC. The first step is done in the GC_INITIAL segment. We check each argument and push it onto the root stack if it must be collected. A special case is that an argument is a void variable. If so, we simply make a new void with the destination address. After processing an argument, the ST register moves down to the next argument. We repeat this process until all arguments having been examined. It seems that these arguments have been copied at the bottom of the copying area.

Now the initial root set is ready to be traced and the control comes to the GC_FETCH segment. First, we check if the root set is empty. If so, all useful results have been collected. Thus the COMPACTING segment will move the copied data back to the collecting region, restore calling arguments, and promote generation line. The argument handling method is simple and efficient, however, a shortcoming is that the set of arguments is duplicated upon each garbage collection: the collected set is at the bottom of copying area and must be restored on the top of the stack before returning back to the mutator. Fortunately, the number of arguments is usually very small (less than ten cells in most cases), the memory overhead for such duplication is negligible.

On the other hand, if the root set is not empty, register AX holds a traceable root. There are three possible cases with respect to AX:

- **hit_mid(AX):** AX is a pointer inside of the collecting region. This is because the collecting region might involve multiple generations, and each of them has an associated remembered set which records its own cross-generation pointers. Some roots remembered during execution might fall into the collecting region, i.e., they are not roots in the current old generations. Interestingly, generation delimiters involved in the root set are also falling into this test. We could kill two birds with one stone. Therefore, we simply drop this “root”, and go back to fetch the next one.

- **hit_choice(AX):** AX is a root remembered for GC as well as backtracking. A GC-root will be collected and discarded after collection, however, a backtrack-root must be collected and retained in the trail in case of later backtracking. Hence, we have to make an extra copy of AX for the potential backtracking. This is done simply by moving the bottom root to the top of the root stack (trail) and inserting AX into that vacated slot. It looks like that a copy of AX
is trailed again for backtracking and the current AX will be consumed with garbage collection.

- otherwise, AX is a root purely for garbage collection.

Now let us examine the GC_SHUNTING segment. When we reach here, register AX holds the root to be traversed (collected). It must be either a pointer to the old generation (such as a cross-generation root), or a pointer to the copied region (such as an argument root). This code segment will dereference the given root and carry out two possible optimizations: *partial variable shunting* and *root reversal*.

For a given root AX, we search for its final binding along its referencing chain. During dereferencing, if an intermediate binding BX is still a reference, we have four possible cases:

- BX hits an already copied variable. In our implementation, forwarded pointers are used to represent copied instances. A forwarded pointer is defined as a pointer to the copy region. As the copy region was the free area before garbage collection, it is impossible that the mutator has a reference points to that region. In other words, a forwarded pointer must be set by the collector. Thus AX is bound to the destination address of that copied variable. Go to GC_FETCH.

- BX hits a reference to the old generation. There is no need to traverse an instance in the old generation, because if there exist cross generation bindings in that old instance, those bindings must be remembered in the root set (some of them might have already been copied and some of them are to be collected). Thus AX is bound to this old instance. Go to GC_FETCH.

- BX is an unbound (a self-referencial) variable inside of the collecting region. First, we reverse BX towards its root AX. Next, we have to check the root AX: if AX is a root in the copied region, we make it unbound with its destination address (in this case, BX is forwarded to that copied variable); otherwise, AX must be a root in old generation, we make it to be unbound (in this case, BX is backwards to old generation). I call this method as root reversal, where the backward reversal only applies to the LVM architecture. The reason is straightforward: the LVM has only one type of variables - stack variables, and therefore has no constraint on the variable binding directions. Go to GC_FETCH.

- BX is an intermediate reference inside of the collecting region, we simply throw this reference away and replace BX with the next binding. This technique is the so called variable shunting. The reason I call this as partial variable shunting is because it only applies to reference chains inside of the collecting region, i.e., young generations.
This code segment stops traversing a given root and goes back to fetch the next root in the first three cases, or continues dereferencing until a non-variable binding being found and then goes to the GC_COPY segment.

In order to prevent duplication of shared live data during copying, [22, 35] adopt a variant of Cheney’s algorithm - a two-phase copying collector,:

**Phase 1**: Mark the live data.

**Phase 2**: For a given root, scan backward/forward to find a marked block and then copy the whole block and set up forward-pointers.

The elegance of Cheney’s algorithm is that copying collection can be made iterative, using just two pointers. One problem of this variant is that it involves two rounds of traversing of the live data. Another problem is that it requires two extra bits per heap cell: one for marking a copied as forwarded, and another for indicating a live internal cell.

CGC takes a different copying strategy by simulating a recursive algorithm [37], *i.e.*, using root stack to hold unvisited roots, transitively traversing each root and creating a copy along the traversing. It does not need a marking phase, and neither does it need any extra bits for GC purpose. In addition, recursive copying traverses the live data in depth-first order, which generally yields better locality than breadth-first order (such as the Cheney’s copying) for logic and functional programs [16]. The disadvantage is that the size of the root stack is bounded by the length of the longest path through the live data structure. However, as CGC incrementally collects small chunks (in most cases) of memory, the length of the longest path of the live data is bounded by the size of the collecting region.

Now, we come to the GC_COPY segment. At this point, AX holds the original root and BX holds its non-variable binding. We are faced with four possible cases with respect to the type and value of BX:

- **BX** is a constant. We assign BX to AX directly. Go to GC_FETCH.

- **BX** is a floating point reference. If it refers to a floating point value in the old generation, an assignment is make; otherwise, we construct a new floating reference pointing to a copied floating point value. Go to GC_FETCH.

- **BX** is a list/structure instance located in the old generation. Thus, an assignment is made. Go to GC_FETCH.

- **BX** is a list inside of the collecting region. We first check if it has been copied (see later discussion for more details). If yes, AX is assigned to this copied instance. Otherwise, AX is constructed by a new list term with its proper final destination address. Then we make a copy of the list, push the head root into
the root set (if necessary), set up tail root (last root optimization to get rid of a push-pop operation), replace the tail element to the forward pointer and continue collecting this tail root, *i.e.*, go to GC_SHUNTING.

- BX is a structure inside of the collecting region. We first check if the instance has been copied. We use the functor field for this purpose, that is, if the functor field is a forwarded pointer, then we simply assign the root with this copied instance. Otherwise, a copy must be made. The whole structure is copied in its original order. After copying the functor, the old functor cell is filled with the forward pointer to the copied instance. During copying arguments, new roots are added into the root set if necessary. In order to make the code compact, I did not use last root optimization. Thus, upon completion, control jumps back to GC_FETCH.

The collector algorithm is simple and easy to understand. However, a question remains to be discussed is: without a marking phase, does CGC create duplications?
A similar question is: how can we tell whether a live object has been or not yet been copied?

Let D be a live object with a possible type of variable, list, or structure. Constant type is omitted in our discussion because a live constant is always assigned to the root directly.

If D is a variable, there is no duplication. An important property of root reversal is that it prevents from creating duplication of a variable when the variable is shared by different roots. This property is illustrated by Fig. 12, where two cross-generation roots are collected in the order of R1 and then R2. An unbound variable X (where @X indicates X self-referential) is shared by R1 directly and R2 indirectly. Two cases are given: (a) Cheney's copying without marking - X is duplicated; and (b) CGC with root reversal - X is backwarded to @R1.

Generally speaking, from the collector algorithm we can easily figure out the following assertion:

A live variable D has not been copied iff D is inside of the collecting region and self-referential.

When D is first met, it is set either forward to an instance in the copied region or backward to the root in old the generation. A subsequent visit to D will fall to either hit_copied_var() or hit_old(), and in either case, no copy will be made. A potential advantage of root reversal is that there is no copy at all if a cross-generation root points to an unbound variable which resides in the collecting region and does not be shared as an internal object by other roots. For example, if R→@X then we have a shunted @R←X after collection, i.e., no copy is created in the copy region.

The second case is that D is a live structure instance. We assert that:

A live structure instance D has not been copied iff D's functor is not a forwarded pointer.

When D is first met, it is copied completely and its argument roots are added to the root set for further traverse. Its functor field is set forward to the copied instance at the same time the copy is made. Hence, a subsequent visit to D will fall into the truth branch of hit_copied_structure(). As a consequence, the root will be assigned to the copied instance directly and no further copy or traverse will be conducted.

The last case is that D is a live list instance. A list is recursively defined to be either empty or a node of the form [H|T] (read as the Head and the Tail) where T is a list. A list is therefore a chain of nodes linked by their T fields, and terminated by a nil - the empty list, or an unbound variable - an open end list. The term [H|T] corresponds to a cons pair in Lisp, and the head and tail correspond to car and cdr respectively. As a list does not have a functor field (in both LVM and WAM) for use to refer to a copied instance, we have to find another way for preventing from
duplicating a list node. The method we used here is to turn the tail field forward whenever a list instance is copied. Thus, we have an assertion as follows:

A live list instance $D$ of the form $[H|T]$ has not been copied if $T$ is not a forwarded pointer.

When $D$ is first met, it is copied and its two arguments, namely, $H$ and $T$ are added to the root set. After that, we set $T$ forward to its replica in the copy region.

However, this assertion is necessary but not sufficient, i.e., if $D$’s tail $T$ is a forward pointer, it does not imply that $D$ has been copied. This is because there are two possible cases where $T$ could be set forward: (i) a copy of $D$ has been made, or (ii) $T$ was an unbound variable and has been visited alone by another root. Unfortunately, these two interpretations are incompatible, i.e., forward pointers are overused by CGC in collecting list instances and root reversal.

Fig. 13 shows an example of the second case. Form this example, we can see that before collecting $[H|T]$, $T$ has been forwarded during collecting $R_1$. The list node referred by $R_2$ has not been copied yet even though its tail is a forward pointer. Fortunately, from the list definition, we know that the legal value for a tail field could only be one of the three term types: a nil, a sublist, or an unbound variable. The above case occurs only when $T$ is an unbound variable. In other words, if $T$ is a nil or a (sub)list term, it will not be set forward when it is visited alone by other roots.

Hence, by excluding the variable case, we have the following weak assertion:

A live list instance $D$ of the form $[H|T]$ has been copied if $T$ is a forwarded pointer but its copied replica is not a variable.
Figure 14: Duplicating a list node
Although this weak condition can not completely prevent from creating duplication of list nodes during garbage collection, it does rule out duplication in most cases, because the only case a live list node [H|T] might be duplicated is that T is an unbound and this node is visited by more than one root. Fig. 14 gives an example of this situation.

Suppose that we start GC with an initial root set \{R1, R2, R3\}. During collecting R1, the whole list is copied and tails of all list nodes have been set forward to the copied region. Then the collector will simply assign the copied sublist [H3, H4|@T] to R2 because the macro \texttt{hit\_copied\_list()} will return a truth value. However, ambiguity occurs when we meet R3: there is no enough information to figure out whether [H4|@T] has been copied or not. Thus we take a conservative but safe approach - making another copy. In this example, the last list node is thus duplicated. Fortunately, such situation happens on rare occasions and such duplication does not break the semantics of the user program.
7 Performance Analysis

An experimental LVM emulator has been implemented in C++. Ten benchmarks have been tested under the LVM system. Traveling Salesman Problem (tsp) and DNA Matching (dna) come from [22]. The dna program implements a dynamic algorithm for comparing one DNA sequence of length 32 against other DNA-sequences. The others are part of the Berkeley Benchmark suite which include N-Queens problem (queens), Native Reverse (nrev), Quick Sort (qsort), Build and Query a Database (browse), Serialise (serial), Recursive Integer Arithmetic (tak), Accumulative Reverse (reverse), and Boyer-Moore Theorem Prover (boyer). All benchmarks were tested with different input sizes. List inputs are generated by a random number generator written in Prolog. However, inputs to boyer are made up by the previous tautology in conjunction with a truth value in each subsequent test. In addition, the nondeterministic partition/ predicate in qsort has been changed to an if-then-else structure to make the benchmark deterministic. As we are still working on a LVM compiler, benchmarks are partly hand translated.

The testing platform is a SPARC Ultra-5-10 workstation with 16K D-cache, 1MB E-cache, 512MB main memory, sun4u CPU with 360MHZ clock. The LVM emulator is compiled by g++ 2.8.1 with the -O2 option. The CACHE_LIMIT used by the CGC algorithm is 1/3 of the E-cache size. For benchmark statistics, timing is measured in seconds and all memory related figures are in 32-bit words.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Max-stack</th>
<th>Max-trail</th>
<th>CGC</th>
<th>Garbage</th>
<th>Useful</th>
</tr>
</thead>
<tbody>
<tr>
<td>dna(100)</td>
<td>216K</td>
<td>9</td>
<td>100</td>
<td>16M</td>
<td>1.9K</td>
</tr>
<tr>
<td>boyer(1)</td>
<td>900K</td>
<td>99</td>
<td>16</td>
<td>1.4M</td>
<td>160K</td>
</tr>
<tr>
<td>nrev(2000)</td>
<td>833K</td>
<td>75</td>
<td>170</td>
<td>13M</td>
<td>762K</td>
</tr>
<tr>
<td>qsort(10K)</td>
<td>629K</td>
<td>31</td>
<td>14</td>
<td>1.4M</td>
<td>167K</td>
</tr>
<tr>
<td>queens(12)</td>
<td>1K</td>
<td>37</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rev(100K)</td>
<td>493K</td>
<td>3</td>
<td>13</td>
<td>713K</td>
<td>352K</td>
</tr>
<tr>
<td>serial(10K)</td>
<td>758K</td>
<td>1308</td>
<td>27</td>
<td>27M</td>
<td>226K</td>
</tr>
<tr>
<td>tak(21,16,8)</td>
<td>671K</td>
<td>8</td>
<td>100</td>
<td>15M</td>
<td>0.4K</td>
</tr>
<tr>
<td>tsp(100)</td>
<td>203K</td>
<td>17</td>
<td>3599</td>
<td>357M</td>
<td>584K</td>
</tr>
<tr>
<td>browse(10K)</td>
<td>732K</td>
<td>12</td>
<td>151</td>
<td>12M</td>
<td>581K</td>
</tr>
</tbody>
</table>

Table 7.1: Garbage Collection and Memory Consumption

Table 7.1 gives garbage collection and memory consumption statistics. Column Max-stack gives the maximum occupancy of the stack and Max-trail shows the maximum number of trail cells consumed in running each benchmark. Column CGC indicates the exact count of CGC invocations and Garbage shows the total amount of freed space. Here I shall point out that the actual memory required in running each benchmark with the given input size will never exceed the double of the corresponding Max-stack size, because a free area of the Max-stack size is more than enough to hold temporary copies during garbage collection. Space freed by back-tracking is not included in my statistics. Finally, column Useful gives the total amount of useful data being collected.

82
Now, let us examine the performance. What performance is expected for a good GC algorithm? To answer this question, I first take remarks from [16]: “The overall execution time for garbage collection typically ranges between a few percent to around 20 percent. ... If a ball-park figure had to be chosen, 10 percent would not be unreasonable for a well-implemented system.” However, this answer seems a little confusing, because the frame of reference, i.e., the program execution time, is not clearly defined. Here, I introduce a more feasible performance measurement. Let $T(P)$ be the execution time of a program $P$ on a machine with infinite virtual memory, $T(GC)$ be the time spend on garbage collection, and $T'(P)$ be the execution time of $P$ incorporated with garbage collection. Thus, taking the 10 percent ball-park figure into account, the following formula:

$$(1.1 \times T(P)) \geq (T'(P) + T(GC))$$

could be used as an expectation for a well-implemented garbage collection algorithm. From this consideration, I tested some benchmarks under LVM with CGC disabled. To be able to run these benchmarks, I have chosen some proper inputs so that their memory requirements fit the capacity of the virtual memory on the testing platform. Table 4 gives the execution times in CGC-enabled (E) and CGC-disabled (D) tests. All times are gathered by the Unix timing facility which returns usr/sys elapsed time. By adding theusr-time and sys-time together, the third row gives the ratio of E/D.

<table>
<thead>
<tr>
<th></th>
<th>dna_100</th>
<th>boyer_3</th>
<th>qsort_100K</th>
<th>serial_100K</th>
<th>brows_10K</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>2.15/0.0</td>
<td>9.06/0.1</td>
<td>4.61/0.0</td>
<td>7.21/0.1</td>
<td>69.85/0.0</td>
</tr>
<tr>
<td>D</td>
<td>2.51/0.7</td>
<td>9.06/1.8</td>
<td>4.52/1.1</td>
<td>7.18/1.6</td>
<td>79.02/0.7</td>
</tr>
<tr>
<td>E/D</td>
<td>67%</td>
<td>85%</td>
<td>97%</td>
<td>83%</td>
<td>88%</td>
</tr>
</tbody>
</table>

Table 7.2: CGC-enabled vs. CGC-disabled

The results in Table 7.2 are beyond our expectation. For these benchmarks,

$$T(P) \geq (T'(P) + T(GC))$$

that is, their performance with CGC is better than, or at least as good as running them on the same machine without conducting garbage collection. One significant reason is that the single stack paradigm incorporated with CGC improves program’s locality. In order to determine the extent to which the cache performance of the test programs has been improved, the usr-times can be compared. Clearly, the time spend on garbage collection is almost absorbed by the improved program’s cache performance. Moreover, as evidenced by the comparison of sys-times, CGC greatly reduces page-faults in virtual memory. The second important reason is that the CGC algorithm is very efficient. For programs involving both short-lived and long-lived objects, CGC serves as a self-purifying memory manager which periodically collects
useful data. For programs involving only short-lived variables, such as tak, CGC implements lazy stack deallocation. It is not true to say that CGC is better than LCO in this case, because the latter offers the best cache performance. However, after having tested tak under some stack/heap-based emulator (see Table 7.3), I found that the performance penalty caused by CGC is trivial.

<table>
<thead>
<tr>
<th></th>
<th>input</th>
<th>20</th>
<th>22</th>
<th>24</th>
<th>26</th>
<th>28</th>
<th>30</th>
<th>32</th>
<th>LVM/SICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>tak</td>
<td></td>
<td>0.25</td>
<td>0.85</td>
<td>2.39</td>
<td>5.66</td>
<td>13.14</td>
<td>25.53</td>
<td>48.67</td>
<td>1.0</td>
</tr>
<tr>
<td>S_J</td>
<td>LVM</td>
<td>0.28</td>
<td>0.9</td>
<td>2.42</td>
<td>5.82</td>
<td>12.60</td>
<td>25.63</td>
<td>48.8</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>SJ</td>
<td>0.05</td>
<td>0.18</td>
<td>0.48</td>
<td>1.15</td>
<td>2.52</td>
<td>5.08</td>
<td>9.85</td>
<td>4.95</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>input</th>
<th>30</th>
<th>50</th>
<th>70</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>255</th>
<th>LVM/SICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsp</td>
<td></td>
<td>0.33</td>
<td>2.28</td>
<td>8.35</td>
<td>33.49</td>
<td>64.67</td>
<td>525</td>
<td>1.413</td>
<td>1.46</td>
</tr>
<tr>
<td>LVM</td>
<td>0.48</td>
<td>3.43</td>
<td>12.68</td>
<td>51.05</td>
<td>233.7</td>
<td>785</td>
<td>2.096</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>SJ</td>
<td>0.1</td>
<td>0.67</td>
<td>2.44</td>
<td>9.94</td>
<td>51.16</td>
<td>149</td>
<td>304</td>
<td>5.22</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>input</th>
<th>14</th>
<th>16</th>
<th>18</th>
<th>20</th>
<th>22</th>
<th>24</th>
<th>26</th>
<th>LVM/SICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>queens</td>
<td></td>
<td>0.09</td>
<td>0.57</td>
<td>2.81</td>
<td>15.91</td>
<td>62.89</td>
<td>47.22</td>
<td>54.26</td>
<td>1.09</td>
</tr>
<tr>
<td>LVM</td>
<td>0.1</td>
<td>0.65</td>
<td>3.08</td>
<td>17.13</td>
<td>74.47</td>
<td>51.58</td>
<td>59.36</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>SJ</td>
<td>0.02</td>
<td>0.1</td>
<td>0.49</td>
<td>2.81</td>
<td>28.1</td>
<td>8.1</td>
<td>9.54</td>
<td>6.22</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>input</th>
<th>1K</th>
<th>2K</th>
<th>3K</th>
<th>4K</th>
<th>5K</th>
<th>10K</th>
<th>12K</th>
<th>LVM/SICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>now</td>
<td></td>
<td>0.26</td>
<td>0.98</td>
<td>2.28</td>
<td>4.15</td>
<td>6.69</td>
<td>27.79</td>
<td>41.99</td>
<td>2.24</td>
</tr>
<tr>
<td>LVM</td>
<td>0.58</td>
<td>2.41</td>
<td>5.3</td>
<td>9.62</td>
<td>15.26</td>
<td>65.1</td>
<td>94.22</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>SJ</td>
<td>0.11</td>
<td>0.35</td>
<td>0.86</td>
<td>1.55</td>
<td>2.53</td>
<td>11.52</td>
<td>18.85</td>
<td>5.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>input</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>LVM/SICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>dig</td>
<td></td>
<td>0.42</td>
<td>2.64</td>
<td>7.94</td>
<td>22.97</td>
<td>69.35</td>
<td>216.21</td>
<td>690</td>
<td>1.19</td>
</tr>
<tr>
<td>LVM</td>
<td>0.53</td>
<td>3.11</td>
<td>9.08</td>
<td>26.21</td>
<td>75.68</td>
<td>253</td>
<td>823</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>SJ</td>
<td>0.16</td>
<td>1.07</td>
<td>3.25</td>
<td>9.54</td>
<td>28.99</td>
<td>90.61</td>
<td>289</td>
<td>2.85</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>input</th>
<th>1K</th>
<th>5K</th>
<th>10K</th>
<th>100K</th>
<th>1M</th>
<th>1.5M</th>
<th>2M</th>
<th>LVM/SICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>laye</td>
<td></td>
<td>0.02</td>
<td>0.1</td>
<td>0.22</td>
<td>3.79</td>
<td>156.11</td>
<td>320.04</td>
<td>515</td>
<td>1.27</td>
</tr>
<tr>
<td>LVM</td>
<td>0.03</td>
<td>0.15</td>
<td>0.28</td>
<td>4.61</td>
<td>491.48</td>
<td>413.3</td>
<td>697</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>SJ</td>
<td>0.01</td>
<td>0.04</td>
<td>0.08</td>
<td>1.56</td>
<td>59.26</td>
<td>113.54</td>
<td>182</td>
<td>3.83</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>input</th>
<th>1K</th>
<th>10K</th>
<th>100K</th>
<th>1M</th>
<th>4M</th>
<th>5M</th>
<th>LVM/SICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td></td>
<td>0.06</td>
<td>0.70</td>
<td>8.06</td>
<td>114.78</td>
<td>337.9</td>
<td>244.0</td>
<td>*</td>
</tr>
<tr>
<td>LVM</td>
<td>0.06</td>
<td>0.65</td>
<td>7.21</td>
<td>56.26</td>
<td>317.7</td>
<td>433</td>
<td>548</td>
<td>1.0</td>
</tr>
<tr>
<td>SJ</td>
<td>0.02</td>
<td>0.32</td>
<td>4.27</td>
<td>76.0</td>
<td>224.82</td>
<td>299</td>
<td>*</td>
<td>1.45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>input</th>
<th>10K</th>
<th>1M</th>
<th>5M</th>
<th>6M</th>
<th>7M</th>
<th>8M</th>
<th>LVM/SICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>reverse</td>
<td></td>
<td>0.02</td>
<td>0.19</td>
<td>4.35</td>
<td>22.12</td>
<td>28.77</td>
<td>34.59</td>
<td>41.63</td>
</tr>
<tr>
<td>LVM</td>
<td>0.02</td>
<td>0.28</td>
<td>2.88</td>
<td>13.71</td>
<td>16.63</td>
<td>19.4</td>
<td>22.13</td>
<td>1.0</td>
</tr>
<tr>
<td>SJ</td>
<td>0.0</td>
<td>0.12</td>
<td>3.32</td>
<td>17.9</td>
<td>23.79</td>
<td>30.13</td>
<td>34.98</td>
<td>(0.63)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>input</th>
<th>100</th>
<th>1K</th>
<th>10K</th>
<th>50K</th>
<th>70K</th>
<th>100K</th>
<th>200K</th>
<th>LVM/SICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>browse</td>
<td></td>
<td>0.47</td>
<td>4.73</td>
<td>32.12</td>
<td>403.17</td>
<td>714.69</td>
<td>1397</td>
<td>4925</td>
<td>(0.50)</td>
</tr>
<tr>
<td>LVM</td>
<td>0.67</td>
<td>7.2</td>
<td>72.06</td>
<td>430.98</td>
<td>689.35</td>
<td>1060</td>
<td>2908</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>SJ</td>
<td>0.20</td>
<td>1.95</td>
<td>21.99</td>
<td>261.38</td>
<td>914.93</td>
<td>1400</td>
<td>4309</td>
<td>(0.64)</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3: LVM vs. SICStus

84
Next, we compare the LVM with the SICStus Prolog 3.8.5 - the most recent, most popular high-performance (commercial) Prolog system. All benchmarks are compiled to compact code ($S_c$) and fast code ($S_f$) and tested by SICStus 3.8.5 on the same platform. Table 7.3 gives the execution times gathered from SICStus and the LVM emulator respectively.

In fact, the current LVM emulator is just an experimental, low-efficiency version. It is written in ANSI C++ with a case-switch engine (10%-30% slower than a threaded-code engine). As there are no register variables in the LVM model, arithmetic instructions have to use stack-oriented computation, and they are designed to cope with both integer and floating point operations. Furthermore, built-in predicates, such as functor/3, arg/3 ..., are implemented by stand-alone functions which perform poorer than inline built-in functions. On the other hand, SICStus is a mature implementation, the product of many person-years of skilled labor, much of it directed at tuning the emulator. This means that emulated code is not so easy to improve upon as one might suppose [25].

<table>
<thead>
<tr>
<th></th>
<th>input</th>
<th>100</th>
<th>200</th>
<th>1K</th>
<th>10K</th>
<th>100K</th>
<th>500K</th>
</tr>
</thead>
<tbody>
<tr>
<td>SICStus</td>
<td>1</td>
<td>6</td>
<td>13</td>
<td>45</td>
<td>315</td>
<td>801</td>
<td>1,712</td>
</tr>
<tr>
<td>LVM</td>
<td>1</td>
<td>6</td>
<td>13</td>
<td>45</td>
<td>315</td>
<td>801</td>
<td>1,712</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>input</th>
<th>1K</th>
<th>10K</th>
<th>100K</th>
<th>1M</th>
<th>3M</th>
<th>4M</th>
<th>5M</th>
</tr>
</thead>
<tbody>
<tr>
<td>SICStus</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>32</td>
<td>40</td>
<td>43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LVM</td>
<td>1</td>
<td>27</td>
<td>292</td>
<td>3,157</td>
<td>9,500</td>
<td>15,866</td>
<td>17,450</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>input</th>
<th>1K</th>
<th>5K</th>
<th>10K</th>
<th>100K</th>
<th>1M</th>
<th>1.5M</th>
<th>2M</th>
</tr>
</thead>
<tbody>
<tr>
<td>SICStus</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>216</td>
<td>356</td>
<td>561</td>
<td></td>
</tr>
<tr>
<td>LVM</td>
<td>0</td>
<td>6</td>
<td>14</td>
<td>243</td>
<td>11,861</td>
<td>27,927</td>
<td>49,188</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>input</th>
<th>1K</th>
<th>100K</th>
<th>1M</th>
<th>5M</th>
<th>6M</th>
<th>7M</th>
<th>8M</th>
</tr>
</thead>
<tbody>
<tr>
<td>SICStus</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>6</td>
<td>16</td>
<td>17</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>LVM</td>
<td>0</td>
<td>13</td>
<td>145</td>
<td>731</td>
<td>877</td>
<td>1,024</td>
<td>1,170</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>input</th>
<th>1K</th>
<th>10K</th>
<th>50K</th>
<th>70K</th>
<th>100K</th>
<th>200K</th>
</tr>
</thead>
<tbody>
<tr>
<td>SICStus</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>20</td>
<td>18</td>
<td>27</td>
<td>25</td>
</tr>
<tr>
<td>LVM</td>
<td>0</td>
<td>12</td>
<td>151</td>
<td>768</td>
<td>1,077</td>
<td>1,540</td>
<td>3,082</td>
</tr>
</tbody>
</table>

Table 7.4: GC Invocations - LVM vs. SICStus

As expected, SICStus emulator should be faster than the LVM emulator, especially for benchmarks involving intensive arithmetics and built-in calls. The expected performance ratios are that the LVM should be 10%-30% slower than $S_c$, and 3 to 7 times slower than $S_f$. Examining Table 7.3, SICStus is faster for all benchmarks when the input size is small. When we increase the input size, some benchmarks keep the same performance ratio whereas some greatly narrow the performance gap and at certain breakthrough points they perform better than their counterparts under
SICStus. One factor which influences the SICStus performance is garbage collection. Taking two Sf examples, among the 299 seconds for executing `serialise(4M)` and 34.98 for `reverse(8M)`, 149 and 31.97 seconds are spend on garbage collection respectively. That is, the performance penalty caused by SICStus garbage collection reaches up to 50 and 91 percent for these special cases. Further analysis of these benchmarks are to be done. Specifically, I need to find the reasons of why the LVM performs worse than I expected on the `nrev` and why the SICStus performs so poor on `browse`.

Table 7.4 shows the number of GC-invocation under two systems. The LVM invokes garbage collection much more frequently than the SICStus does. In order to see the CGC-related memory overhead, Table 7.5 gives the maximum trail usage of some benchmarks. As those benchmarks are deterministic programs (except `serial` whose trailed roots include some shallow backtracking bindings), I found that the maximum consumption of the trail with respect to CGC is virtually independent of the benchmark input size. Namely, the trail used for remembered sets is not linear to the input size, instead, it varies in a very small range.

<table>
<thead>
<tr>
<th></th>
<th>input</th>
<th>20</th>
<th>100</th>
<th>200</th>
<th>1K</th>
<th>10K</th>
<th>100K</th>
<th>500K</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LVM</td>
<td>10</td>
<td>9</td>
<td>12</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>serial</td>
<td>input</td>
<td>1K</td>
<td>10K</td>
<td>100K</td>
<td>1M</td>
<td>3M</td>
<td>5M</td>
<td>5.5M</td>
</tr>
<tr>
<td></td>
<td>LVM</td>
<td>338</td>
<td>1.308</td>
<td>2.140</td>
<td>7.214</td>
<td>13.330</td>
<td>17.385</td>
<td>17.608</td>
</tr>
<tr>
<td>quot</td>
<td>input</td>
<td>1K</td>
<td>5K</td>
<td>10K</td>
<td>100K</td>
<td>1M</td>
<td>1.5M</td>
<td>2M</td>
</tr>
<tr>
<td></td>
<td>LVM</td>
<td>8</td>
<td>22</td>
<td>31</td>
<td>44</td>
<td>50</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>reverse</td>
<td>input</td>
<td>10K</td>
<td>100K</td>
<td>1M</td>
<td>5M</td>
<td>6M</td>
<td>7M</td>
<td>8M</td>
</tr>
<tr>
<td></td>
<td>LVM</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>layer</td>
<td>input</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>LVM</td>
<td>99</td>
<td>127</td>
<td>147</td>
<td>147</td>
<td>158</td>
<td>158</td>
<td>158</td>
</tr>
<tr>
<td>browse</td>
<td>input</td>
<td>100</td>
<td>1K</td>
<td>10K</td>
<td>50K</td>
<td>70K</td>
<td>100K</td>
<td>200K</td>
</tr>
<tr>
<td></td>
<td>LVM</td>
<td>0</td>
<td>13</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 7.5: Trail Usage for Remembered Sets

I also gathered detailed statistics, such as the size of collecting region, the number of copied objects, the space being reclaimed, and the size of the trail before and after each round of garbage collection. I found that between 85 and 99 percent of all objects in the young area die within each collection in most tests. This observation further supports the weak generational hypothesis. A set of figures are given in the next section. These figures show the memory utilization of benchmarks with certain inputs.

Note that the measurements and analysis presented here are not universally applicable, more experiments with a wide range of applications must be tested.
Figure 15: Memory Utilization of Benchmarks
Figure 16: Serialise (100K)

'serial100k.data'
Figure 18: Accumulative Reverse (1M)
Figure 19: Traveling Salesman (\textit{tsp50.data})
Figure 21: Boyer-Moore Theorem Prover (4)
8 Conclusion and Future Work

In this report, I presented the specification of a new Prolog execution model - the LVM. It is simple with a small, clean instruction set. It supports coarse-grain two-stream unification which facilitates both byte-code emulation and native-code compilation. It explores the merged heap/stack paradigm for all dynamical memory requirements and embeds CGC, an efficient garbage collector, as an (conditional) inline function of the execution engine. CGC exploits a new policy in the spectrum of generational garbage collection. It introduces the concept of chronological generation to divide execution environment into dynamic generations. It rests upon the cache size to control generation creation and GC invocation. It ingeniously uses the existing trail mechanism to manipulate generation information and remembered sets. Benchmarks show that CGC improves the program’s cache performance (almost) enough to pay its own cost. Although CGC is used in a Prolog execution model, results of this research might be useful in related disciplines of functional, logic, as well as object-oriented programming.

Study on this subject is now being concentrated on two ongoing projects: a LVM compiler and a LVM cache-performance simulator. On the outside, the LVM is just a simplified WAM. Hence, it is expected that a fine-tuned LVM emulator should be competitive with any WAM-based emulator for performance, and most WAM-based compilation techniques and optimizations can be directly adopted. In the development of the LVM compiler, however, two CGC-related issues might require further investigation: how to minimize the initial root set and how to prevent CGC from collecting data which will never be used in the subsequent execution.

As the gap of speed between processor and DRAM memory widens, cache performance is becoming ever more important in language implementation. In order to determine the improvement of cache performance, we are developing a special emulator with respect to our new LVM specification. To cover typical cache implementations, a large of cache parameters should be considered. This emulator will do the trace-driven simulation for directly-mapped and set associative caches with different write miss policies. The objectives of this simulation are to verify and validate our experimental results, and to find important factors which influence the performance of CGC.

Another future project is to extend the LVM model to support multithreading and mobile agents. Mobile Agents are mainly intended to be used for network computing - applications distributed over large scale computer networks. The motto of Mobile Agents is: move the computations to the data rather than the data to the computations. In general, a mobile agent is a self-contained process that can autonomously migrate from host to host in order to perform its task on behalf of a (human) user. Numerous Mobile Agents systems have been implemented or are currently under development.
The objective of this new research is to present an alternative model in the
design space of Intelligent Mobile Agents. To achieve this, we need to extended
the LVM to cope with new issues, such as explicit concurrency, code autonomy,
communication/synchronization and computation mobility. Memory management
considerations play a determining role not only in shaping multithreading strategy,
but also in establishing computation mobile policy. The merged stack/heap archi-
tecture of the LVM makes it much easier and simpler to control autonomous agent
threads and implement agent migration by capturing execution state at thread-level.
Thus, the destination host can reactivate the thread at precisely the point where
migration was initiated, which can be useful for automatic load-balancing or for
fault-tolerant programs. For this long term project, we need to develop a compiler
for an extended Prolog, implement different experimental emulators to verify the
virtual machine architecture and to carry out performance (time/space) analysis on
various benchmarks and platforms.

References


[7] P. Codognet and D. Diaz. wamcc: Compiling Prolog to C. PLILP'95, The


