Efficient Memory Management in a Merged Heap/Stack Prolog Machine

Xining Li
Department of Computer Science
Lakehead University
Thunder Bay, Canada
xli@flash.lakeheadu.ca

ABSTRACT

Traditional Prolog implementations are based on the stack/heap memory architecture: the 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. In this paper, I present a novel memory management approach used in the implementation of Logic Virtual Machine (LVM). The LVM combines the stack and the heap into a single memory block for all dynamical memory requirements, supports coarse-grain two-stream unification, and embeds an efficient garbage collection algorithm, the Chronological Garbage Collection (CGC), to reclaim useless memory cells. An experimental LVM emulator 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 during garbage collection. Some 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. INTRODUCTION

Most Prolog implementations are based on the stack/heap architecture. For example, the WAM [22][1] stores execution environment 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 *last call optimization* (LCO) and *environment trimming* possible. However, heap is allocated on-the-fly. Useless heap space can only be reclaimed on backtracking or by garbage collection. Conventional garbage collection methods may perform poorly. The reason for using stack/heap architecture is that deallocating stack frames is cheap, incremental garbage collection.

A number of alternative garbage collection algorithms have been used over the years. The most popular ones are *reference-counting*, *mark-sweep*, *copying*, *mark-compact*, and *generational* [14]. Some of them have been adopted in various Prolog implementations. For example, SICStus Prolog uses a mark-compact algorithm embedded with variable shunting during collection [18]. Bruynooghe [6] 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 [13] presented a new data structure, attributed variable, combined with a memory management machine, MALI, which encapsulates a garbage collector. Taki [19] 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 [5] exploited a copying algorithm and concluded that copying collection is a viable alternative to the conventional mark-compact algorithm for Prolog. Appleby and *et al* [2] discussed how to apply generational technique to the WAM-based Prolog implementation. Bekkers, Ridoux and Ungaro [4] presented a good survey which gives insight into memory management problems in sequential logic programming language implementations. Demoen, Engels and Tarau [8] implemented a bottom-up copying collector in the context of Bin-Prolog and shown its outstanding practical performance.

In this paper, I present a novel memory management approach used in the implementation of Logic Virtual Machine (LVM). The LVM combines the traditional stack and heap into a single stack for all dynamical memory requirements and cooperates with an efficient garbage collection algorithm, the Chronological Garbage Collection (CGC), to reclaim useless memory cells. CGC draws advantages from copying, mark-compact, generational, and incremental algorithms. Our benchmarks show that the proposed approach has low runtime overhead, good virtual memory and cache performance, and very short, evenly distributed pause times. Our experimental results also revealed that the CGC improves the program's cache performance almost enough to pay its own cost. In some special cases, the LVM interpreter outperforms SICStus Prolog in fast-mode.

PPDP'00, Montreal, Canada.

This paper is organized as follows. In Section 2, I briefly discuss the LVM memory architecture and instruction set. In Section 3, I present an overview of the Chronological Garbage Collection algorithm, focusing on the generation division and the initial root set specification. I proceed to study the performance of the LVM system. Finally, I present the conclusion and outline of future work.

2. AN OVERVIEW OF THE LVM

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.[16]" 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 was built on this paradigm. However, if we followed the trace of the first Prolog implementation, we would time-warped 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) it 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 which facilitates both backtracking and garbage collection; and 5) two-stream unification can be easily implemented at the coarse-grain instruction level.

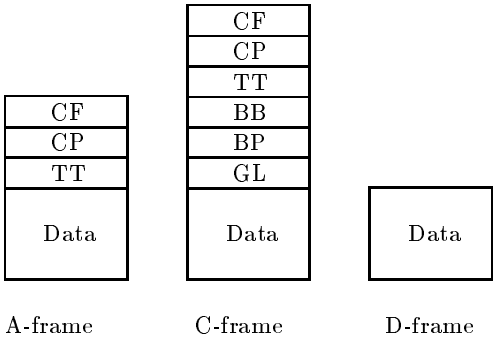

Figure 1: Stack Frames

The LVM allocates a block of storage to hold execution information. The whole block is an addressable array of memory cells and is divided into four segments (from low to high): the *symbol table* which stores various kinds of information about symbols; the *code area* which holds the byte-code of a program; the *stack* which contains all kinds of data, such as variables, structures, control information, choice points, *etc.*, allocated during execution; and the *trail* which saves two types of addresses: addresses of bound variables which must be unbound upon backtracking and addresses of roots which must be collected upon garbage collection.

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

Table 1: System Registers

The LVM adopts the WAM's tag system to represent Prolog terms and defines ten system registers (Table 1) to represent system internal state. These registers will be used and maintained by the LVM run time system. The stack is allocated upwards whereas the trail grows downwards. If the stack top pointer ST and the trail top pointer TT meet, the memory is exhausted. 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 CP and CF registers are used to implement long return (from a chain of calls). The GP register is introduced for efficiency purpose: a sequence of unification instructions can use GP to access their operands if they are located in contiguous memory cells. The BB register is for backtracking. It holds the root of a linked list of choices. The GL register is novel and plays a key role in garbage collection. It holds the most recent *generation line*. 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 to implement conditional jumps in two-stream unification.

The LVM defines three kinds of stack frames: activation (A-

frame), choice (C-frame) and data (D-frame) (see Fig. 1). A C-frame consists of a data portion for input arguments and a six-cell backtrack control portion. It is allocated when a choice point is met. The LVM handles backtracking the same way as the WAM does. A D-frame, if necessary, is created upon a call to the type of clauses which do not have a continuation point (such as facts and single-goal clauses). An A-frame consists of a data portion and a three-cell expansion for control. Control information includes the continuation point CP, the continuation frame pointer CF and the current trail top TT.

The LVM instruction set is given in Table 2. Generally speaking, the control set includes instructions for stack allocation, procedure invocation, control dispatching, and nondeterministic manipulation. Unification instructions are annotated with different types. Let V(n) represent a variable, L(n) a list, and S(n) a structure, at nth position of the current frame. For the unification set, **put**'s are used for setting up calling arguments, **get**'s are used to unify source terms with different annotated terms, and **set**'s are used to construct new instances of compound terms.

Control Instructions		
alloa	n1 n2	allocate an A-frame n1: # of arguments n2: # of additional cells
allod	n1 n2	allocate a D-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
try	n e	allocate a C-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
Unification Instructions		
putvoid		put a void var
putvar	n	put V(n) and make V(n) unbound
putval	n	put V(n)'s value
putcon	c	put constant c
putlist	n	put L(n)
putstr	n	put S(n)
getval	n1, n2	unify V(n1) and V(n2)
getcon	n, c	unify V(n) and c
getlist	n1, n2, n3, e	unify V(n1) with L(n2) n3: nesting level e: branch if V(n1) is an unbound
getstr	n1, n2, n3, e, f	unify V(n1) with S(n2) n3: nesting level e: branch if V(n1) is an unbound f: functor of the structure
setvar	n	set V(n) unbound
setval	n1, n2	set V(n1) to V(n2)
setcon	n, c	set V(n) to c
setlist	n1, n2	set V(n1) to L(n2)
setstr	n1, n2	set V(n1) to S(n2)
setfun	n, f	set V(n) to f
branch	e1, e2, e3, e4	branch wrt NL's value

Table 2: The LVM Instructions

The basic formats of clause translation are given below. 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 → is a relation of *followed by*.

1. Fact clause:
   [allod] → **get** * → **proceed**
2. Chain clause:
   [allod] → **get** * → **put** * → **chaincall**
3. Rule clause: last goal is a user defined goal
   alloa → **get** * → { **put** * → **call** }+ → **put** * → **lastcall**
4. Rule clause: last goal is a system predicate
   alloa → **get** * → { **put** * → **call** }+ → **return**
5. Query:
   { **put** * → **call** }+

Readers who are familiar with the WAM might find that the LVM instructions bear a strong resemblance to the WAM instructions. It is true that a WAM compiler can be easily transformed to a LVM compiler with some minor modifications. However, there are some major differences in their working principle. First, for each matching clause the LVM allocates a stack frame big enough to hold all its dynamic objects (the number of dynamic objects is the amount of flattened terms of that clause). The LVM compiler simply assumes that it has an infinite memory to use, with no concern for memory efficiency. It does not do LCO, nor does it do environment trimming. There is no need to verify binding direction of two variables. There are no *unsafe* variables. Secondly, the WAM passes calling arguments through a set of soft registers, whereas the LVM passes the calling arguments directly to the callee's stack frame (this scheme was first adopted by the NTOAM [24]). In fact, the LVM eliminates the concepts of register variables and heap variables. Only one kind of variables, *i.e.*, stack variables, is used in the LVM model.

Like the WAM, the LVM generates a

$$\textbf{try} \rightarrow \textbf{retry}* \rightarrow \textbf{trust}$$

sequence to handle a nondeterministic procedure. 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 re-put these arguments on the top of the newly allocated choice in order to invoke the trying clause. The **trust** instruction will deallocate the choice frame. As a consequence, arguments are exposed to the last alternative clause.

Another interesting feature of the LVM which differs from the WAM in handling unification is that the LVM implements two stream unification at the coarse-grain level. 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, 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. Marien and Demoen[15] shown how to improve WAM instructions to facilitate two stream unification. Haygood[12] presented an example of two stream code generated by the SICStus compiler, and Van Roy[21] gave a good survey of this algorithm. They all point 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. To illustrate this, let us examine how **getstr** is implemented:

```
getstr n1, n2, n3, e, f
t = dereference(V(n1));
if (tag(t) == REF) {
      *t = make_structure(n2);
      trail(t);
      NL = n3; // assign nesting level to register NL
      goto e; // jump to write stream
}
else if ((tag(t) == STR) && (t→ functor == f)) {
      for (i = 1; i ≤ arity(f); i++) {
            V(n2 + i) = t→argument[i];
      }
      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 and control is transferred to the write stream labeled by *e* where a sequence of **set** instructions will construct the annotated term. Operand *n3* is an integer representing the *nesting level* of the annotated term: 0 for a top level term (a calling argument), 1 for the first nesting level subterms, and so on. 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. On the contrary, the LVM's **getstr** instruction sets up arguments of the matching structure as part of its own function. There is no need to have a **getvar** instruction. Furthermore, the LVM has no **unify**-like instructions at all. The set of **get** instructions is sufficient for coding read stream programs.

In the original WAM, there are no **set** instructions. Ait-Kaci introduced **set** instructions in his WAM introductory book[1]. He indicates that all **set** instructions are equivalent to **unify** instructions in write mode, and more efficient to construct terms after **put** instructions. In other words, their **set** instructions are used *before* unification taking place. The LVM goes one step forward: **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.

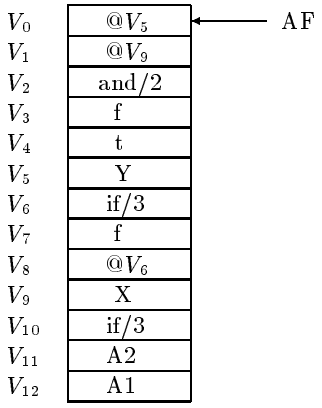

Figure 2: Example: Stack Frame Layout

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 **branch** instructions is able to cope with dispatching points of arbitrary nesting levels. Now, let us consider the following example:

eq(if(X, if(Y, t, f), f), and(X, Y)).

In this example, terms are flattened to eleven distinct variables. Since the clause is a fact with two input arguments, an **allod 2, 11** will be generated to allocate a data frame whose layout is given by Fig. 2, where V_0 to V_{10} represent flattened terms, V_{11} and V_{12} are two calling arguments. Thus, the LVM compiler generates the following two stream code:

Example: the LVM code

```
eq/2:   allod     2, 11                    % eq/2 entry
                                           % READ STREAM
        getstr    12, 10, 0, if/3, w.0     % unify A1 with if/3
        getstr    8, 6, 1, if/3, w.1       % unify nesting if/3
        getcon    4, t                     % unify t in nesting if/3
        getcon    3, f                     % unify f in nesting if/3
r.0:    getcon    7, f                     % unify f in if/3
r.1:    getstr    11, 2, 0, and/2, w.2     % unify A2 with and/2
        getval    1, 9                     % unify X in and/2
        getval    0, 5                     % unify Y in and/2
        proceed
                                           % WRITE STREAM
w.0:    setfun    10, if/3                 % set if/3 functor
        setvar    9                        % set X in if/3
        setstr    8, 6                     % set nesting if/3 in if/3
        setcon    7, f                     % set f in if/3
w.1:    setfun    6, if/3                  % set nesting if/3 functor
        setvar    5                        % set Y in nesting if/3
        setcon    4, t                     % set t in nesting if/3
        setcon    3, f                     % set f in nesting if/3
        branch    r.1, r.0, fail, fail     % if NL = 0 to r.1
                                           % if NL = 1 to r.0
w.2:    setfun    2, and/2                 % set and/2 functor
        setval    1, 9                     % set X in and/2
        setval    0, 5                     % set Y in and/2
        proceed
```

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 [21] pointed out, some bookkeeping overhead is necessary. Detailed examples were given in [15] which shown the 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 **get** instruction for each node along the traversing to form the read stream. In generating write stream, however, **set** instructions are coded in breadth-first only if the visited node is a compound term. As a result, the two stream code generation algorithm (the core algorithm) of the LVM compiler is implemented in about 100 lines of C++ code.

3. 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, this assumption 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 interdependencies, automatic storage reclamation is essential for practical implementations. Moreover, the LVM's single stack 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.

What make a good garbage collection algorithm? Generally speaking, 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. Unfortunately, none existing GC algorithm 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.

Furthermore, 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 asymptotic time complexity is proportional to the size of the entire heap H, *i.e.*, O(H).

Copying: Althrough 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. [5] 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, [8] 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 behavior is that the write boundary may left 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 [20]: "... 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 [3]. 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 untenured.

Based on the study of existing GC algorithms, I developed a new algorithm: Chronological Garbage Collection. CGC combines advantages of generational, copying, mark-compact, and incremental algorithms. Based on the weak generational hypothesis, CGC introduces a concept of *chronological generation* - a dynamical way to create generations. Like copying, CGC traverses from a small set of roots and copies live objects onto a free space. From mark-compact, CGC borrows the idea that at the end of collection, the stack will be compacted into two continuous area: one for active objects and the other for free cells. Finally, CGC controls the frequency of collector invocations by a factor of cache size, and therefore collects garbage incrementally with a trivial pause time.

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 control stack. Namely, the stack is organized as a linked list through a *Continuation Frame* (CF) slot. Different from the WAM which discards the current stack frame *before* issuing the last call, the LVM only resets the continuation environment (CF and CP) 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. We can not discard a finished frame because it may hold long lived data objects, however, we can safely apply a generational garbage collection process to this young generation (the finished frame) provided that it is not frozen by some choice point.

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 bindings (roots) in the forward 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. 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 [23, 17, 25, 11]. We have simulated the cache performance of CGC based on an old experimental version of LVM[9]. 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 our 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, it is expected that the improved cache performance could pay off the cost of CGC.

As it turns out, a simple comparison

$$((\mathrm{ST} - \mathrm{an\_old\_generation\_line}) \geq \mathrm{CACHE\_LIMIT})$$

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.

/* Generic Memory Allocation Algorithm */

```
if (CF > GL){
      // CF is in the nursery space
      if ((ST − GL) ≥ CACHE_LIMIT){
            GL = ST;
            new_generation();
      }
      make_X_frame();
      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);
            cgc();
      }
      make_X_frame();
      goto next_instruction;
}
```

Now, we are ready to continue with implementation de-

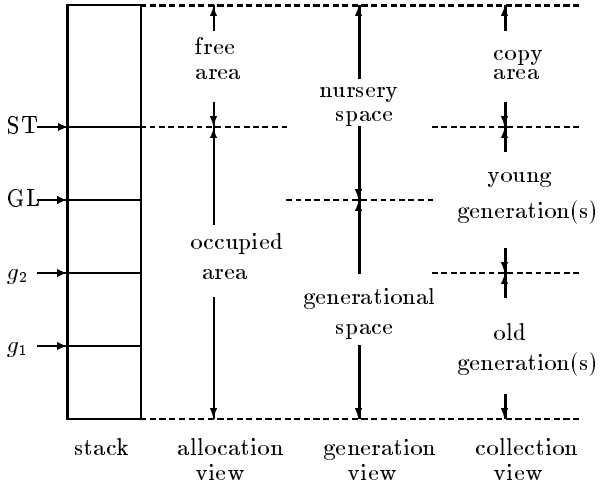

Figure 3: Stack Division and Views

tails. The LVM defines three memory allocation instructions, *i.e.*, **alloa**, **allod** and **try**. In fact, these instructions not only allocate their specified stack frames, but also determine whether a new generation should be created, or a round of garbage collection should be triggered. Assume 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 same *generic memory allocation algorithm.*

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. 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. 3 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 $\geq$ 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.

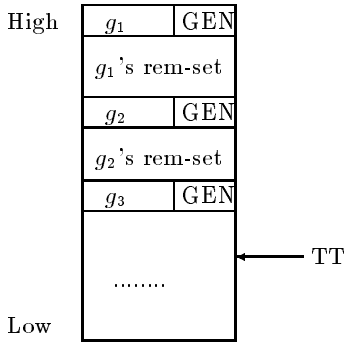

Figure 4: Trail Layout

(CF $\leq$ 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, **alloa** 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.

Now, 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:

$$*\mathrm{TT}--= \mathrm{GL} \mid \mathrm{GEN};$$

As a consequence, the trail is segmented by generation lines, see Fig. 4. 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. 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 after g_i must have their generation search pointers less than

t_i, and they all belong to generation g_j. Thus, for an arbitrary control/choice frame which falls in the generational space, its associated generation line can be found by a call *get_gline(Frame→TT)* to the following function:

```
int* get_gline(int* t){
        while (tag(*t) != GEN) t--;
        return t;
}
```

In generational garbage collection, pointers going forward in time (that is 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*. A write barrier is an inline function which traps and filters each eligible pointer assignment and inserts forward-in-time pointers into a remembered set.

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. In the WAM, *trail* and *detrail* operations are introduced to handle nondeterministic computation. Trail operation is used to record variable bindings if they were unbound before creation of the current choice, and detrail operation 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. As mentioned before, register GL is always greater than or equal to BB. Hence, a generalized trail operation fulfills the requirement of both backtracking and garbage collection:

```
void trail(int* t){ if (t ≤ GL) *TT-- = t; }
```

It is worth to note that a trail operation has to be done for each variable binding even without the presence of the CGC. However, CGC does cause extra overhead because the trail stack is expanded to hold remembered sets as well. Now, let us assume that Fig. 3 and Fig. 4 together give a snapshot of execution, it is easy to see that any currently trailed root goes to g_3's remembered set (Note g_3 is pointed by GL).

A refinement is to dedicate a special trail operation to backtracking. 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 trail operation is exactly the WAM-like trail operation:

```
void btrail(int* t){ if (t ≤ BB) *TT-- = t; }
```

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

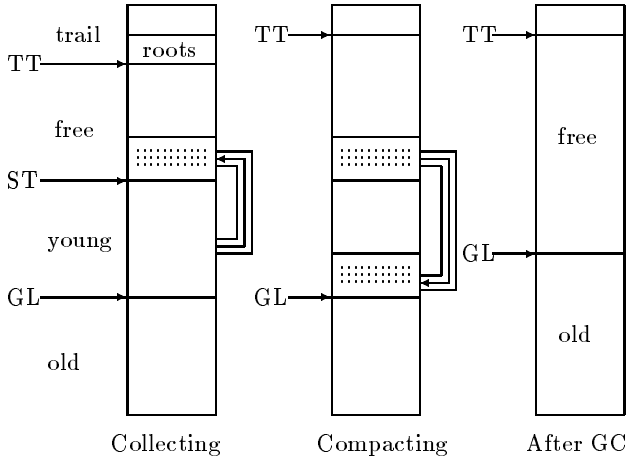

Figure 5: CGC Overview

Now, we have an embryonic form of the CGC algorithm, refer to Fig. 5. At some stage of execution, the collector is invoked. Variable *temp* points to the generation delimiter returned by the *get_gline()* function, and GL holds the corresponding generation line (see the *generic memory allocation algorithm*). Thus, 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 (collecting region), the space between ST to TT is the copy area which will be used to save a temporary copy of useful data, and the space from TT to *temp* stores remembered sets to be collected. 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:

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

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_gline(...)*), then remembered sets saved from TT to t_i give the first portion of g_i's root set. 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.

The collection phase transitively traverses from the initial root set and creates copies of all reachable objects located in the young area. For a given root R, we have three possible cases:

- R is a pointer inside the collecting region. This is

because the collecting region might involve multiple generations, and each of them has an associated remembered set. Some roots remembered during execution might fall into the collecting region (young generations), i.e., they are not useful in old generations. Therefore, we simply drop R.

- R is a root remembered for garbage collection as well as for backtracking. We have to make an extra copy of R in the trail in case of later backtracking. This is done by moving the bottom root to the top of the trail and inserting R into that vacated slot. It looks that R is remembered in the previous old generation and will be collected in this generation. Otherwise,

- R is a root purely for garbage collection. Thus we collect it.

Most copying collectors use Cheney's breadth-first algorithm [7] 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 [5] 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 [10], *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. Furthermore, CGC implements cheap *variable shunting* by simply dropping all intermediate references which reside above GL.

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 [8]). 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 [3] 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.

The above 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. 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. However, some optimizations could be done during compilation to reduce the size of remembered sets or to *early reset* some remembered variables. Unfortunately, space does not allow for further discussion of these issues.

4. 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 [5]. 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/4* 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.

Benchmark	Max-stack	Max-trail	CGC	Garbage	Useful
dna(100)	248K	9	100	16M	1.9K
boyer(1)	700K	99	16	1.4M	163K
nrev(2000)	833K	75	170	13M	762K
qsort(10K)	629K	31	14	1.4M	167K
queens(12)	1K	37	0	0	0
rev(100K)	493K	3	13	713K	352K
serial(10K)	758K	1308	27	2.7M	226K
tak(24,16,8)	671K	8	100	15M	0.4K
tsp(100)	293K	17	3599	357M	584K
browse(10K)	732K	12	151	12M	581K

Table 3: Garbage Collection and Memory Consumption

Table 3 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 backtracking is not included in my statistics. Finally, column *Useful* gives the total amount of useful data being collected.

Now, let us examine the performance. What performance is expected for a good GC algorithm? To answer this question, I first take remarks from [14]: "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 the usr-time and sys-time together, the third row gives the ratio of E/D.

	dna_100	boyer_3	qsort_100K	serial_100K	browse_10K
E	2.15/0.0	9.08/0.1	4.61/0.0	7.21/0.1	69.85/0.0
D	2.51/0.7	9.06/1.8	4.52/1.1	7.18/1.6	79.02/0.7
E/D	67%	85%	97%	83%	88%

Table 4: CGC-enabled *vs.* CGC-disabled

The results in Table 4 are beyond our expectation. For these benchmarks,

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

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 periodicly 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, I found that the performance penalty caused by CGC is trivial.

Perhaps the most interesting result of this research is that the LVM emulator outperforms SICStus (fast-mode) in cases which involve intensive garbage collections. All benchmarks are compiled to compact code (S_c) and fast code (S_f) and tested by SICStus 3.1 on the same platform. The following tables give the execution times gathered from SICStus and the LVM system respectively.

dna							
input	20	100	200	1K	10K	100K	500K
S_c	0.32	1.55	3.08	16.39	199.9	2,331	28,716
S_f	0.11	0.53	1.07	4.05	53.27	635	15,351
LVM	0.41	2.15	4.25	21.26	211.3	2,123	10,599
serial							
input	1K	10K	100K	1M	3M	5M	5.5M
S_c	0.05	0.65	7.37	86.09	262.6	526	1,177
S_f	0.02	0.32	4.02	52.57	167.7	386	969
LVM	0.06	0.65	7.21	96.26	317.7	547	603
qsort							
input	1K	5K	10K	100K	1M	1.5M	2M
S_c	0.02	0.11	0.24	4.96	273.22	1,006	9,703
S_f	0.01	0.04	0.08	2.38	150.87	747	9,197
LVM	0.03	0.15	0.28	4.61	191.48	413	697
reverse							
input	10K	100K	1M	5M	6M	7M	8M
S_c	0.01	0.25	2.58	15.57	24.48	39	149
S_f	0.01	0.17	1.74	11.38	19.35	33	146
LVM	0.02	0.28	2.88	13.71	16.63	19	22
boyer							
input	1	2	3	4	5	6	7
S_c	0.38	2.33	6.98	20.27	61.38	189	Seg. F
S_f	0.19	1.06	3.24	9.47	27.4	84	Seg. F
LVM	0.53	3.11	9.08	26.21	79.68	253	823
browse							
input	100	1K	10K	50K	70K	100K	200K
S_c	0.45	5.16	51.64	406.96	726.15	1,367	4,895
S_f	0.17	1.8	20.87	253.99	503.13	1,064	4,338
LVM	0.67	7.2	72.06	459.38	689.35	1,060	2,908

Table 5: LVM *vs.* SICStus

tak							
input	20	22	24	26	28	30	32
S_c	0.21	0.71	1.96	4.66	10.14	21	39
S_f	0.05	0.16	0.43	1.02	2.2	4.5	8.5
LVM	0.28	0.9	2.42	5.82	12.65	25	49
tsp							
input	30	50	70	100	150	200	255
S_c	0.28	1.99	7.35	30.08	148.69	465	1,208
S_f	0.09	0.59	2.16	8.57	42.29	130	337
LVM	0.48	3.43	12.68	51.05	253.7	785	2,056
queens							
input	14	16	18	20	22	24	26
S_c	0.06	0.36	1.76	9.88	99.29	29	33
S_f	0.01	0.1	0.51	2.84	28.79	8.4	9.7
LVM	0.1	0.65	3.08	17.13	174.47	51	59
nrev							
input	1K	2K	3K	4K	5K	10K	12K
S_c	0.24	0.94	2.75	4.93	6.33	27	49
S_f	0.11	0.38	0.89	1.67	2.68	13	20
LVM	0.58	2.41	5.3	9.62	15.26	65	95

Table 6: LVM *vs.* SICStus

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 (indexed) stack variables to store intermediate results, 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 [12].

As expected, SICStus emulator should be faster than the LVM emulator, especially for benchmarks involving intensive arithmetics and builtin calls. Examining the above tables, SICStus is faster (roughly in the range of 10% to 100%) when the input size is small. When we increase the input size, some benchmarks (Table 6) keep the same performance ratio whereas some (Table 5) greatly narrow the performance gap and at certain breakthrough points they perform better than their counterparts under SICStus, not only the compact code but also the fast code. For *qsort(2M)*, the LVM emulator is 12 times faster than the SICStus fast code.

dna							
input	20	100	200	1K	10K	100K	500K
S_f	3	8	15	0	5	65	1,241
LVM	20	100	200	1,005	10,066	100,679	503,400
serial							
input	1K	10K	100K	1M	3M	5M	5.5M
S_f	0	2	7	13	19	22	64
LVM	1	27	292	3,157	9,500	15,866	17,450
qsort							
input	1K	5K	10K	100K	1M	1.5M	2M
S_f	0	0	0	18	126	499	5,374
LVM	0	6	14	243	11,861	27,927	49,188
reverse							
input	10K	100K	1M	5M	6M	7M	8M
S_f	0	2	3	3	9	5	27
LVM	0	13	145	731	877	1,024	1,170
boyer							
input	1	2	3	4	5	6	7
S_f	1	5	13	24	48	35	Seg. F
LVM	16	98	308	857	2,465	8,141	24,694
browse							
input	100	1K	10K	50K	70K	100K	200K
S_f	0	1	7	13	58	13	21
LVM	0	12	151	768	1,077	1,540	3,082

Table 7: GC Invocations - LVM *vs.* SICStus

How could this happen? The key is garbage collection. Taking two fast code examples, among the 15,351 seconds for executing *dna(500K)* and 9,197 for *qsort(2M)*, 13,316 and 9,057 seconds are spend on garbage collection respectively. That is, the performance penalty caused by SICStus garbage collection reaches up to 87 and 98 percent for these special cases. The major reason is that SICStus adopts a *mark and compact* garbage collector: as the heap residency of a program increases, the performance of the collector degrades; as less free space is recovered, collection become more frequent. This is further evidenced by Table 7 which gives the data on GC invocations. For example, from *dna(100k)* to *dna(500k)*, gc-invocations in SICStus change from 65 to 1,241, which is almost four times of the ratio of two inputs; whereas gc-invocations in the LVM change from 100,670 to 503,400, which shares the same ratio of two inputs. This observation can be found in other benchmarks.

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. Table 8 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.

dna							
input	20	100	200	1K	10K	100K	500K
LVM	10	9	12	8	9	9	10
serial							
input	1K	10K	100K	1M	3M	5M	5.5M
LVM	338	1,308	2,410	7,243	13,330	17,385	17,608
qsort							
input	1K	5K	10K	100K	1M	1.5M	2M
LVM	8	22	31	44	50	53	53
reverse							
input	10K	100K	1M	5M	6M	7M	8M
LVM	2	3	3	3	3	3	3
boyer							
input	1	2	3	4	5	6	7
LVM	99	127	147	147	158	158	158
browse							
input	100	1K	10K	50K	70K	100K	200K
LVM	0	13	12	12	12	12	12

Table 8: Trail Usage for Remembered Sets

Note that the measurements and analysis presented here are not universally applicable, more experiments with a wide range of applications must be tested. In addition, the latest release of SICStus offers better performance on garbage collection, I was unable to compare with this new version for the lack of resource. Nevertheless, the benchmarks and their performance studied in this paper confirm my hypothesis that the merged heap/stack paradigm incorporated with CGC offers a novel, practical technology in the design of high-performance Prolog systems.

5. CONCLUSION

In this paper, I proposed 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 combines the advantages of copying, mark-compact, generational, and incremental algorithms. 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. As a result, the LVM emulator outperforms SICStus (version 3.1) in fast mode for cases involving intensive garbage collections. 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.

6. ACKNOWLEDGMENTS

I would like to express my appreciation to the Natural Science and Engineering Council of Canada for supporting this research. Thanks to anonymous referees for valuable comments and suggestions.

7. REFERENCES

[1] H. Ait-Kaci. *Warren's Abstract Machine: a Tutorial Reconstruction*. MIT Press, 1991.

[2] K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, 1988.

[3] D. Barrett and B. Zorn. Garbage collection using a dynamic threatening boundar. In *Conference on Programming Language Design and Implementation*, pages 301–314. SIGPLAN'95, ACM, 1995.

[4] Y. Bekkers, O. Ridoux, and L. Ungaro. Dynamic memory management for sequential logic programming languages. In *IWMM'92, LNCS 637, Springer*, pages 344–355, 1992.

[5] J. Bevemyr and T. Lindgren. Simple and efficient copying garbage collection for Prolog. In *PLILP'94, LNCS 844, Springer*, 1994.

[6] M. Bruynooghe. Garbage collection in Prolog interpreters. In *Implementation of Prolog*, pages 259–267. Ellis Horwood Ltd., 1984.

[7] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[8] B. Demoen, G. Engels, and P. Tarau. Segment order preserving copying garbage collection for WAM based prolog. In *Proceedings of the 1996 ACM Symposium on Applied Computing*, pages 380–386, 1996.

[9] Y. Ding and X. Li. Cache performance of chronological garbage collection. In *Proceedings of CCECE'98*, pages 1–4, 1998.

[10] R. R. Fenichel and J. C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, 1969.

[11] M. J. R. Goncalves and A. W. Appel. Cache performance of fast-allocating programs. In *Proceedings of the 7th FPCA, ACM Press*, pages 293–305, 1995.

[12] R. C. Haygood. Native code compilation in SICStus Prolog. In *Proceedings of the 11th International Conference on Logic Programming, Vol. 34(1)*, pages 191–204, 1994.

[13] S. L. Huitouze. A new data structure for implementing extensions to Prolog. In *PLILP'90, LNCS 456, Springer*, pages 136–150, 1990.

[14] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[15] A. Marien and B. Demoen. A new scheme for unification in WAM. In *Proceedings of the International Symposium on Logic Programming*, pages 257–271, 1991.

[16] P. Messina, D. Culler, W. Pfeiffer, W. Martin, J. Oden, and G. Smith. The high-performance computing continuum: Architecture. *Communications of ACM*, 41(11):36–44, 1998.

[17] M. B. Reinhold. Cache performance of garbage collected program. *SIGPLAN 94-6, ACM*, pages 206–217, 1994.

[18] D. Sahlin and M. Carlson. Variable shunting for the WAM. *SICS Research Report R91:07*, 1991.

[19] K. Taki. Parallel inference machine PIM. *Fifth Generation Computer Systems*, pages 50–72, 1992.

[20] P. Tarau. Ecological memory management in a continuation passing Prolog engine. In *International Workshop IWMM92, LNCS 637, Springer*, pages 344–355, 1992.

[21] P. Van Roy. 1983-1993: The wonder years of sequential Prolog implementation. *J. Logic Programming*, 19, 20:385–441, 1994.

[22] D. H. D. Warren. An abstract prolog instruction set. *Technical Note 309, SIR International*, 1983.

[23] P. R. Wilson, M. S. Lam, and T. G. Moher. Caching considerations for generational garbage collection. In *Proceedings on Lisp and Functional Programming, ACM*, pages 32–42, 1992.

[24] N. Zhou. Parameter passing and control stack management in Prolog implementation revisited. *ACM Trans. on Programming Languages and Systems*, pages 1–28, Nov. 1996.

[25] B. G. Zorn. The effect of garbage collection on cache performance. In *Tech. Report CU-CS-528-91, Univ. of Colorado at Boulder*, 1991.