The MT Evaluator Virtual Machine: A Virtual Machine for Pure Functional Languages

Steve M. O'Brien, Gregory P. Van De Moere, Joan E. Feeney, Kai H. Fan,
Peter M. Laurina, Kristine Joy S. Apon, Marco T. Morazán
Seton Hall University
Department of Mathematics and Computer Science
400 South Orange Avenue
South Orange, NJ 08544 USA
E-mail: {obrienst,vandemgr,feeneyjo,fankai,laurinpe,aponkris,morazanm}@shu.edu

ABSTRACT

The MT system is being developed as a test bed for different design and implementation choices in the development of a pure functional language with an all-software intelligent distributed virtual memory system. The distributed memory system allows for memory management to occur in parallel with program execution. In order for a language implemented using MT to fully take advantage of the performance benefits of managing memory in parallel it must be compiled instead of interpreted. In this article, we describe work in progress on the MT evaluator virtual machine which has been designed specifically to evaluate compiled code for a pure list-based functional language. The MT evaluator is a switched-based virtual machine that executes bytecode. We describe the major components of the MT system and their implementation in C++. Preliminary empirical evidence on memory performance is presented.

1. INTRODUCTION

Programming languages that provide a very high-level of abstraction, such as functional languages, have failed to become part of the industrial mainstream of programming, because they are considered slow. Sequential functional languages have been rendered slow by poor interaction with virtual memory[4, 7, 13]. Virtual memory interaction is affected by the memory allocation algorithm, the paging algorithm, and the type of garbage collection scheme used. If memory allocation and garbage collection algorithms are not carefully designed to conform with the expected access patterns of the language then locality of reference can be severely reduced causing excessive paging traffic and rendering the language slow. Parallelism has been used to make functional languages faster by executing garbage collection and program evaluation in tandem and by parallelizing user code. As in the case of sequential functional languages, parallel functional languages have not achieved their theoretical potential due to the lentitude associated with copying distributed data structures to access them locally and with dereferencing non-local pointers[5, 8, 9, 10, 11, 12].

The MT architecture is based on an all-software DVM tailored to the needs of a pure functional language [9]. The basic computational unit is not a processor,instead, it is an MT node. Figure 1 displays an abstract view of an MT node. One processor executes the program (called the evaluator node) and the rest of the processors form a runtime intelligent backing store that provides memory management services to the evaluator. The backing store processors are responsible for intra-node virtual memory and communication management.

The current MT architecture divides the DVM system into four management networks based on the major data structures needed to evaluate a functional program: the MT heap, the MT stack, the MT function space, and the MT garbage collector. The current version of MT is implemented without a garbage collector and the evaluator runs a string based interpreter. The lack of a garbage collector has not yet become a major concern, because the benchmarks used to profile the system have not consumed all of the available virtual address space. More importantly, there is empirical evidence that strongly suggests the MT allocation algorithm compactly stores list-based structures rendering the paging performance of *first-in first-out* (FIFO) and the *least recently used* (LRU) page replacement algorithms the same for MT heap pages [9, 8, 10, 11]. For MT stack pages, LRU is superior to FIFO and a paging algorithm that implements LRU without its costly overhead per access has been developed for the MT stack and is described in [12].

The MT initiative is now focusing on replacing the string based interpreter at the processor running the MT evaluator with a virtual machine that evaluates compiled byte code of functional programs. Interpreters are useful tools for interactive program development, but for a system designed for performance it is necessary to compile programs. Unlike interpreted code, compiled code is efficient and fast because it does not have to perform any syntactic analysis at runtime and eliminates the need to unnecessarily save values on the stack. In the remaining sections of this article,

Proceedings of MASPLAS'02
Mid-Atlantic Student Workshop on Programming Languages and Systems
Pace University, April 19, 2002

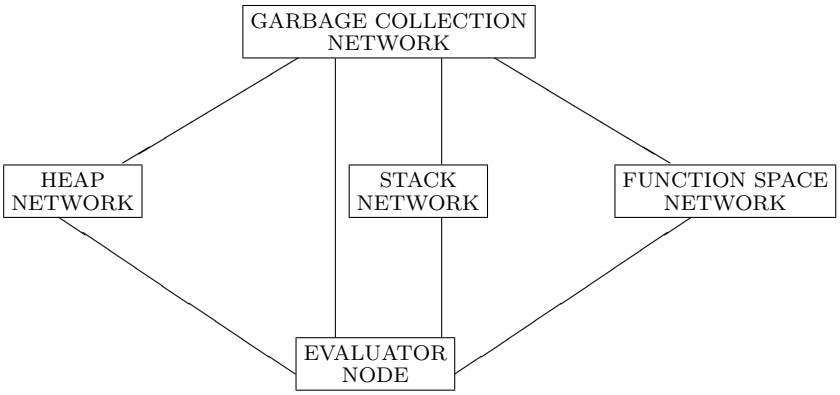

Figure 1: Abstract view of the architecture of an MT Node

we describe the implementation of the MT evaluator virtual machine and the different MT components in C++. We present some preliminary empirical numbers that we have collected and present ideas for future work in our concluding section.

2. THE MT EVALUATOR VIRTUAL MACHINE

The principal class in the C++ implementation of the MT Evaluator virtual machine is the MTmachine class. An instance of this class builds the appropriate data structures needed in a virtual machine. The MT evaluator virtual machine includes a heap object, a stack object, a function space object, and a symbol table object. The classes that implement these objects are described in the following sections.

In addition, the MTmachine class has a set of registers and implements a fetch-execute cycle. This set of registers, the heap object , the stack object, the function space object, and the symbol table object are all manipulated by the set of MT primitives that make up the instruction set.

2.1 Registers

The registers store the state of the evaluator and control the flow of the program. The registers are:

- **pc**: The 'program counter' register contains the virtual address of the next instruction to be executed.
- **cont**: The continue register contains the virtual address of the instruction to which control is transferred to upon execution of a branch or goto primitive.
- **env**: The 'environment register' holds the virtual stack address of the activation record for the current function being evaluated.
- **newenv**: The 'new environment register' holds the virtual stack address of the new activation record that is being built to apply a user defined function to its arguments.
- **val**: The 'value register' stores the result of applying any primitive or user-defined function.
- **flag**: The 'flag register' holds a boolean value that is accessed during the execution of branch instructions.

2.2 MT Primitive Instructions

There are 68 MT primitives some of which are specific for functional languages and would not be needed by imperative languages like C/C++ and Java. We can divide the set of primitives into primitive value functions used to compute a value and primitive control functions used to manipulate registers and the state of the machine. The set of MT primitive functions includes the arithmetic operators, the relational operators, the boolean operators, list operators (cons, car, cdr), tag operators (number?, symbol?, eq? atom? null?, pair?), and a random number generator primitive. These primitives expect their input to be stored on the top of the stack. The primitives pop their arguments off the stack and store the result in the val register. The cons primitive differs from the other primitives, because it allocates its arguments in the heap before returning the resulting list S-expression in the value register.

The primitive control functions include the stack operators (push, pop, and stackAccess), the save and restore operations for each register, the test-and-branch primitives, the S-expression creation primitives, a read primitive, a print S-expression primitive, and a goto primitive. The branch and goto primitives change the value of the pc register to transfer program control to a different part of instruction sequence. The stack operators push and pop suffice to implement all the primitive value functions. The stackAccess primitive is needed to implement lexical addressing to access arguments in an activation record created for a user-defined function.

Each MT primitive is implemented as a C++ function that returns a boolean value indicating if the instruction was suc-

```
begin loop
  fetch next instruction.
  execute instruction.
  if not(terminate?) branch begin-loop.
end loop
```

Figure 2: Pseudo-code for the Fetch-Execute Cycle

cessfully executed. This set of primitives suffices to implement any functional language. We expect, however, to add primitives to make lambda operations and the manipulation of first-class functions more efficient.

2.3 Code Execution

The basic algorithm of evaluation implements a fetch-execute loop. At each step an instruction is fetched and executed. The execution of an instruction changes the state of the machine by changing the value of a register and/or the heap, the stack, and symbol table. Each primitive instruction updates the pc after it has been executed. The cycle continues until a primitive instruction fails to execute properly or the pc reaches the end of the instruction sequence. The pseudocode for this loop is displayed in Figure 2.

When the fetch-execute cycle terminates without an error the value computed is the only element on the stack. This element is then popped of the stack and printed to the screen. If at any point, during the computation, a primitive instruction is unable to execute the loop is terminated and an error message is displayed.

3. THE MT HEAP

The heap is used by the MT evaluator exclusively to store lists. Heap memory is only allocated by executing the cons primitive, which takes in two arguments, commonly referred to in the Lisp world as the car and cdr. The result of a cons operation is tagged as a list.

The virtual heap space is implemented as an array of pages constituting the virtual address space and an array of frames that hold the pages currently accessible. The number of heap pages is much larger than the number of heap frames, which is typical of a modern hardware architecture. A demand-paging algorithm is implemented to swap data between the pages and the frames creating the illusion of a larger memory space than the one the machine actually has.

In the current version of the MT evaluator virtual machine, the virtual address space is completely held at the evaluator node. This is done for debugging purposes. Eventually the heaps virtual address space will be distributed over several computers in a Beowulf-type architecture [14]. The distributed architecture of the MT System allows paging management to be parallelized by shifting management routines to its intelligent backing store.

The stack and code are kept as separate objects. No stack or code data is stored in the heap, and vice versa, allowing for separate paging policies for the different segments of memory.

Each heap word can hold an S-expression. Heap memory is allocated sequentially starting at 0 and is accessed using virtual addresses. The page number and offset of an S-expression is calculated from the virtual address by using modular arithmetic and integer division. For example, if a heap page size can hold 256 S-expressions and a data item is stored in virtual address 317, we have that the data item is on page 1 (317 DIV 256) with an offset of 61 (317 MOD 256). After computing the page number and offset the page table is checked to see if the needed page is currently held in a frame. If so, the memory can be accessed directly from the frame. If not, the fault manager is called to swapped in the needed page.

3.1 Heap Implementation in C++

The heapmem class contains the following instance variables used for paging management and to track performance. These are:

- **oldestframe**: Holds the frame number containing the page that has been held in the frame array for the longest period of time. This variable is used by the FIFO paging policy.
- **heapfaults**: This variable is used to record the number of heap faults.
- **next**: Holds the next virtual heap address to be allocated.
- **heapaccesses**: A variable used to record the number of times the heap is accessed (any read or write operation).
- **heapallocations**: A variable used to track the total number of heap objects allocated.
- **TIMESTAMPS[NUMOFHEAPFRAMES]**: This an array which is managed like a binary min-heap (heapified every time a different page is accessed) to implement LRU.
- **frametable[NUMOFHEAPFRAMES]**: An array that stores the page numbers held in the frames.
- **pagetable[NUMOFHEAPPAGES]**: An array containing a PageTableEntry for each heap page. The PageTableEntry structure is defined below.

The heapmem class provides two public methods, getSexp(int vaddress) and allocSexp(char tag, int v1, int v2), to access and allocate heap data. The getSexp method returns the S-expression at a given virtual address. If this virtual address refers to a location not contained in the heap frames, the fault manager is invoked prior to returning the requesting data. The allocSexp method stores a new S-expression in the heap created from its input.

The private methods of the heapmem class manage the virtual address space and hide the details of implementation

```
struct Sexpdef     struct PageTableEntry
{                  {
char tag;            int framenum;
int car,cdr;         bool valid,dirty;
}                  }
```

Figure 3: S-expression and Page Table structures

from the virtual machine class. The most important private method is the fault manager which has been written to implement LRU and FIFO. Previous studies on MT using a string-based interpreter for program execution concluded that the paging performance of FIFO and LRU for MT heap pages is virtually the same [8, 10, 12]. The overhead associated with LRU has made FIFO the preferred paging policy.

In order to reduce the overhead associated with LRU, we are exploring an implementation that keeps pages ordered using a binary tree implementing a min-heap. The top of the binary min-heap always holds the least recently used page. A trace is kept of the last n different pages that are accessed is used to rebuild the heap to always find the least recently used page quickly. This algorithm may be effective if the working-set of pages changes slowly (i.e. exhibits a great deal of locality of reference) and the heapifying process can be performed by the heap management network.

3.2 S-expressions and Page Table Entries

Figure 3 displays the implementation of an S-expression and a page table entry. An S-expression has a char that is used as a tag. The tag of an S-expression identifies the type of data it holds. The current tags supported for user prorgams are integers, reals, list, function, and nil. Other tags defined are only used internally by the MT machine for printing S-expressions and pushing the values of registers onto the stack.

Every $heappage_i$ has a page table entry at position i. Page table entries have three fields. The valid field indicates if the page is held in the heap frames. If so, the frame number indicates which frame it is stored in. The dirty field indicates if a page needs to be swapped out when it is elected for eviction from the heap frames.

4. THE MT STACK

The MT stack is used for parameter passing and function calling. In functional languages, efficient implementation of stack memory can profoundly affect performance, because function calling is prevalent and more common than in imperative languages. The MT stack holds S-expression structures instead of pointers to S-expressions as is common in many Lisp/Scheme implementations [2, 3]. This strategy increases locality of reference and list compaction [9]. The eq? primitive is slightly more complicated, but the gains in virtual memory performance justify this choice.

4.1 Implementation

The MT stack is an object defined by a class with 8 public methods. They are push(), pop(), getStackEntry(), getStackTop(), getStackFaults(), getStackAccesses(), getNumOfPushes(), getMaxStackTop(). The first four are used for stack access and the latter are used to measure performance. There are also 20 private methods used to manage the virtual address space.

The stack virtual address space is divided into pages of which only a small subset can be held at the processor running the evaluator at any given time. To compute the value of a function, *f*, the arguments to *f* and any necessary flow control information (e.g. return address) is pushed onto the stack and then *f* is applied. A demand paging algorithm is implemented to swap pages between the frames of the MT evaluator and MT's backing store for stack pages when a fault occurs. We have implemented both FIFO and the MT stack replacement algorithm [11] to verify that previous paging results using MT are still valid for the new system with MT evaluator virtual machine.

The MT stack frames are implemented as an array of pages where each page is an array of S-expressions. The stack grows towards higher addresses as S-expressions for either data or flow control are pushed. Since there are no local function definitions in MT [1], activation records are accessed in a strict LIFO manner making LRU an optimal page replacement algorithm [11]. The MT stack page replacement algorithm was developed based on the proof that LRU is optimal. In essence, it always swaps out the least recently used page without having to time stamp pages. The reader is referred to [11] for details.

5. THE MT FUNCTION SPACE

In the MT System, code space consists of six main objects: code pages, code frames, a code page table, a code frame table, a label table, and a function table. Each has a specific purpose in managing the code during the execution of the program. The code pages represent the entire code virtual address space while the code frames contain a subset of code pages. The code space and code frame classes have methods used to manipulate code memory. The code frame table and code page table each maintain needed information in order to perform virtual memory management.

The CODE class has only two public functions: one that performs the insertion of an instruction and one that retrieves an instruction. When the MT machine tells the code space to insert an instruction it is placed sequentially in the next available virtual address. Retrieval of an instruction consists of obtaining the location of a virtual address from within the code space and returning the instruction to the MT machine. Having only those two functions available publicly allows the details of how code space is implemented to be hidden from the MTmachine class.

The label table maintains the labels of the code generated by the compiler. A label marks a location in the code for use with branch statements and goto's. The label table maintains a hashed list of label names and the associated index into the code. The index into the code is the first instruction that the program will run when told to jump to that label. This allows the use of loops and recursion in a program.

A function table stores the names of user-defined functions.

[1] Local functions definitions can be removed through a process called lambda-lifting [6]

It is similar in structure and design of the label table. It contains a virtual address into the code space where the function begins and an integer value of the number of parameters the function needs. The function names are hashed into an array for fast access to the information associated with it.

Three paging algorithms have been implemented to work in code space: FIFO, LRU, and a double linked circular list version of LRU. FIFO is implemented as the standard first in, first out algorithm and LRU is the a standard least recently used algorithm. The third algorithm uses LRU in determining which page will be swapped out when a page fault occurs, but it stores the ages of the frames differently than the typical LRU. The ages are stored in a separate data structure keeping all the other information on the frame table. The ages are stored in a double linked list structure where the order the elements are in represents the age of the frames. Each element of the linked list contains an index corresponding to the frame it represents. The three algorithms are being compared and contrasted to each other. As programs are developed using the MT machine, test data is being gathered to determine which algorithm is the most efficient.

6. THE MT SYMBOL TABLE

The MT-evaluator uses a hashed symbol table to store non-boolean non-numeric primitive type data. Symbols are used extensively by functional programmers to do symbolic computing. Symbols in MT are also used to store function names and labels into the instruction sequence. As discussed in the previous section, function and label symbols are used for flow control and these are kept in a separate table from symbols used by a programmer.

The symbols in the MT evaluator virtual machine are represented by a data structure containing a key, a string, the length of the string, and the status of the symbol. The key is an integer that is used to identify the symbol and determines where the symbol is hashed. The string represents the printable version of the symbol. The length stores the length of the string representation of the symbol. Finally, status indicates if a symbol entry is in use.

The symbolTable class contains an array of symbols and methods to manage and access the symbol table. Every symbol is hashed into the array by generating a numerical identifier (the key) based on the string representation of the symbol. The key is passed as input to a hashing function that returns the address in which the symbol is to be stored. If there is a collision, the hashing function is re-applied until a free symbol table entry is found.

The symbolTable class tracks the number of unsuccessful accesses to the symbol table during program execution. These may arise when two or more symbols are hashed to the same location. This number will be a good indicator of hashing performance, because the amount of time needed by the hashing function is constant. We have implemented two hashing functions: linear hashing and double hashing. To test out the efficiency of each function, random symbols were generated for different sizes of the symbol table. Preliminary data is presented in the next section comparing the number of collisions generated by linear hashing and double hashing.

```
(define (mklist n)
(if (= n 0)
'()
(cons (random n) (mklist (- n 1)))))
```

Figure 4: Code to create a list of n random numbers.

7. EMPIRICAL RESULTS

In this section, we present preliminary empirical data collected using the MT evaluator virtual machine. We present one example of data collected for the heap and stack using a hand-compiled function to create a list of 200 random numbers. We also present data comparing the performance of our linear and double hashing functions for the symbol table.

7.1 Creating a List of Random Numbers

We hand-compiled the Scheme function displayed in Figure 4 and used it to create a list of 10000 random integers. The heap and stack pages each stored 512 S-expressions. The number of heap and stack frames were both set to 10.

The compiled code (printed with labels and comments) is displayed in Figure 5.

Table 1 presents data on the heap and stack performance under FIFO as the page replacement algorithm for each. Heap and stack pages do not share a common pool of frames. That is, a stack page could not displace a heap page and vice versa. The fault rate is defined as $\frac{faults}{accesses}$. As expected, FIFO is a competitive page replacement algorithm and the number of stack accesses is much higher than the number of heap accesses. The reason for this is that the MT stack holds actual values instead of pointer to values. In a system where the stack holds pointers to data we would expect heap accesses to be higher for mklist.

The maximum height the stack reaches is 30005. Comparing the number of allocations with the maximum stack height suggests that the stack should not be heap allocated as is done in some functional languages (e.g. SMLNJ [1]). The stack memory can be recycled without having to call a garbage collector. Furthermore, if the stack were heap-allocated the paging performance of the heap would degrade by having garbage stack allocations intertwined with live-data.

The data in this example demonstrates the type of experiments we can run. The system will be used to design the distributed virtual memory system of the heap, stack, and code space.

7.2 Symbol Hashing Performance

Table 2 displays performance data for linear and double hashing of randomly generated symbols into the symbol table. We measured the number of collisions generated by inserting 100-900 (in increments of 100) symbols into tables of different sizes. Table 2 displays the average over all experiments. This preliminary data suggests that linear hashing causes up to 28% more collisions than double hashing. The

Structure	Accesses	Faults	Fault Rate	Allocations
Heap	40000	60	0.0015	20000
Stack	270013	99	0.0004	120006

Table 1: Heap and stack data for (mklist 10000)

Inserted Elements	Linear Hashing	Double Hashing
100	3	3
200	11	11
300	24	21
400	43	40
500	67	65
600	104	94
700	140	125
800	199	181
900	330	238

Table 2: Collisions for Linear Hashing vs. Double Hashing

performance gap between linear hashing and double hashing grows larger as the number of symbols inserted into the symbol table increases. The number of collisions under double hashing, however, grows faster than the number of collisions under linear hashing as the table becomes fuller. This may suggest that as the number of symbols inserted increases, linear hashing may perform better than double hashing. At this moment, however, we have not gathered enough empirical evidence to reach a conclusion.

8. FUTURE WORK

The MT evaluator virtual machine has been designed to replace a string-based interpreter used in previous implementations of the MT system. The evaluator is a switch-based virtual machine that executes instructions that are byte encoded. The virtual machine is intended to execute code produced by a compiler for any functional language that uses applicative-order evaluation. That is, all arguments to a function are evaluated before the function is applied. This evaluation contrasts with functional languages, such as Haskell, that use lazy evaluation.

The MT evaluator virtual machine will be used to continue fine tuning the paging performance of the MT heap and the MT stack. In addition, it will serve as the first MT platform in which the paging properties of code space can be studied. This work will help develop a model that explains how functional languages access memory.

9. REFERENCES

[1] Andrew W. Appel and Zhong Shao. An Empirical and Analytical Study of Stack vs. Heap Cost for Languages with Closures. Department of Computer Science,CS-TR-450-94,Princeton University, 1994.

[2] D. Bartley and J.C. Jensen. The Implementation of PC Scheme. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 86–93, Cambridge, Massachusetts, 1986.

[3] S.E. Fahlman and D.B. McDonald. Design Considerations for CMU Common Lisp. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 137–156, Cambridge, MA, USA, 1991. MIT Press.

[4] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12:611–612, 1969.

[5] B. Goldberg. *Multiprocessor Execution of Functional Languages*. PhD thesis, Department of Computer Science, Yale University, New Haven, Conneticut, 1988.

[6] T. Johnsson. Lambda lifting: transforming programs to recursive equations. *Proceedings of the IFIP Conference on Functional Programming and Computer Architecture, ed. Jouannaud*, LNCS 201, 1995.

[7] David A. Moon. Garbage Collection in a Large Lisp System. *Proc. of the 1984 ACM Symp. on Lisp and Functional Programming*, pages 235–246, 1984.

[8] Marco T. Morazán and Douglas R. Troeger. A Case Study of Heap Paging in the MT System. In Markus Mohnen and Pieter Koopman, editors, *Proc. of the* 12^{th} *Workshop on Implementation of Functional Languages*, pages 201–214, Aachen, Germany, 2000. Aachener Informatik-Berichte.

[9] Marco T. Morazán and Douglas R. Troeger. The MT Architecture and Allocation Algorithm. In Phil Trinder, Greg Michaelson, and Hans-Wolfgang Loidl, editors, *Trends in Functional Programming*, volume 1, pages 97–104, Bristol, UK, 2000. Intellect.

[10] Marco T. Morazán and Douglas R. Troeger. A Case Study of List-Memory Paging in a Distributed Memory System for Functional Languages. In Martin A. Musicante and E. Hermann Haeusler, editors, *Proc. of The* 5^{th} *Brazilian Symposium on Programming Languages*, pages 80–95, Curitiba, Brasil, 2001. Universidade Federal do Paraná.

[11] Marco T. Morazán, Douglas R. Troeger, and Myles Nash. Designing an All-Software Based Distributed Virtual Memory: The Paging Performance of The MT

```
start   ; starting label
    SETCONT "DONE" ;cont = DONE label
    SETENV 0 ;env = address of activation record
    SENV ;save env on the stack
    SCONT ;save cont on the stack
    MKINTSEXP 200 ;push 200 (initial value of n)
    GOTO "mklist" ; pc = mklist
mklist ; label mklist
    SACC 2 ;get n from stack
    SVAL ;save val onto the stack
    MKINTSEXP 0 ;push 0 onto the stack
    BRE "mklistbase" ;if n == 0 branch to mklistbase
    STOP2NENV ;newenv = address of new activation record
    SENV ;save the env register
    SETCONT "mklistaccum" ;cont = label mklistaccum
    SCONT ;save the cont register
    SACC 2 ;access the value of n
    SAVEVAL ;save n on the stack for MINUS
    MKINTSEXP 1 ;push 1 on the stack
    MINUS ;subtract 1 from n
    SAVEVAL ;save n-1
    NENV2ENV ;env = address of new activation record
    GOTO "mklist" ;loop to mklist
mklistbase ;label mklistbase
    POP ;pop the top element on the stack
    RCONT ;restore cont register
    RENV ;restore env register
    MKNILSEXP ;push nil on the stack
    CONT2PC ; goto instr pointed to by cont register
mklistaccum ;label mklistaccum
    RANDOM ;generate a random number
    SAVEVAL ;push random number on the stack
    SACC 3 ;get the cdr of the list
    SAVEVAL ;push the cdr on the stack for cons
    CONS ;create new list
    POP ;pop the cdr
    POP ;pop n
    RCONT ;restore cont register
    RENV ;restore env register
    SAVEVAL ;save the new list
    CONT2PC ;goto instr pointed to by cont register
    donelabel ;label donelabel
```

Figure 5: Compiled code for mklist

Stack. In Silvia Teresita Acuna and Cecilia Maria Laserre, editors, *Proc. of The* 1^{st} *Iberoamerican Conference on Software Engineering and Knowledge Engineering*, pages 109–120, Buenos Aires, Argentina, 2001. Universidad Nacional de Jujuy (Editorial UNJU).

[12] Marco T. Morazán, Douglas R. Troeger, and Myles Nash. Paging in a Distributed Virtual Memory. In Kevin Hammond and Sharon Curtis, editors, *Trends in Functional Programming*, volume 3, pages 75–86, Bristol, UK, 2002. Intellect.

[13] Robert A. Shaw. *Empirical Analysis of a Lisp System*. PhD thesis, Department of Computer Science, Computer Systems Laboratory, Stanford University, Stanford, California, 1988.

[14] Thomas L. Sterling, John Salmon, Donald J. Becker, and Daniel F. Savarese. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. Scientific and Engineering Computation. MIT Press, 1999.