Paging Performance in the MT Evaluator Virtual Machine

Victor Encarnacion, Patrick DeSomma and Marco T. Morazán
Seton Hall University
E-mail: {encarnvi,desommpa,morazanm}@shu.edu

ABSTRACT

The performance of functional languages is closely related to the manner in which they utilize memory and it is commonly believed that functional languages are slow due to their poor interaction with memory. To make functional languages faster research efforts have focused on parallelism, on memory allocation and performance, and on compiler technology. Each of these lines of research has advanced considerably, but have rarely been combined into one comprehensive approach. The MT system is being developed as a test bed for novel implementation techniques to improve memory performance by utilizing parallelism and modern compiler technology to manage a distributed memory system to support a program evaluator. In this paper, we first briefly describe the MT evaluator virtual machine and its new implementation in C++. We then present performance measurements of the paging behavior of MT's heap, stack, and code space using three benchmarks that represent typical programs written using a functional language. The results confirm previous findings that suggest FIFO ought to be used over LRU as the page replacement policy for MT's heap while LRU ought to be used for MT's stack. We also present the first empirical measurements taken on the paging behavior of MT's code space. These measurements suggest that LRU performs better than FIFO, but that FIFO is a competitive page replacement policy for MT's code space.

1. INTRODUCTION

The performance of functional languages is closely related to the manner in which they utilize memory. It is commonly believed that functional languages are slow, because of their poor interaction with memory [13, 22]. To make functional languages faster research efforts have focused on parallelism [12, 21], on memory allocation and performance [5, 7, 13, 28], and on compiler technology [4, 10, 11, 25]. Each of these lines of research has advanced considerably leading to empirical studies that challenge the perception that functional languages are slow. For example, the memory consumption rate for Lisp is comparable to that of Java [8]. The functional advantage, however, is that development time is much lower and software reliability is higher.

Despite the importance of each of the above lines of research, rarely do we see a comprehensive approach to the implementation of functional languages that considers the impact of design decisions in one category on the other categories. The MT system is being developed as a test bed for novel implementation techniques to improve memory performance by utilizing parallelism and modern compiler technology to manage a distributed memory system to support a program evaluator. The first implementation of MT in Occam on a network of transputers was a string-based interpreter that revealed that FIFO should be used as the page replacement policy for the MT heap [16, 18] and that LRU is optimal for the MT stack. Based on the proof of the optimality of LRU for the MT stack, the MT stack replacement algorithm (MTSRA) was developed. MTSRA is also optimal, but does not incur the per access overhead associated with LRU [17, 19]. In essence, these studies suggest that different paging policies ought to be used for the different memory spaces of a functional language.

The recent development of parallel machines using off-the-shelf technology (e.g. Beowulf machines [24]) and of libraries to support parallel programming (e.g. MPI [26]) has enabled the possibility to port MT to these systems. In this article, we briefly describe the implementation of the new MT system in C++. This implementation is based on a virtual machine that evaluates compiled code. We present empirical measurements taken using three benchmark programs that validate that FIFO is as competitive as LRU as a paging policy for the MT heap and that LRU is optimal for the MT stack. The same benchmarks were used to study for the first time the paging behavior of MT's code space, and these preliminary results suggest that LRU is superior to FIFO. The article ends with some conclusions and directions for future research.

2. THE MT SYSTEM

The MT system divides memory into 5 distinct spaces that are managed independently and are implemented as 5 different address spaces in a distributed virtual memory (DVM) system implemented in software and tailored to the needs of a functional language. These memory spaces are: the MT heap, the MT stack, the MT code space, the MT evaluator virtual machine space, and the garbage collector space. MT's heap, stack, and code space store units of data called

Proceedings of MASPLAS'04
Mid-Atlantic Student Workshop on Programming Languages and Systems
Seton Hall University, April 3, 2004

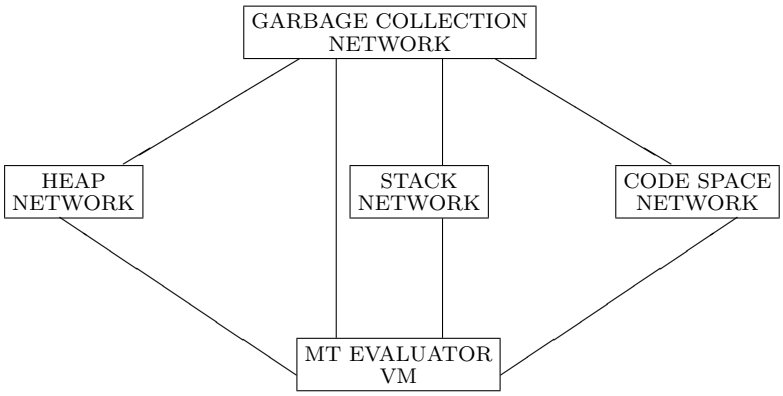

Figure 1: Abstract view of the architecture of an MT Node

MT S-expressions. Each MT S-expression has a tag and a value. Tags distinguish the type of value that is held by an MT S-expression. All values are represented by an integer except lists which are represented by two integers. For example, an INT tag means that the value is an integer, a SYMB tag means that the value is an index into the symbol table, a FUNCT tag means that the value is an index into the function table, an OP tag means that the value is a primitive function (e.g. +, *, and cons) and a BOOL tag means that the value is a boolean. The only compound type is a list that is represented with a LIST tag and two addresses into the heap for the *car* and the *cdr* of the list.

The symbol table, T, the label table, L, and the function table, F, are content addressable memory. There is a hashing function, $hash(str, table)$, that takes as input a string representing a symbol, a label, or a function name, and a table and that adds *str* to *table* if necessary and returns the location in the table where the string is stored. T stores the printable versions of symbols. F stores *known functions* including primitives. $F[i]$, where $i = hash(f, F)$ for some function f, stores a function record for f containing its name, $F[i].name$, its address in code space, $F[i].addr$, its number of parameters, $F[i].params$, and its length, $F[i].len$, in terms of the number of primitive instructions in its compiled code. $L[i]$, where $i = hash(l, L)$, for some compiler generated label named l, stores a record containing the string representing l, $L[i].name$, and the address in code space for l, $L[i].addr$.

2.1 The MT Heap

The MT heap is only used to store dynamically allocated recursive data types built from lists. As program evaluation advances, list data is MT-heap allocated using the MT allocation algorithm described below. Unlike some functional programming systems, the MT stack is not heap allocated and is not maintained as a list. The decision not to heap allocate the stack was made based on the observations that MT-stack memory space can be recycled without the interference of a garbage collector and that not heap-allocating the stack promotes intra-list locality by reducing the amount of garbage (e.g. discarded stack memory) interwoven with live data in the heap [14].

The virtual heap space is divided into pages. A subset of these pages can be held in a set of frames available at the evaluator at any given time. A demand paging algorithm is implemented to service requests by the evaluator for heap pages that are not contained in its frames. Heap pages can be held at the processor running the evaluator or in the heap management network[1]. Each heap page can hold 512 S-expressions. Each S-expression is represented by using a byte as a tag and 64 bits for data. Thus, each page has 72-bit wide words making them roughly $4K$ which is a common page size in hardware-based virtual memory systems [9].

The MT allocation algorithm eliminates the use of pointers except where they are absolutely necessary (e.g. for recursive data types like lists and trees). For example, the creation of a new list when executing **cons** requires pointers to the two MT-heap allocated parts of the list being created. These two parts, however, are first allocated in the heap as literals leading to only cons-cells to contain pointers. Heap space is allocated linearly from the first to the last available address. The MT allocation algorithm only allocates heap space to execute **cons**. The arguments to **cons** are not heap allocated until both arguments have been evaluated.

Previous empirical studies have suggested that the MT allocation algorithm fosters locality of reference [14] and that $FIFO$ is as competitive as LRU as a page replacement policy [16, 18]. This property of being able to perform paging using $FIFO$ is very desirable, because $FIFO$ is easily implemented in software and does not incur a per access overhead like LRU.

2.2 The MT Stack

[1]When enabled heap pages can also be held in the garbage collection network.

The MT stack is used for parameter passing and flow control. The arguments passed to a function application are stored on the stack forming the function's activation record. Along with the arguments to a function, each activation record also contains the information needed to continue the evaluation of any previously interrupted function application. After a function has been applied its activation record is popped off the stack, the result is pushed onto the stack, and control is returned to the calling function.

As the MT heap, the MT stack's virtual address space is divided into pages that can hold 512 S-expressions. In a previous study, empirical evidence established that LRU is superior to FIFO as a page replacement policy for MT stack pages [19]. In fact, it was mathematically proven that LRU is optimal and based on that proof the MT stack page replacement algorithm (MTSRA) was developed and implemented. MTSRA is also optimal, but does not incur the overhead associated with LRU.

2.3 The MT Code Space

The MT code space is used to store sequences of MT instructions represented as S-expressions. Each S-expression has a unique tag associated with one of MT's primitives. Instructions in MT's code space are allocated linearly from beginning to end. When compiled code is loaded into the code space, functions, symbols, and compiler-generated labels are inserted into the function table, symbol table, and label table.

The first empirical measurements of the paging performance of MT's code space are presented later in this article. The first implementation of MT did not have code space memory, because it was solely based on a string-based interpreter.

3. THE NEW IMPLEMENTATION OF MT

The current version of the MT system is not solely based on a string-based interpreter. Instead, the program evaluator is a virtual machine called The MT Evaluator Virtual Machine (MTEVM). The MTEVM is implemented in C++ and is a register-based machine that has access to the memory spaces and tables of the MT system. These memory spaces and a set of registers define the state of the MTEVM. A set of computation, flow control, and register-machine primitives are defined that operate on the MT spaces and the set of registers. The MTEVM implements a fetch-execute cycle that loops through the instruction stream stored in code space until current computation ends. At each iteration of the loop, a primitive instruction is executed that changes the state of the machine. This set-up is similar to other approaches taken for virtual machines for functional languages [1, 4, 10, 20] some of which store instruction sequences as lists. Nonetheless, the approaches are equivalent in that they define a mechanism to store the necessary information to execute the next instruction without necessarily defining this mechanism as a fetch-execute cycle.

In addition to the MT memory spaces and tables, there are six registers that define the state of the MTEVM. The registers are described as follows:

- **pc**: The *program counter register* is used to store the address of the next instruction stored in code space to be executed.
- **val**: The *value register* is used to temporarily store the result of applying any primitive or user-defined function.
- **cont**: The *continue register* is used to store the address of the next instruction to execute upon returning from a call to a non-primitive function.
- **env**: The *environment register* is used to store the virtual stack address of the activation record for the non-primitive function currently being evaluated.
- **newEnv**: The *new environment register* is used to store the virtual stack address of an activation record that is under construction to call a non-primitive function.
- **temp**: The *temporary register* is used to hold instructions retrieved from code space, and for temporary storage by some MT primitives.

Figure 1 displays an abstract view of the MT system. When this article was written, the heap, the stack, and the code space had not been distributed. Instead, the MT system was implemented on a single processor machine and the distributed virtual memory system was simulated. The results presented in this article, therefore, were obtained from this simulation of a distributed virtual memory. The use of a simulator does not invalidate our results, because the virtual memory performance will be exactly the same when the MT spaces are distributed. Instead of paging occurring within a simulator, it will occur across a network of processors. This does not affect the number of accesses, the number of page faults, or the page fault rate of a program being evaluated.

4. PAGING ALGORITHMS

Virtual memory makes it possible for a program to access an address space that is larger than the amount of physical memory available. This is done by combining the relatively fast and small main memory with a slow and large secondary storage such as a hard disk. Virtual memory allows for the execution of processes that may not reside completely in main memory. Processes, therefore, have a logical address space that is mapped to a physical address space.

Physical memory is divided into units of fixed size called frames, while logical memory is divided into units of the same size called pages. Whenever a process is executed, a subset of its pages are loaded from logical memory into any free frames in physical memory. A page fault occurs whenever a process requires a page that is not currently in physical memory. If all the frames are occupied, a victim-page is chosen to make room for the page that caused the fault. The victim-page is written to backing store if it has been modified and the needed page is read into the frame of the victim-page. A page replacement algorithm chooses the victim-page to be replaced. Two common page replacement algorithms are First-In-First-Out (FIFO) and Least-Recently-Used (LRU).

Benchmark	Heap	Stack	Code
MM	868,054	7,368,798	3,774,149
PERMS	203,589	1,534,318	681,099
NQ	1,527,164	35,275,373	18,869,273

Table 1: The number of heap, stack, and code space accesses for MM, PERMS, and NQ.

The FIFO algorithm chooses as the victim the page that has been resident in memory for the longest time. This is most often implemented as a queue. When a page is loaded into memory it is inserted at the tail of the queue. The page at the head of the queue is the one chosen as the victim to be replaced. FIFO works on the assumption that the oldest page in memory is the least likely to be used next. The FIFO algorithm's biggest advantage is speed. At all times, the location of the next page to be replaced is known; whenever a page fault occurs, the frame with the oldest page is overwritten without having to search for it. A potential downside of FIFO, however, is that it may swap out pages that are being accessed at the time of a fault.

The LRU algorithm chooses as the victim-page the page that has not been accessed for the longest amount of time. This is often done by maintaining a clock and by time-stamping a frame with the value of the clock each time a page is accessed. When a page needs to be replaced, the algorithm chooses the one with the smallest time-stamp. LRU works on the assumption that the least-recently-used page is the one least likely to be accessed next. Since the page access sequence is usually not known in advance, when a page fault occurs it is not known which page will not be accessed for the longest time. LRU attempts to make a reasonable approximation based on the page access patterns of the recent past. This algorithm is very costly, given that a clock has to be incremented with every access and the set of frames have to be searched in order to find the one with the earliest time-stamp. Therefore, LRU is usually not implemented. Instead, LRU is approximated in practice [23]. These approximation algorithms, however, do not perform as well as LRU.

The performance of page replacement policies is usually measured by computing page fault rates. It is generally believed among computer scientists that LRU is superior to FIFO, but there exists no proof of this and the empirical data supporting this point of view is not fully explained. Baer states that very strong experimental evidence shows that on average, for a given program, the number of faults incurred by FIFO is greater than the number of faults incurred by LRU [2]. On the other hand, Belady showed that FIFO suffers from anomalous behavior, but the same paper [3], as well as later studies by Denning [6], showed no clear winner among several page replacement policies. In general, the empirical record on paging algorithms is somewhat sparse, and even inconsistent. As recently as 1995, Wilson et. al. state that even locality, the basis of modern paging algorithms, is poorly understood [27]. A review of the literature reveals that little has changed since then.

5. BENCHMARKS AND EXPERIMENTS

5.1 Benchmarks

Our set of benchmarks for this study include matrix multiplication (MM), permutation generation (PERMS), and a solver for the *n-queens* problem (NQ). The benchmarks were first written in Scheme and then compiled for the MTEVM. The benchmark descriptions are as follows:

- **MM**: This program was used to multiply two randomly-generated 35x35 matrices. Matrices are represented as a list of rows where each row is a list of numbers. The program first checks to see if the number of columns of the first matrix matches the number of rows in the second. If this test is passed, it performs a series of vector dot-products between the rows of the first matrix and the columns of the second. For each dot product a list representing a column is constructed by traversing the second argument to MM. The solution matrix has its number of rows equal to those of the first argument and its number of columns equal to those of the second argument.
- **PERMS**: This program takes as input a list, l, of n unique elements, and generates all $n!$ permutations of l. The list of permutations is constructed by recursively inserting the first element of l into each permutation obtained from the elements in the cdr of l. The benchmark was executed using as input a list of 7 randomly generated integers.
- **NQ**: In chess, the queen is able to attack from all 8 directions and at any range. Therefore, any piece located with a direct line-of-sight on the same rank, file, or diagonal as a queen is liable to be taken. This program generates a list of solutions to the classic computer science exercise of how to place n queens on an $n \times n$ chessboard without having any of them threaten each other. The benchmark was used to find the 92 solutions for placing 8 queens on an 8×8 board. Solutions are represented as a linked list of eight numbers representing the positions in each column where a queen can be placed.

The number of heap, stack, and code space accesses performed by each benchmark is displayed in Table 1. The most memory intensive of the benchmarks is NQ having generated 55,671,810 combined accesses to the heap, stack, and code space. NQ was followed by MM and PERMS with 12,011,001 and 2,419,006 combined accesses respectively. The number of code space accesses generated by each benchmark is roughly half the number of corresponding stack accesses. This is because each instruction that is fetched from code space performs, on average, two stack manipulations. The primitives that do not access the stack are the flow control (i.e. branching primitives and GOTO) and the register-to-register transfer primitives. The primitives that perform the most stack manipulations are those

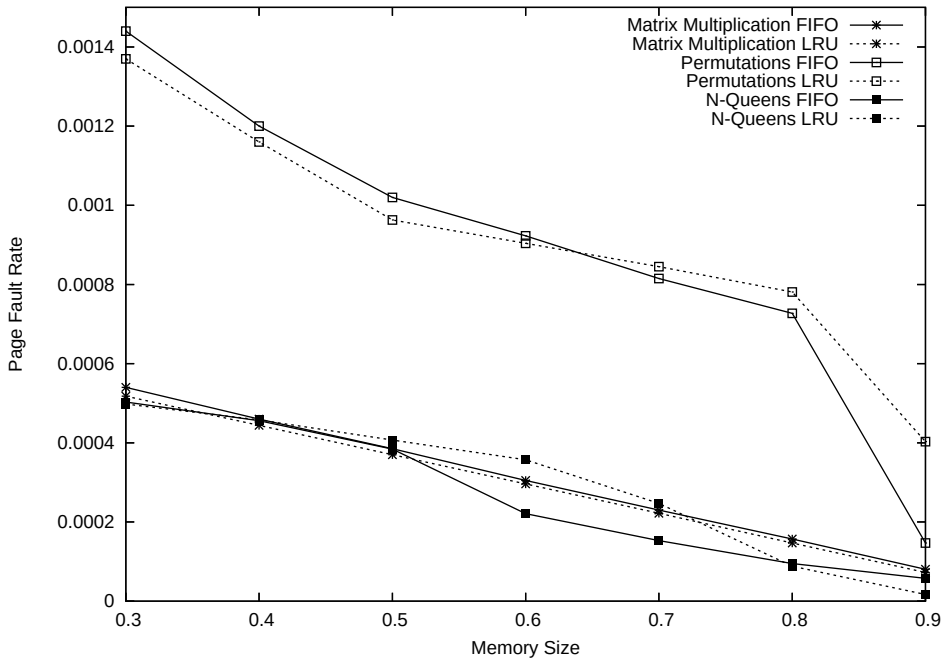

Figure 2: MT heap page fault rates for MM, PERMS, and NQ.

Memory Size	MM	PERMS	NQ
30%	0.0422	0.0540	0.0105
40%	0.0364	0.0381	-0.0043
50%	0.0405	0.0612	-0.0564
60%	0.0311	0.0217	-0.3798
70%	0.0363	-0.0349	-0.3820
80%	0.0625	-0.0692	0.0741
90%	0.1111	-0.6341	2.3846

Table 2: Relative Differences in Paging for the MT Heap.

related to the construction and removal of activation records of non-primitive functions. A number of code space accesses that exceeds half the number of corresponding stack accesses indicates that that there is more branching and function calling being performed. This can be expected of functional programs, because they usually contain many functions.

5.2 Experimental Methodology

A test run of each of the benchmarks was initially done to determine the minimum required number of heap, stack, and code pages needed for execution. Each benchmark was executed 7 times for both FIFO and LRU. For each execution, the number of frames allocated to the heap, the stack, and the code space was incremented by 10% of the minimum number of pages needed starting at 30% and going up to 90%. For each of the experiments, the number of accesses and the number of page faults were computed. From these numbers we computed page fault rates[2], and the relative difference[3] between FIFO and LRU/MTSRA. Each experiment was performed by first loading the compiled code for one of the benchmarks into MT's code space and then entering a string into the MTEVM's Scheme interpreter representing the initial call to the benchmark. The interpreter parsed the string into an abstract syntax tree, setup the MTEVM to apply the function to its arguments, allowed the MTEVM to carry out the computation, and printed the value returned by the MTEVM.

[2]The page fault rate is defines as $\frac{PageFaults}{NumberOfAccesses}$.

[3]The relative difference is defined as $\frac{FaultRate_{FIFO} - FaultRate_{LRU}}{FaultRate_{LRU}}$.

6. STACK AND HEAP PAGING REVISITED

In this section we present empirical measurements taken on the paging behavior of the MT heap and stack in the new C++ implementation. The results confirm that the performance gap between FIFO and LRU is too small for MT heap pages to justify the use of LRU. In fact, FIFO performs better than LRU for some of the experiments. The results also validate that MTSRA is superior to FIFO as a paging policy for the MT stack.

6.1 Heap Paging

Figure 2 displays the page fault rates for the MT heap. The graph shows that FIFO's performance closely follows that of LRU. FIFO is competitive due to the manner list-based

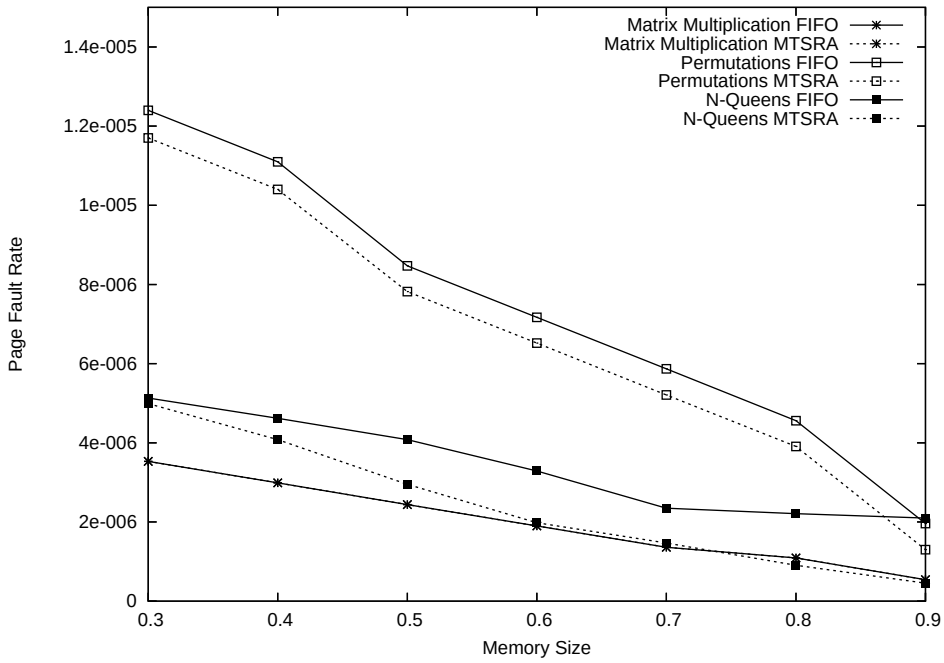

Figure 3: MT stack page fault rates for MM, PERM, and NQ.

Memory Size	MM	PERMS	NQ
30%	0.0000	0.0556	0.0284
40%	0.0000	0.0625	0.1319
50%	0.0000	0.0833	0.3846
60%	0.0000	0.1000	0.6571
70%	0.0000	0.1250	0.5962
80%	0.0000	0.1667	1.4375
90%	0.0000	0.5000	3.6250

Table 3: Relative Differences in Paging for the MT Stack

structures are stored in memory by the MT heap allocation algorithm. Simple lists, whose elements are not compound, are allocated linearly in consecutive memory locations. For the allocation of such lists FIFO performs as well as LRU, because it requires a linear traversal of consecutive pages [15]. Similarly, if a simple list creation in MT is followed by a traversal of the list, FIFO performs as well as LRU [15]. The allocation of trees (i.e. lists that contain compound data) in MT has the property that the backbone list is allocated in consecutive memory locations for each sublist. Thus, for the traversal of these backbones we expect FIFO to perform as well as LRU. Differences in paging behavior arise when a subtree is traversed [15].

The relative difference between FIFO and LRU is displayed in Table 2. For MM (which generated the most heap allocations), LRU outperformed FIFO, but the difference in performance never exceeded 0.12 with biggest relative differences occurring at 80% and 90% of the total number of heap pages needed to execute the benchmark. For these main memory sizes, the difference in the number of actual page faults was less than 10 and would not be a significant factor in performance. Between 30% and 70%, the average performance difference was less than 0.04. For the PERM benchmark, LRU outperformed FIFO by differences that never exceeded 0.10 when the number of frames was set to less than 70% of the total needed. At 70% and above, FIFO performed better than LRU. For N-Queens, the performance of FIFO and LRU are virtually the same for memory sizes between 30 and 50% of the total number of pages needed with FIFO performing slightly better than LRU. FIFO outperformed LRU by a significant margin at a memory sizes of 60% and 70% of the total number of heap pages needed for execution. The relative difference was approximately 0.38 with a significant real difference in page faults of 207 and 144 page faults respectively. LRU outperforms FIFO at 90% of the total number of heap pages needed to execute NQ. For this size of heap memory, LRU has enough memory to retain all the pages that contain live data allowing it to quickly to remove from the heap frames the pages that are garbage. FIFO, on the other hand, swaps out pages being used before swapping out pages containing garbage and, thus, causes more faults than LRU. For programs manipulating very large list-based data structures it is unlikely that a machine can hold 90% of the used live data in main memory. Therefore, the significance of LRU outperforming FIFO

for this single execution of the NQ benchmark is lessened.

Based on the empirical data of the paging behavior of the MT heap, we conclude that FIFO, instead of LRU, ought to be used as its page replacement policy. We do not, however, have a proof that FIFO is always better than LRU. Until such a proof is available, if at all, it will always be necessary to continue to study the paging performance of the MT heap with new benchmarks that exhibit page access patterns that differ from the tested benchmarks.

6.2 Stack Paging

The page fault rates for the MT stack using FIFO and MTSRA are shown in Figure 3. The matrix multiplication benchmark shows a rare case in which both paging algorithms performed identically. There is no difference in the number of page faults between FIFO and MTSRA. Both algorithms fault on the exact same pages. The stack grows to a maximum height, performs work with the memory resident stack pages, and shrinks before returning the result. Under these circumstances, there can not be a difference between FIFO and MTSRA. A growing stack is a linear traversal of pages for which FIFO and MTSRA perform the same. The ordering of stack pages between FIFO and MTSRA differs while the stack grows and shrinks within the memory resident stack pages. This difference in ordering does not affect paging behavior, because no faults occur. Faults begin to occur again only after the stack begins to shrink. FIFO and MTSRA swap out different pages (FIFO the bottommost stack page while LRU the topmost stack page), but since neither page is accessed again this difference has no impact on paging performance. The PERMS benchmark shows a difference between FIFO and MTSRA, but the curves for the most part ran parallel to each other. The real difference between MTSRA and FIFO is a single page fault for each of the seven experiments. This indicates that, as for MM, the stack grows to a maximum, work is performed without faulting, and faults occur again when the stack shrinks. The difference with PERMS is that during stack shrinking a single page swapped out by FIFO was needed before the shrinking could continue. This is probably caused by an activation record that straddled a page boundary. The NQ benchmark displays a paging performance difference between FIFO and MTSRA. For this benchmark, the stack grows and shrinks beyond the pages currently held in the set of heap frames. Therefore, FIFO's performance can not match that of MTSRA which, as mentioned before, is optimal.

Table 3 shows the relative differences in stack paging for the three benchmarks. The differences for MM were zero, because FIFO and MTSRA performed identically. The relative differences for the PERMS benchmark increases with the amount of memory allocated, suggesting a real difference in performance where none exists. In this case, the relative difference is misleading because MTSRA only outperformed FIFO by one fault for each PERMS experiment. Naturally, as the number of page faults decreases and the real difference in page faults remains constant the relative difference increases. For NQ, the relative difference between FIFO and MTSRA grows as more frames are allocated to the stack. This growth is expected for computations that make the stack grow, shrink slightly past the lowest numbered stack page in memory, and then start growing again. This causes FIFO to fault on the pages it has just swapped out.

Given that we have not changed the stack accessing discipline in MT, we expected MTSRA to outperform FIFO. It was surprising, however, to see FIFO perform just as well as MTSRA for two benchmarks. If MTSRA incurred a per access overhead as LRU, the data may have suggested further study of FIFO. When using MTSRA, as with FIFO, the victim page is always known. Thus, we conclude that the use of MTSRA ought to be continued. If the stack accessing discipline is changed, however, stack paging must be revisited.

7. VIRTUAL MEMORY PERFORMANCE OF CODE SPACE

The size of the compiled images of our benchmarks is relatively small. This forced us to choose a page size of 16 S-expressions for code space in order to conduct this initial study. In this section, we present preliminary empirical data on the paging behavior of MT's code space that suggests that LRU performs better than FIFO as a page replacement policy. FIFO, however, is a competitive paging algorithm for code space.

The page fault rates in MT's code space are displayed in Figure 4. With the exception of N-Queens producing a page fault rate of approximately 0.032 at 30% of the total number of code space pages needed to execute the benchmark, the page fault rate for each of the benchmarks is below 0.012 for both FIFO and LRU. This data suggests that FIFO may be a viable paging algorithm for MT code space. Increasing the number of frames available to 50% or more of the total number of code space pages required to run a benchmark results in sharp drops of 2 to 3 orders of magnitude in the page fault rate for both FIFO and LRU. In MM, the knee of the graph occurs at 50%. For PERMS, the knee occurs at 60%, and for NQ, it occurs at 40%. For memory sizes beyond the knee of the graph, the page fault rates dropped to below 0.00009.

Table 4 shows the relative difference between FIFO and LRU in paging MT's code space. For MM, FIFO performs better than LRU for some memory sizes, but its relative performance advantage never exceeds 0.20. LRU, on the other hand, outperforms FIFO by a relative difference of almost 0.80. Once the number of frames allocated to code space is past the knee of the graph (i.e. 50%) the page fault rate is not a significant factor in performance for both FIFO and LRU with the real difference in page faults never exceeding 3. For both PERMS and NQ, LRU outperforms FIFO in all the experiments. In some experiments the number of faults for FIFO is more than twice that of LRU.

Even though LRU performs better than FIFO, we can not conclude that LRU ought to be used. Despite LRU's superiority, FIFO is a competitive paging algorithm for the MT code space.

8. CONCLUSIONS AND FUTURE WORK

In this paper we briefly described the MT system and its five distributed memory spaces implemented in C++. The goal

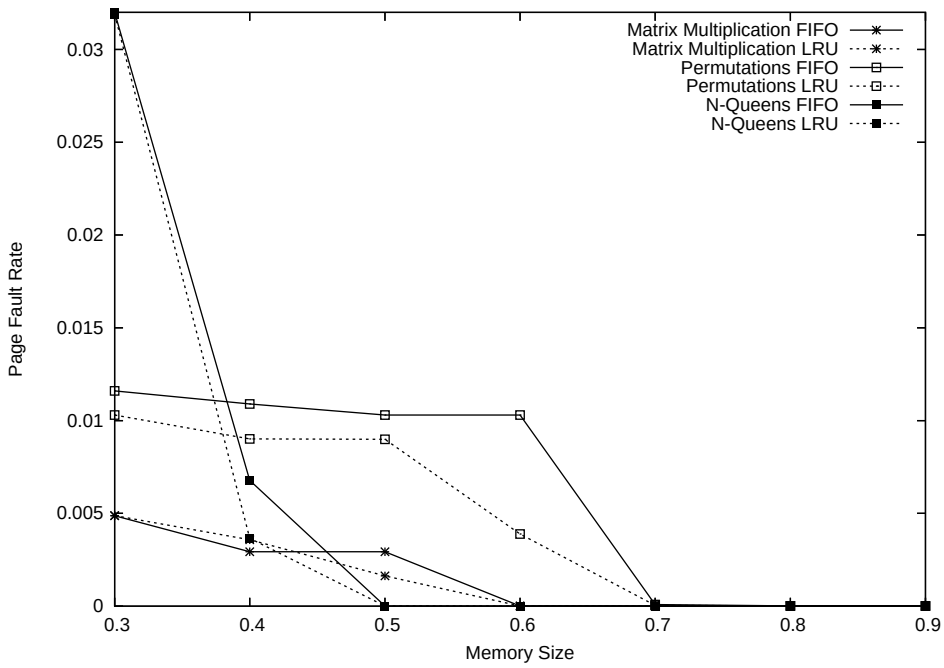

Figure 4: MT code space page fault rates for MM, PERM, and NQ.

Memory Size	MM	PERMS	NQ
30%	-0.0019	0.1235	0.0032
40%	-0.1814	0.2117	0.8493
50%	0.7974	0.1422	0.0719
60%	-0.0526	1.6406	0.0368
70%	0.1333	0.8710	0.2700
80%	0.2727	0.5000	1.6739
90%	0.3000	1.7500	1.7692

Table 4: Relative Differences in Paging for the MT Code Space

of this project is to study the interaction between functional languages and virtual memory and to test novel implementation techniques for functional languages. Although the system is currently implemented as a simulated DVM on a single-processor machine, the memory will be distributed on a networked cluster.

We presented measurements for the paging behavior in three of MT's memory spaces during the execution of three benchmarks written in a functional language. For the MT heap and stack, we reaffirmed previous findings that stated FIFO ought to be used as the paging policy for the MT heap and MTSRA ought to be used as the paging policy for the MT stack. We also presented for the first time benchmark data for the paging behavior of the MT code space. This data suggests that LRU performs better than FIFO, but that FIFO is a competitive paging policy for the MT code space.

The data presented for the MT code space suggests an interesting line of research that we plan to pursue if our conclusions remain valid for larger benchmarks. We noted that page fault rates drop dramatically after the knee in each of the graphs. The knee for each graph, however, occurs at different sizes of MT code space memory. This suggests that segmentation may be more efficient than demand paging for implementing the distributed virtual memory system for MT's code space. Our preliminary idea is to bring segments, defined as sets of functions that call each other, into the space allocated for code space at the MTEVM. In this manner, we avoid having to swap-in individual function pages every time control moves to a different set of functions.

9. ACKNOWLEDGEMENTS

We would like to thank the Department of Mathematics and Computer Science and the University Research Council of Seton Hall University for the support they have provided to undertake this research.

10. REFERENCES

[1] H. Abelson and J. Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* McGraw-Hill, 1990.

[2] Jean-Loup Baer. *Computer Systems Architecture.* Computer Science Press, 1980.

[3] L.A. Belady. A Study of Replacement Algorithms for a Virtual-storage Computer. *IBM Systems Journal*, **5**:78–101, 1966.

[4] Luca Cardelli. Compiling a Functional Language. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, pages 208–217, New York, 1984. ACM Press.

[5] R. Courts. Improving Locality of Reference in a Garbage Collecting Memory Management System. *Communications of the ACM*, 33(9):1128–1138, 1988.

[6] Peter Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, SE-6:64–84, 1980.

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

[8] Erann Gat. Lisp as an alternative to Java. *Intelligence*, 11(4):21–24, 2000.

[9] Hayes. *Computer Architecture and Organization*. McGraw-Hill, 1988.

[10] Peter Henderson. *Functional Programming: Application and Implementation*. Prentice-Hall International, Englewood, NJ, USA, 1980.

[11] Simon L. Peyton Jones and David Lester. *Implementing Functional Languages*. Prentice Hall International Series in Computer Science, 1992.

[12] P. Kelly. *Functional Programming for Loosely-Coupled Multiprocessors*. Research Monographs in Parallel and Distributed Computing, MIT Press, Cambridge, MA, USA, 1989.

[13] 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.

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

[15] 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* 5th *Brazilian Symposium on Programming Languages*, pages 80–95, Curitiba, Brasil, 2001. Universidade Federal do Paraná.

[16] Marco T. Morazán and Douglas R. Troeger. List-Heap Paging in a Distributed Virtual Memory System for Functional Languages. In Hamid Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1689–1695. CSREA Press, 2003.

[17] Marco T. Morazán, Douglas R. Troeger, and Myles Nash. Designing an All-Software Based Distributed Virtual Memory: The Paging Performance of The MT Stack. In Silvia Teresita Acuna and Cecilia Maria Laserre, editors, *Proc. of The* 1st *Iberoamerican Conference on Software Engineering and Knowledge Engineering*, pages 109–120, Buenos Aires, Argentina, 2001. Universidad Nacional de Jujuy (Editorial UNJU).

[18] 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.

[19] Marco T. Morazán, Douglas R. Troeger, and Myles Nash. The MT Stack: Paging Algorithm and Performance in a Distributed Virtual Memory System. *CLEI Electronic Journal*, 5(1), 2002.

[20] P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, 1964.

[21] Wolfgang Schreiner. Parallel Functional Programming — An Annotated Bibliography. Technical Report 93-24, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 1993.

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

[23] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, Reading, Massachusetts, USA, 1994.

[24] 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.

[25] B.K. Szymanski. *Parallel Functional Languages and Compilers*. Frontier Series, ACM Press, New York, NY, USA, 1991.

[26] William Gropp and Ewing Lusk and Anthony Skjellum. *Using MPI*. Scientific and Engineering Computation. The MIT Press, 1999.

[27] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review . *Proc. of the 1995 International Workshop on Memory Management*, Springer-Verlag LNCS, 1995.

[28] Benjamin Zorn. Comparing Mark-and-sweep and Stop-and-copy Garbage Collection. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 87–98, 1990.