The MT Stack: Paging Algorithm and Performance in a Distributed Virtual Memory System

Marco T. Morazán *
Department of Mathematics and Computer Science, Seton Hall University, USA - morazanm@shu.edu
Douglas R. Troeger †
Department of Computer Science, City University of New York, USA - csdrt@cs.engr.ccny.cuny.edu
Myles Nash ‡
Department of Computer Science, City University of New York, USA - csmwn@cs.engr.ccny.cuny.edu

ABSTRACT

Advances in parallel computation are of central importance to Artificial Intelligence due to the significant amount of time and space their programs require. Functional languages have been identified as providing a clear and concise way of programming parallel machines for artificial intelligence tasks. The problems of exporting, creating, and manipulating processes have been thoroughly studied in relation to the parallelization of functional languages, but none of the necessary support structures needed for the abstraction, like a distributed memory, have been properly designed. In order to design and implement parallel functional languages efficiently, we propose the development of an all-software based distributed virtual memory system designed specifically for the memory demands of a functional language. In this paper, we review the MT architecture and briefly survey the related literature that lead to its development. We then present empirical results obtained from observing the paging behavior of the MT stack. Our empirical results suggest that LRU is superior to FIFO as a page replacement policy for MT stack pages. We present a proof that LRU is an optimal page replacement policy. Based on this proof the MT stack page replacement policy was developed and implemented. We outline the paging algorithm and present an argument of partial correctness. The MT stack page replacement policy is superior to LRU, because it does not incur the expensive time penalties associated with implementing LRU in software.

KEYWORDS: Functional Languages, All-Software Distributed Virtual Memory, Stack Behavior, Paging Performance, Artificial Intelligence Programming.

1. INTRODUCTION

Functional languages are closely intertwined with modern software and knowledge engineering. In software engineering, LISP-like languages provide software engineers with an environment in which they can rapidly build powerful prototypes that evolve as the needs of end-users change. Furthermore, they provide an environment in which specification testing can be easily and rapidly done, and they facilitate the development of provably correct software systems. In knowledge engineering, as with most artificial intelligence (AI) applications, the inherent flexibility and orientation toward symbol manipulation of LISP-like languages make these the preemi-

*Partially supported by the Seton Hall University Research Council.
†Partially supported by NSF grant CDA-9114481.
‡Partially supported by NSF grant HRD-9703600.

nent programming languages. AI programs often manipulate complex information whose natural representations make full use of LISP's ability to create novel data structures; in addition, functional languages permit full flexibility in defining and manipulating programs as well as data.

Given the important role that functional languages play in both software and knowledge engineering, improving the performance these languages is critical. This is especially true for AI programs due to the significant amount of time and space they require [41]. Parallelism holds great promise for improving the performance of functional languages. Writing parallel programs, however, remains a difficult and non-intuitive task, despite efforts to parallelize AI programming languages such as LISP. Parallel programming is difficult, because programs must be partitioned into independent tasks, these tasks must be mapped onto a network of processors, and communication between tasks must be explicitly programmed without causing the system to deadlock [40].

In order to design and implement parallel functional languages efficiently, we propose the development of an all-software based distributed virtual memory system designed to efficiently satisfy the memory demands of a functional language. There is, however, no clear understanding of the paging behavior of functional languages. The MT system is being developed to study the design of an all-software distributed virtual memory (DVM) for a list-based pure functional language. Efficient storing and accessing of the stack data in functional languages is of particular importance, because function calling is more prevalent than in other languages. In this paper, we present results obtained from observing the paging behavior of the MT stack. Our focus in this paper is on the policy used for swapping stack pages in and out of the evaluator's local (and private) memory. We empirically establish that LRU [1] performs better than FIFO [2]. Furthermore, we argue for the optimality of LRU and present a page replacement algorithm that behaves like LRU for stack pages, but that does not incur the expensive time penalties per stack access associated with implementing LRU in software. DVM systems to date have not been designed or developed to satisfy the needs of a specific type of programming language [27, 31, 39]. To the authors' knowledge MT is the first such attempt.

2. RELATED WORK

2.1. Parallelism in Functional Languages

Functional languages which are used by AI developers, such as LISP, provide a clear and concise way to program parallel computers [11, 13, 16, 17, 25, 28, 40, 49]. These languages are attractive candidates for parallelization, because no new language constructs need to be developed to extract parallelism, the results of all programs are deterministic, deadlock can not arise, and establishing program correctness is no more difficult than in its sequential counterpart [40]. In fact, the parallel code may look the same as sequential code.

We can identify two approaches to the exploitation of parallelism in functional languages. The first has focused on running the evaluator in parallel with a garbage collector. The second approach identifies parallelism in user code. In this approach, the amount of parallelism that can be exploited depends on the type of code a programmer writes. One of the goals of the MT system is to foster locality of reference without employing a garbage collector. The results presented in this article were obtained from a system in which

[1] The *least recently used* (LRU) page replacement algorithm always selects as the victim for swapping out the page in memory that has not been accessed in the longest period of time.

[2] The *first-in first-out* (FIFO) page replacement algorithm always selects as the victim for swapping out the page that has been memory resident for the longest period of time.

garbage collection was not enabled. Therefore, we will not survey distributed garbage collection literature. The interested reader is referred to [1] for a survey of distributed garbage collection algorithms.

There have been several attempts to parallelize user code [6, 15, 18, 29, 30]. These approaches attempt to evaluate arguments in parallel (horizontal parallelism) or attempt to pipeline processes that produce values with processes that consume values (vertical parallelism). Some systems require the programmer to develop parallel algorithms [40], to use predefined templates [6], or to use annotations [29]. These approaches break the abstraction barrier that functional languages provide and require the programmer to identify parallelism, thus, focusing the programmer away from the problem they want to solve.

Systems that do not require the programmer to identify parallelism have not achieved their theoretical potentials [15, 35]. Some of the inefficiency has been associated with the excessive copying of data between processors or the amount of dereferencing of objects that are not locally stored [15]. In fact, Goldberg [15], who used matrix multiplication as one of his benchmarks, concluded that exploiting implicit parallelism proved successful only for programs without large shared data structures and that more work needed to be done on data partitioning.

Parallel functional languages have also fallen victim to granularity. That is, these systems will take longer than their sequential counterparts to execute some programs due to communication overhead and/or to processor speed. The granularity problem, as suggested by Goldberg's conclusion, is in part due to storage. That is, how large data structures are stored and accessed is what makes the granule of computation too small. There have been very few initiatives to design and develop an efficient distributed memory system (i.e. storage system) that is needed by parallel functional languages. The exporting, creation, and manipulating of processes has been thoroughly studied [42], but none of the necessary support structures (e.g. virtual memory and network topology) for the abstraction have been studied in-depth to date.

The pHluid system [9] is a distributed memory implementation of a parallel functional language. The compiler for this system produces fine grain multithreaded code which generates parallelism at two levels: horizontal parallelism and parallel communication. Parallel communication is achieved by overlapping the communication requests of the multiple threads executing at one node. One of the main objectives of the system is to minimize communication. The pHluid system separates objects that can be explicitly deallocated from those that need to be garbage collected. In this manner, the amount of communication generated by the garbage collection processes is reduced. The memory system for pHluid is not based on a DVM system and the only distributed data structure is the heap. Each processing node owns a part of the heap. All processing is halted when a node runs out of heap space to participate in a global garbage collection process.

The approaches taken to exploit parallelism have mostly focused on analyzing user code. An exception have been systems that run a parallel garbage collector. Both of these approaches, however, have failed to parallelize the engine that evaluates programs. The MT System, in contrast, is an initiative to apply parallelism to memory management beyond garbage collection. The main goals of MT are to speed-up memory accesses and memory management during program evaluation by parallelizing the engine that executes programs.

2.2. DVM Systems

General purpose DVM [3] systems have been developed to provide a single large address space within a multiprocessor. The idea is to provide the illusion of a single address space comprised of the memories at each processor. In order to provide this illusion several design choices must

[3] In the literature some authors also use the term distributed shared memory.

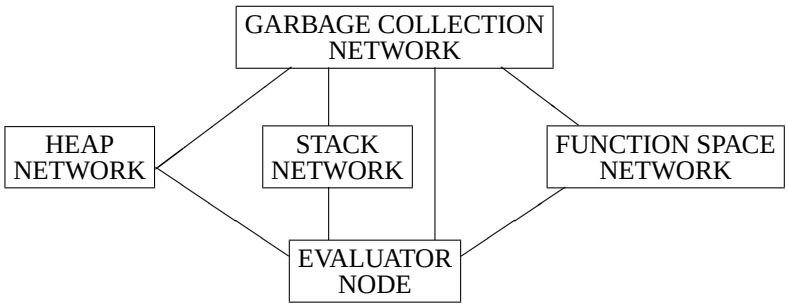

Figure 1: **Abstract view of the architecture of an MT node**

be made regarding structure, granularity, access, coherence semantics, scalability, and heterogeneity [39].

Structure refers to the layout of data in memory. Most DVM systems do not structure memory beyond viewing it as an array of pages like, for example, Ivy [31]. In MT, heap memory and stack memory is structured as an array of pages. Each page is an array of words where each word can hold an S-expression (i.e. nil, a value, a symbol, or a cons-cell). Heap and stack memory as well as the other MT components (see below) will have their own address space. In this regard, MT is novel being the only DVM system with multiple address spaces.

Granularity refers to the size of the sharing unit. In traditional DVM systems sharing occurs between processes that need to access the same data on different processors. In the current version of MT, sharing occurs between the different components of a functional system. The sharing unit in MT is a page. Pages have been used in other DVM systems such as Ivy [31] and Mirage [10].

Coherence semantics define how memory updates are propagated throughout the system. Functional programmers assume a coherence semantics called *strict consistency* and this is what is implemented in MT. In a system with strict consistency a read operation returns the last value written to the memory location requested. In a general purpose DVM system this is an ambiguous concept, because many processes may attempt to write to the same location at the same time. In the current version of MT, the semantics are clear given that there is only one evaluator and only the evaluator can mutate data.

Scalability defines how the system's performance is affected by an increase in the number of available processors. Unlike tightly-coupled processors [48], the distributed nature of the MT System potentially permits the number of processors to increase without memory becoming a bottleneck. Furthermore, there are no global broadcasts or propagation of clock signals that can cause a bottleneck.

The degree to which different types of processors are used is called heterogeneity. The current MT System is a homogeneous system. The all-software nature of our design, however, facilitates the integration of different types of processors.

3. THE MT SYSTEM

The MT architecture is based on an all-software DVM tailored to the needs of a pure functional language [34, 36]. Unlike previous attempts to apply parallelism to functional languages by parallelizing user code, the primary goal of the MT system is to exploit parallelism to make memory management faster. In MT, the basic computing entity is an MT node instead of a processor.

At a high level of abstraction, each MT node consists of five different memory spaces: the heap, the stack, the function space, the evaluator, and the garbage collector (the major components of a functional system). Figure 1 displays an abstract

view of an MT node which can be implemented on a Beowulf machine [47] using MPI [19]. Each component of the language is has its own backing store which is managed by an all-software DVM system. In essence, each MT node gives its evaluator an intelligent backing store that organizes data and code according to the demands of the evaluator and/or any other component. In contrast, pHluid [9] and GUM [32, 33] only attempt to efficiently provide the illusion of a large shared memory space and do not attempt to exploit intelligence to separate program evaluation from memory management locally at each computing element. If a system with multiple MT nodes were desired, data structures can be shared by communicating via the DVM system without interrupting program evaluation while simultaneously providing efficient local access to distributed data structures.

The management of the distributed address space is done in the management networks of a MT node in parallel with the computation taking place at the evaluator. Unlike classical virtual memory on sequential systems, MT's separate memory spaces suggest that each component does not have to share a common address space with the other components.

The results we present in this article are of interest to designers of sequential functional languages as well as designers of parallel functional systems that parallelize user code. Designers of sequential functional systems will be interested in the characterization of the stack access patterns that we provide. In addition, our results suggest efficient ways of organizing memory which is still a major concern in sequential systems as evidenced by [43]. Designers of traditional parallel functional systems will be interested in our results, because they suggest an efficient mechanism to share data between multiple evaluators. Figure 2 displays an abstract view of a system with multiple MT nodes. Each MT node is a computing element that can communicate with other MT nodes through the interconnection network. The interconnection network may or may not implement a fully connected graph (e.g. the interconnection network may be a hypercube).

This approach is expected to deliver superior virtual memory performance for parallel functional languages by providing efficient access to a large virtual memory space within a MT node. In addition to efficient sharing of data within a MT node, MT also offers the possibility of having a truly parallel garbage collector, parallel replication for data sharing, multiple paging policies (for the different components), multiple views of memory (e.g. pages and segments), parallel communication, and parallel resolution of faults.

3.1. The MT Language

The MT language is a pure subset of LISP which includes the arithmetic operators, the relational operators, the Boolean operators, cons, car, cdr, and the predicates null?, eq?, atom?, list? and symbol?. In addition, the MT language has two random number generating primitives. We chose this small subset of LISP in order to gain insight into how virtual memory is used by a list-based system. Once an efficient DVM is designed and implemented for this small language, we can build on it to add more complex objects such as closures and continuations. The implementation of these more complex objects is still a major concern as evidenced in recent literature [5, 23, 44].

The MT set of primitives is contained in most Lisp implementations (e.g. [2, 7, 8, 21, 22]). This subset includes the primitives that are most commonly used by programmers [45]. Computational, heap, and stack intensive programs have been written in MT such as matrix multiplication, graph searching, and a metacircular interpreter for MT (all of which are part of our benchmarking set). The semantics of the MT primitives are the same as those of Common Lisp or Scheme. The MT evaluator is implemented using applicative-order evaluation and uses the MT allocation algorithm for heap allocation [36].

The evaluator node runs an interpreter for the MT language. The current version of MT is implemented on a network of transputers [24]. This

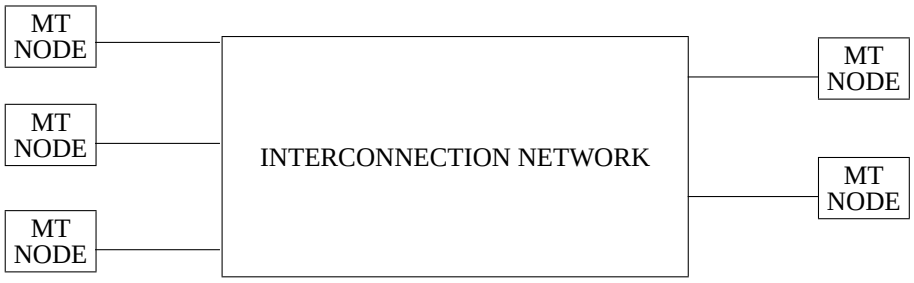

Figure 2: **Abstract view of a traditional parallel functional system with 5 MT nodes**

choice was made due to availability; however our system is typical of LISP implementations and there is nothing that makes our results specific to the transputer. The function space has not been distributed and the garbage collector has not been enabled in the current version of MT. This fact, however, does not affect the memory access patterns of the data stack (see below for the definition of the data stack) which is the focus of this article. The next version of MT is being develop to study the memory demands made by distributing user-defined function space, by making functions first-class, and by adding closures and continuations.

3.2. The MT Data Stack

The MT data stack (henceforth the MT stack) is used for parameter passing. That is, the arguments passed to a function are stored on the stack. Along with the arguments to a function, each activation record also contains the address of the previous activation record [4]. Each stack page can hold 512 objects (e.g. numbers, symbols and cons-cells). Each object is 72 bits wide making each page roughly 4KB which is a commonly used page size in modern computers.

The MT stack has its own address space and is not heap allocated despite the fact that stack memory is dynamically allocated. The decision not to heap-allocate the stack was based on the observation that stack memory can be recycled without having to call a garbage collector. The memory space used by an activation record can be immediately recycled after the function it was created for is applied. Thus, after popping an activation record off the stack the space it occupied can immediately be used for the next activation record. In Figure 1, the virtual channel between the stack network and the garbage collection network only exists because the stack may contain pointers into the heap that are needed to identify heap memory that can safely be recycled when the collector is enabled.

When a function is called an activation record is created to hold the arguments it is passed. As each argument is computed it is pushed on the stack. The construction of an activation record is complete when all arguments to a function are evaluated. This construction may be temporarily suspended in order to create other activation records to compute the values of the arguments being passed in. The partially computed activation record is left on the stack and any other activation records needed to compute arguments are pushed on top of it. When an argument is computed it is returned on the top of the stack which is in MT the location where that argument is to be stored in the suspended activation record. When the construction of an activation record is complete the function it was created for is applied.

We present the empirical data collected from 80 experiments using our benchmarks. Data on stack fault rates for both FIFO and LRU along with the relative difference between the two is presented. In addition, we present an argument establishing the optimality of LRU for the MT

[4] Flow control information (e.g. return address) is stored in the function stack that is part of the function space in MT.

stack. We then proceed to describe the MT stack page replacement algorithm that simulates LRU and avoids the overhead traditionally associated with software implementations of LRU.

The data on stack fault rates refers to how pages are swapped between the evaluator and the stack network. That is, they serve as an indication of the expected traffic on the virtual channel between the evaluator node and the stack network which is represented as a line segment in Figure 1. In contrast with other studies [9, 32], the first goal of MT is to make local access of distributed data structures efficient by studying the paging performance of the different memory spaces. After we have achieved this goal, we can focus on the task of how to efficiently share these data structures by exploiting parallel accesses and replication.
The fault rates reported count all accesses to stack data regardless of how the local memory hierarchy stores data at the evaluator. The use of registers and cache memory, for example, does affect memory access time, but does not affect the number of times the MT stack is accessed. Our software system does not assume the availability of registers, cache, and/or any other special hardware beyond the existence of random access memory at each processor that is part of an MT node. All stack data is held in a data structure defined in software.

4. VIRTUAL MEMORY PERFORMANCE OF THE MT STACK

We chose five programs to run as benchmarks. The selected programs are representative of many pure Lisp programs; we note in passing that Gabriel's benchmarks [12] use assignment. The MT measurements presented in this section were taken on a system employing an exclusive pool of frames for the stack pages. That is, the stack does not compete with other components for memory frames (i.e. for memory space) with other MT components at the evaluator node. Thus, the results pertain exclusively to stack behavior.

4.1. Benchmarks Used

A brief description of the benchmarks used is given below. The number of run-time stack accesses for each benchmark is presented in Table 1 which range from 3.8 to 22.4 million. The number of stack accesses performed by our benchmarks is comparable to the total number of memory accesses performed by benchmarks used by Bobrow et. al. [4]. In their study, the total number of memory accesses ranged from 1.2 to 8.4 million. To the authors' knowledge past studies have not specified the number of accesses to any individual component of a functional system.

- **ins**: This is an implementation of insertion-sort. The program was used to sort a randomly generated list of three hundred positive integers. The input list of random numbers is traversed once and the list of sorted numbers so far is repeatedly traversed until it contains all the elements of the input list.

- **qs**: This is a version of quicksort. Each sublist of unsorted numbers created is traversed twice. The first pass extracts the numbers less than or equal to the pivot while the second pass extracts the numbers greater than the pivot. These traversals, however, do not occur back-to-back. Instead, they are separated by the quicksorting of the smaller elements. This program ran on a list of one thousand randomly generated positive integers.

- **MM**: This is matrix multiplication. Each row of the matrix is represented by a list. The resulting matrix is built one row at the time. This means that computing AxB requires each column of B to be extracted multiple times (i.e. once for each row of A). After creating matrices A and B this program spends most of its time traversing list-based structures. The program was used to multiply two randomly generated 20x20 matrices.

- **fp**: This program performs a breadth-first search of an undirected and unweighted graph G to find a path from a to b. G is represented using adjacency lists. Starting with a all the unvisited neighbors of the last unvisited node are added to queue of nodes that are reachable but not yet visited. The program ends when b is reached and the path from a to b is displayed. The graph G used as input contained 26 nodes and 98 edges. The fp program spends most of its time traversing different list-based structures like the graph's adjacency lists, and a queue. This program does not necessarily traverse the whole graph structure.

- **bfs**: This program performs a breadth-first search of the same graph, G, used in **fp**. In this program, the graph is completely traversed. This program is used as input to **eval**. The results obtained by running this program directly on MT yielded very little virtual memory activity and are not reported.

- **eval**: This is an interpreter for MT written in MT. It takes as inputs an expression, an environment, and a function table. One of the reasons eval was chosen as a benchmark is the fact that it is unclear from a syntactical inspection what the dominating access pattern is. These experiments were the most memory intensive of the whole set.

4.2. Empirical Data

Table 2 presents the page fault rates [5] observed for our benchmarks. Each benchmark was executed with the same input using different sizes of memory allocated to the stack at the evaluator node. Stack memory size is presented as a percentage of the total number of stack pages needed during the execution of the benchmark in the column labeled memory size. Table 3 presents the relative difference [6] between FIFO and LRU for stack pages.

Benchmark	Stack Accesses
ins	3,855,724
qs	4,419,143
MM	5,808,857
fp	4,134,935
eval	22,430,082

Table 1: Benchmark Statistics

Unlike the MT heap, for which FIFO is a superior page replacement algorithm[34], the data strongly suggests that LRU is a better page replacement policy for stack pages. For most memory sizes, LRU achieved a significantly lower fault rate. For small memory sizes (i.e. at 20%), FIFO achieved the same fault rate as LRU for all benchmarks except quicksort. This tight performance between the two policies is expected for small memory sizes since the working set of the programs is larger than the number of memory frames allocated to the stack.

As expected, for both algorithms fault rates fall as the memory size is increased. No anomalies, such as Belady's anomaly [46], are observed for FIFO. The gains obtained from increasing the stack's memory size level off when memory can hold between 50% to 60% of the total number of stack pages used. Thus, the system should strive to allocate to the stack at the evaluator node a number of frames that can hold half of the total number of stack pages accessed. Allocating more stack frames would not justify the overhead incurred in management. Furthermore, these extra frames can probably be better employed by other MT components. In the 50% to 60% interval, we can also observe some of the largest relative differences between FIFO and LRU. This suggests that employing LRU is of critical importance for the MT stack. Since MT is an all-software based system, we can adopt different paging policies (e.g. FIFO for heap pages and LRU for stack pages). Such a design would not be considered

[5] We define the page fault rate as $\frac{Number of Stack Page Faults}{Number of Stack Accesses}$. In the number of accesses we include all accesses made to stack data located in the evaluator's private local memory regardless of where it is stored (i.e. RAM, cache, or register).

[6] We define the relative difference as $\frac{FIFO_{faultrate} - LRU_{faultrate}}{LRU_{faultrate}}$.

feasible for a hardware dependent system.

The performance gap between LRU and FIFO is much larger for the MT stack than for the MT heap [34, 37, 38]. This is evidenced by the relative difference between the two page replacement algorithms. For matrix multiplication and for eval, the maximum difference is 50%. For insertion-sort, FIFO produced up to 60 % more faults than LRU while for graph searching LRU can be twice as good as FIFO. For quicksort the relative difference reaches 600 % in favor of LRU. Such differences suggest that FIFO is not a viable page replacement algorithm for the MT stack.

5. The MT STACK PAGING ALGORITHM

All the information needed to evaluate a function is stored within its activation record. Since there are no nested functions or global variables in MT, it follows that the evaluator only needs to access the top activation record on the stack for the information it requires to apply a function or to build the next activation record. Within an activation record there may be random access but there is no random access between activation records. In fact, the MT stack accesses memory pages in a mostly last-in first-out discipline if LAR [7] $< SPS$ [8]. When and activation record straddles the boundary between two pages the top two pages may not be accessed in a strict LIFO manner. For this memory access pattern, nonetheless, LRU is an optimal page replacement policy [9].

5.1. LRU is Optimal for the MT Stack

The assumption needed about the activation record size in comparison to the stack page size is reasonable for a real system since the number of parameters to a LISP function is usually between 2 and 3 [45] which makes activation records much smaller than our page size. The following theorem establishes our claim.

Theorem 1 *Given that $LAR < SPS$, LRU is an optimal page replacement policy for the MT stack.*

Proof: For the proof, it is important to remember that stack pages are held in a set of frames that are allocated for exclusive use by the stack. This means that stack pages can not be swapped out by non-stack pages (e.g. heap pages). Without loss of generality, assume that the stack grows towards higher addresses.

As the stack grows, stack memory is allocated linearly to hold each new activation record. This means that higher numbered pages are being accessed as the stack grows. Similarly, when the stack is shrinking, stack space is deallocated linearly as each activation record is popped. Now, let low and $high$ be the index of the lowest and highest numbered stack pages, respectively, held in the evaluator's memory at any given time and let SP_{low} and SP_{high} be the pages themselves. Since allocation and deallocation of activation records is always linear, we have that all stack pages from SP_{low} to SP_{high} are contained in the stack frames at the evaluator before the first fault occurs. Under these conditions the evaluator can only fault if it tries to access SP_{low-1} or SP_{high+1}. Faulting on SP_{high+1} means the stack is growing and that the least recently used page is the one numbered SP_{low}. This page is also the memory resident page that will not be accessed for the longest period of time since all pages from $high$ down to $low + 1$ must be accessed at least once before accessing SP_{low} again. This follows from the assumption that all activation records are smaller than a page. Thus, when faulting on SP_{high+1}, LRU selects as a victim the same page the optimal page replacement algorithm would. When the fault is serviced all pages SP_{low+1} to SP_{high+1} are memory left resident. These are all the stack pages from the low-

[7] The size of the largest activation record.

[8] The stack page size

[9] The optimal page replacement policy always picks as its victim for swapping out the page that will not be referenced again for the longest period of time [3].

Memory	**ins**	**ins**	**qs**	**qs**	**MM**	**MM**	**fp**	**fp**	**eval**	**eval**
Size	FIFO	LRU	FIFO	LRU	FIFO	LRU	FIFO	LRU	FIFO	LRU
20	2.09e-3	2.09e-3	6.65e-5	5.57e-5	2.29e-3	2.29e-3	2.29e-3	2.29e-3	2.55e-3	2.55e-3
30	1.20e-4	8.35e-5	4.30e-5	3.53e-5	2.60e-6	1.70e-6	1.60e-4	1.07e-4	1.20e-6	8.00e-7
40	1.20e-4	8.35e-5	3.71e-5	2.85e-5	1.50e-6	1.40e-6	7.62e-5	3.96e-5	1.20e-6	8.02e-7
50	4.48e-5	2.75e-5	2.67e-5	1.72e-5	1.20e-6	1.00e-6	4.10e-6	2.40e-6	3.12e-7	2.67e-7
60	4.38e-5	2.75e-5	2.42e-5	1.27e-5	1.20e-6	1.00e-6	4.10e-6	2.40e-6	2.22e-7	1.78e-7
70	4.38e-5	2.75e-5	1.63e-5	5.40e-6	9.00e-7	7.00e-7	1.20e-6	1.00e-7	2.22e-7	1.78e-7
80	2.15e-5	4.90e-6	9.50e-6	2.70e-6	5.00e-7	3.00e-7	7.00e-7	5.00e-7	1.33e-7	9.00e-8
90	2.15e-5	4.90e-6	3.20e-6	5.00e-7	5.00e-7	3.00e-7	7.00e-7	5.00e-7	1.33e-7	8.90e-8

Table 2: Page Fault Rates for the MT Stack

Memory	**ins**	**qs**	**MM**	**fp**	**eval**
20	0.00	0.20	0.00	0.00	0.00
30	0.50	0.22	0.50	0.50	0.50
40	0.50	0.30	0.13	0.92	0.50
50	0.59	0.55	0.17	0.70	0.17
60	0.59	0.91	0.17	0.70	0.25
70	0.59	2.00	0.25	0.25	0.25
80	3.37	2.50	0.50	0.50	0.50
90	3.37	6.00	0.50	0.50	0.50

Table 3: Relative difference between FIFO and LRU for MT stack Pages

est indexed to the highest indexed which are resident at the evaluator. It follows that this condition is invariant and allows us to conclude that all subsequent faults will leave stack memory in a similar state. Faulting on SP_{low-1} means that the stack is shrinking and that the least recently used page is SP_{high} This page is also the page that will not be accessed for the longest period of time. Thus, LRU is optimal for MT stack pages. **Q.E.D**

5.2. The MT Stack Page Replacement Algorithm

In an all-software based distributed virtual memory system (local) access time is proportional to the overhead associated with LRU derived from time stamping after every access. The proof that LRU is optimal for the MT stack suggests a page replacement algorithm that will not incur the heavy penalty in access time associated with time stamping. The access time for accesses that do not cause a fault is constant. For accesses that cause a fault the code needed to be executed at the evaluator can be done in a constant amount of time. The algorithm does not require stack pages to be time stamped and only requires four variables.

The design roles of the four variables are as follows:

- $slow$: the index of the lowest numbered stack page resident at the evaluator.
- $shigh$: the index of the highest numbered stack page at the evaluator.
- $flow$: the index of the frame holding the lowest numbered page.
- $fhigh$: the index of the frame holding the highest numbered page.

Let *SFRAMES* be the number of frames allocated to the stack in the evaluator's memory and

$pagenum$ be the index of the stack page that is to be accessed. The memory and faulting algorithm can be described as follows:

1. Load evaluator stack frames from 0..*(SFRAMES-1)* with the stack pages 0..*(SFRAMES-1)*. Set $slow$ and $flow$ to 0 and $shigh$ and $fhigh$ to *(SFRAMES-1)*.
2. Upon receiving a stack request determine if it is between $slow$ and $shigh$. If so goto 3 else goto 4 to service the fault before accessing the stack.
3. Access frame *((flow + pagenum - slow) MOD SFRAMES)* by the appropriate displacement. Goto 2.
4. If $pagenum > shigh$
   (a) Victim is SP_{slow} and is stored in frame $flow$.
   (b) Swap SP_{slow} out to backing store
   (c) Swap requested page in from backing store into frame $flow$.
   (d) Increase all four variables by *1 (MOD SFRAMES)*.
   (e) Access the requested page and goto 2.
5. If $pagenum < slow$
   (a) Victim is SP_{shigh} and is stored in frame $fhigh$.
   (b) Swap SP_{shigh} out to backing store.
   (c) Swap requested page in from backing store into frame $fhigh$.
   (d) Decrease all four variables by 1.
   (e) If ($flow$ or $fhigh$) becomes negative set it to *(SFRAMES-1)*.
   (f) Access the requested page and goto 2.

5.3. Proof of Partial Correctness

In order to establish the correctness of the MT stack replacement algorithm (MTSRA), we must prove that MTSRA causes the same paging behavior as LRU. That is, we must establish that when a fault occurs under MTSRA the following holds:

- LRU would fault on the same page.
- MTSRA will swap out of the evaluator's memory the same page LRU would swap out.

Proving that the following assertions are invariant will help us establish that MTSRA produces the same paging behavior as LRU:

- Stack pages $slow$ to $shigh$ are held in the evaluator's memory.
- Stack pages $slow$ to $shigh$ would also be held in the evaluator's memory under LRU.
- $stackpage_i$ is held in $stackframe_m$ where *m = (flow + (i - slow)) MOD SFRAMES*.

The memory access stream of program P written in MT on input x will always be the same regardless of the paging algorithm employed. Therefore, we know that MTRSA is equivalent to LRU if it always faults on the same page LRU would fault on and if it always swaps out the same page LRU would swap out.

Proof: The proof is achieved by induction on the number of times a stack fault occurs. For the base case, we must establish that the invariant properties hold when there are no faults. Step 1 of MTSRA initializes stake frames in order with $SP_0..SP_{SFRAMES-1}$, initializes $flow$ to 0, and $fhigh$ to *(SFRAMES - 1)*. Stack frames are initialized in the same manner for LRU. LRU and MTSRA would hold the same pages, $SP_0..SP_{SFRAMES-1}$, in the evaluator's memory. SP_i, *0 ≤ i ≤ (SFRAMES - 1)*, is stored in the frame given by *m = (flow + (i - slow)) MOD SFRAMES*.
In addition, there is no paging activity so all the invariant properties hold for our base case.

For the inductive step, assume that the three invariant assertions hold after k stack faults. As long as another fault does not occur LRU and MTSRA exhibit the same paging behavior. Recall that activation records on the MT data stack are accessed in a LIFO manner and that the size every activation record is strictly smaller than the stack page size. When a fault occurs, it can only

be caused by attempting to access SP_{slow-1} or $SP_{shigh+1}$. Faulting on SP_{slow-1} means that MTSRA swaps out SP_{high} and correctly updates the four variables needed by MTSRA to restore the invariant. By the inductive hypothesis we know that LRU will have the same pages in memory. Since the memory access stream is the same for LRU and MTRSA, we can conclude that LRU would also fault on SP_{slow-1}. Faulting on SP_{slow-1} means that the stack is shrinking. Given that activation records are accessed in a LIFO manner and that the stack page size is larger than any activation record, SP_{high} is the page the has not been accessed in the longest period of time. This is the page LRU selects for eviction. MTSRA will swap out the page stored in $frame_{fhigh}$ which by the inductive hypothesis is SP_high. An analogous analysis establishes that faulting on $SP_{shigh+1}$ will cause MTSRA to behave like LRU. Thus, we can conclude that MTSRA and LRU exhibit the same paging behavior. **Q.E.D.**

5.4. MTSRA is Optimal

The above proof establishes that MTSRA causes the same paging behavior as LRU meaning that they both fault on the same pages and they both swap out the same page for any given fault. When MTRSA swaps out the least recently used page it is swapping out the memory resident page that will not be accessed for the longest period of time as established in Theorem 5.1. This means that MTSRA is optimal for MT stack pages.

MTSRA is superior to LRU, however, because it does not incur the overhead associated with implementing LRU in software. MTSRA does not time stamp a page after every access, page searching time is constant since it always knows where the page to be swapped out is stored, and no page table is required. These properties significantly reduce the stack's effective access time and fault service time.

6. FUTURE WORK AND FINAL REMARKS

The MT system is being developed as a research tool and test bed for the development of a distributed virtual memory system tailored to the demands of a pure parallel functional language. This is the first effort made to design the necessary storage support needed by a specific type of language. This approach is expected to provide the necessary tools for data sharing and accessing both at the intra-node and inter-node levels.

We have empirically established that LRU is superior to FIFO as a page replacement policy for MT stack pages. For small stack memory sizes FIFO will perform as well as LRU because the system's paging performance will be poor. If the evaluator can only cache a small number of pages then the stack's working-set will be too large causing excessive paging. As stack memory space is increased the performance of both FIFO and LRU improves, but LRU's performance significantly surpasses FIFO's performance.

We have also proven that LRU is optimal as a page replacement policy for MT stack pages. Based on our proof, we have developed the MT stack page replacement algorithm which is also an optimal page replacement policy for MT stack pages. We have established the partial correctness of our page replacement algorithm. This algorithm yields a memory access time that is constant for accesses that do not cause a fault. The code needed by the evaluator to resolve a fault is also executed in a constant amount of time. Furthermore, MTSRA is also superior to LRU, because stack pages do not have to be time stamped. The time and space complexity of the MT stack page replacement algorithm is superior to that of LRU.

Stack space in MT is further being studied to establish how efficiently stack frames are being used. Given that MT has several address spaces it is important to determine how efficiently frames are being used. We expect our findings to suggest ways to initially allocate frames and how to dynamically change frame allocations at the eval-

uator.

The MT system is currently being expanded to study the memory accessing patterns to function space. In this new version, functions will be first-class and the function space will be distributed. One of the interesting questions that we will need to answer is whether any changes necessary should be integrated into the current MT memory spaces and data structures or whether new memory spaces and data structures are justified. In particular, we are studying the effects of introducing complex objects such as closures and continuations. The flexibility inherent in software design allows us to consider designs that would be considered prohibitive in hardware and design different memory spaces for the different components of a functional language.

REFERENCES

[1] S.E. Abdullahi, E.E. Miranda, and G.A. Ringwood. Collection Schemes for Distributed Garbage. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of the International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 43–81, Saint-Malo, France, 1992. Springer-Verlag.

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

[4] D. Bobrow and M. Grignetti. Interlisp Performance Measurements. Technical Report BBN Report No. 3331, Bolt Beranek and Newman Inc., 1976. Available from the National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161.

[5] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing Control in the Presence of One-Shot Continuations. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 99–107, Philadelphia, Pennsylvania, 1996.

[6] M. Cole. *Algorithmic Skeletons: Structured Management of Parallelism*. Research Monographs in Parallel and Distributed Computing, MIT Press, Cambridge, MA, USA, 1989.

[7] R. Kent Dybvig. *The SCHEME Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1987.

[8] 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.

[9] Cormac Flanagan and Rishiyur S. Nikhil. pHluid: The Design of a Parallel Functional Language Implementation on Workstations. In *Proceedings of the International Conference on Functional Programming*, Philadelphia, USA, 1996.

[10] B. Fleisch and G. Popek. Mirage: A Coherent Distributed Shared Memory. *Operating System Review*, **23**(5):78–101, 1966.

[11] Daniel P. Friedman and David S. Wise. Applicative Multiprogramming. Technical Report 72, Computer Science Department, Indiana University, Bloomington, Indiana, 1978.

[12] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge, MA, USA, 1985.

[13] J. Gaudiot and L. Lee. Multiprocessor Systems Programming in a High-level Dataflow Language. *Parallel Architectures and Languages Europe, Volume 1: Parallel Architectures (PARLE 1987), LNCS 258*, pages 134–151, 1987.

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

[15] B. Goldberg and P. Hudak. Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor. In J. H. Fasel and R. M. Keller, editors, *Graph Reduction: Proceedings of a Workshop at Santa Fé, New Mexico*, number 279 in Lecture Notes in Computer Science, pages 94–113, Santa Fe, New Mexico, 1986. Springer-Verlag.

[16] R. Goldman, R. Gabriel, and C. Sexton. Qlisp: An Interim Report. In Ito and R.H. Halstead [26], pages 161–181.

[17] D. Grit. Sisal on Message Passing Architectures. In Ito and R.H. Halstead [26], pages 721–731.

[18] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Scientific and Engineering Computation. The MIT Press, London, England, 1999.

[19] Kevin Hammond and Sharon Curtis, editors. *Trends in Functional Programming*, volume 3, Bristol, UK, 2002. Intellect. ISBN 1-8410-070-4.

[20] S. Hekmatpour. *Lisp: A Portable Implementation*. Prentice Hall, New York, 1989.

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

[22] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing Control in the Presence of First-Class Continuations. *SIGPLAN Notices*, **25**(6):66–77, 1990.

[23] Inmos. *Transputer and Occam 2 Toolset: User Guide*. Inmos, 1993.

[24] T. Ito and M. Matsui. A Parallel Lisp Language PaiLisp and its Kernel Specification. In Ito and R.H. Halstead [26], pages 58–100.

[25] T. Ito and Jr. R.H. Halstead, editors. *Parallel Lisp: Languages and Systems*, number 441 in Lecture Notes in Computer Science, Sendai, Japan, 1989. Springer-Verlag.

[26] K. Johnson, M.F Kaashoek, and A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. *Operating System Review*, **29**(5):213–228, 1995.

[27] R.H. Halstead Jr. New Ideas in Parallel Lisp: Language Design, Implementation, Programming Tools. In Ito and R.H. Halstead [26], pages 2–57.

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

[29] D. Koorey. Tlisp: A Concurrent Lisp for the Transputer. *SIGSAM Bulletin*, **26**(4):15–23, 1992.

[30] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Department of Computer Science, Yale University, New Haven, Conneticut, 1986.

[31] Hans-Wolfgang Loidl. Investigating the Memory Management in a Parallel Graph Reducer. In Markus Mohnen and Pieter Koopman, editors, *Proceedings of the* 12^{th} *Workshop on Implementation of Functional Languages*, pages 185–200, Aachen, Germany, 2000. Aachener Informatik-Berichte.

[32] Hans-Wolfgang Loidl. Load balancing in a parallel graph reducer. In Hammond and Curtis [20], pages 63–74. ISBN 1-8410-070-4.

[33] Marco T. Morazán. *Towards Fast Functional Languages Via Distributed Virtual Memory*. PhD thesis, Department of Computer Science, The Graduate School of The City University of New York, New York, New York, 1999. Available from UMI Dissertation Services http://www.bellhowell.inforlearning.com.

[34] Santos Martins. Parallel Implementations of Functional Languages. In Herbert Kuchen and Rita Loogen, editors, *Fourth International Workshop on the Parallel Implementation of Functional Languages*, RWTH Aachen, Germany, 1992. Aachener Informatik-Berichte.

[35] 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*, pages 97–104, Bristol, UK, 2000. Intellect.

[36] 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, *Proceedings of The 5^{th} Brazilian Symposium on Programming Languages*, pages 80–95, Curitiba, Brasil, 2001. Universidade Federal do Paraná.

[37] Marco T. Morazán, Douglas R. Troeger, and Myles Nash. Paging in a distributed virtual memory. In Hammond and Curtis [20], pages 75–86. ISBN 1-8410-070-4.

[38] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, **24**(8):52–60, 1991.

[39] Simon Peyton-Jones. Parallel Implementations of Functional Programming Languages. *The Computer Journal*, **32**(2), 1989.

[40] Elaine Rich and Kevin Knight. *Artificial Intelligence*. McGraw-Hill, New York, NY, 1991.

[41] Wolfgang Schreiner. Parallel Functional Programming — An Annotated Bibliography. Technical Report 93-24, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 1993. Available via ftp://ftp.risc.uni-linz.ac.at/pub/reports/parlab/pfpbib.ps.Z.

[42] Manuel Serrano and Hans-J. Boehm. Understanding Memory Allocation of Scheme Programs. In *Proceedings of the 2000 International Conference on Functional Programming*, pages 245–256, Montreal, Canada, 2000. ACM Press.

[43] Zhong Shao and Andrew W. Appel. Space-Efficient Closure Representations. Technical Report CS-TR-454-94, Department of Computer Science, Princeton University, Princeton, New Jersey, 1994.

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

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

[46] 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, London, England, 1999.

[47] Harold S. Stone. *High-Performance Computer Architecture*. Addison-Wesley, Reading, Massachusetts, USA, 1990.

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