A

Towards Fast Functional Languages Via Distributed Virtual Memory

by

Marco T. Morazán Sohnle

A DISSERTATION SUBMITTED TO THE GRADUATE FACULTY IN COMPUTER SCIENCE OF THE CITY UNIVERSITY OF NEW YORK IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

1999

This manuscript has been read and accepted by the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

September 15, 1999 — Prof. Douglas R. Troeger

Date

Chair of Examining Committee

September 15 '99 — Prof. Stanley Habib

Date

Executive Officer

Member Supervisory Committee

Prof. Carol Tretkoff
Prof. Manuel Núñez
Prof. Dominic Duggan

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

TOWARDS FAST FUNCTIONAL LANGUAGES VIA

DISTRIBUTED VIRTUAL MEMORY

by

Marco T. Morazán Sohnle

Adviser: Professor Douglas R. Troeger

Functional languages have been identified as an excellent choice for parallel programming since a parallel functional program looks no different from a sequential functional program. Unfortunately, parallel functional languages have not fulfilled their theoretical potential since they have fallen victim to granularity.

We propose a novel architecture, the MT architecture, based on an all-software distributed virtual memory tailored to the needs of a functional language. Sequential functional languages are considered slow due to their poor virtual memory interaction. The MT architecture is an effort to apply parallelism to all the automatic memory management that functional languages provide.

We present empirical evidence that favors using FIFO as a heap-page replacement policy instead of LRU. Similarly, we present empirical evidence that suggests an optimal stack-page replacement algorithm for the current version of MT. We conclude by presenting algorithms for prefetching and garbage collecting.

Dedicated to Don Quijote de La Mancha, Bilbo Baggins, and Captain Jean Luc Picard. May we all have the soul of the poet, the heart of the adventurer, and the mind of the inquisitive explorer.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

1.1.1 Parallel Programming

Demands for faster execution have led to the development of parallel computers. In general, parallel computers or multiprocessors, are extremely difficult to program due to difficulties in developing correct and efficient code. When parallel extensions of sequential imperative languages are used for parallel programming, program correctness can only be achieved if processes are correctly synchronized to avoid deadlock and nondeterministic results. Furthermore, programs must work correctly regardless of the number of processors in the system.

The difficulties in the development of efficient code can be attributed to the overhead that parallel programmers, unlike sequential programmers, must resolve to get even the simplest of programs to execute. The overhead associated with parallel programming can be itemized as follows:

- The partitioning of the program into independent tasks that can be executed in parallel.
- The mapping of the problem graph onto the network graph.
- The mapping of data in memory (whether we use a tightly-coupled or loosely-coupled multiprocessor).
- The sharing of values between processes must be explicitly coded whether it is via message passing or via shared variables
- The development of algorithms that can be easily implemented on the network with which you are working.

This list clearly indicates that there is a need for more abstract parallel programming environments. If parallelism is to become ubiquitous, it is unreasonable not to hide the details of the machines being used through the development of a very high-level programming languages.

1.1.2 Parallel Functional Languages

Functional languages have been identified as providing a clear and concise way to program parallel computers [26, 32, 31, 29, 47, 41, 60, 67]. These languages are attractive candidates 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 [60].

There have been two major approaches to the exploitation of parallelism in functional languages. The first has taken the approach of parallelizing user code. In this approach, the amount of parallelism that can be exploited depends on the type of code a programmer writes. The second approach has focused on running the evaluator in parallel with a garbage collector.

Parallelizing User Code

There have been several attempts to parallelize user code [19, 30, 33, 49, 50]. These approaches attempt to evaluate arguments in parallel (horizontal parallelism) or attempt to pipeline processes that produce values with processes that consume their values called (vertical parallelism). Some systems require the programmer to develop parallel algorithms [60], to use predefined templates [19], or to use annotations [49]. In other words, these systems break the abstraction barrier that functional languages provide and require the programmer to identify parallelism.

Systems that do not require the programmer to identify parallelism have not achieved their theoretical potentials [30, 55]. 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 [30]. In fact, Goldberg [30], 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.

Regardless of how parallelism is identified, these systems have 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 how little processor time it takes to perform the required task. The granularity problem, as suggested by Goldberg's conclusion, is due to storage. That is, how large data structures are stored and their transmission to other nodes is what makes the granule of computation too small. There has been no effort made to develop an efficient (virtual) memory system (i.e. storage system) that functional languages require. The exporting, creation, and manipulating of processes has been thoroughly studied [61], but none of the necessary support structures (e.g. virtual memory and network topology) for the abstraction have been properly designed.

Parallel Garbage Collection

One of the reasons list processing systems are considered slow is the automatic recycling of dynamically allocated memory through a process called *garbage collection.* As computation advances, memory that was allocated to hold temporary values can be reused in order to give the illusion of an infinite memory. In modern sequential systems, however, the primary goal of the garbage collector is to compact data. That is, in modern computer systems with gigabytes or terabytes of backing store the main

goal of the collector is not to give the illusion of an infinite memory. Instead, the goal of the collector is to compact data in order to reduce the size of the program's working set. Furthermore, the compacting of live data will be done in a way that fosters spatial locality of reference.

Although the mission of garbage collection has changed, the manner in which garbage collection is pursued has changed very little. Garbage collection is still criticized for:

- Interrupting the evaluator for too long in order to recycle memory space.
- The amount of paging garbage collection causes.
- The amount of time the collector and the evaluator spend synchronizing.

In order to minimize the cumulative effect of these, parallel garbage collectors have been developed. That is, collectors that run simultaneously with the evaluator. These collectors, however, are extensions of their sequential counterparts and many still require running some of the time in stop-the-world mode.

We believe that in order to have a garbage collector run in parallel with an evaluator, it is necessary to understand how the evaluator uses memory. Knowing how the evaluator uses memory (e.g. how objects are accessed) will suggest a way the collector should compact live data. It seems unreasonable to propose parallel or distributed garbage collection techniques if one does not know how live data should be com-

pacted. Perhaps this is why, as reported by Abdullahi et. al. [1], the overwhelming majority of proposed parallel or distributed garbage collecting algorithms have never been implemented. This suggests that garbage collection design should be closely tied to and be a part of system design. In other words, garbage collection should not be an afterthought. Instead, as the system or language design advances so, should the design thoughts on the garbage collector.

1.2 The Design of the MT System

We understood the problems that were being faced by the parallel functional languages community and decided that it was necessary to go back to the implementation of sequential systems to learn where parallelism can be best applied. Parallelizing user code is too difficult, because there lacks a set of good heuristics to control granularity. Instead, we focused on how to parallelize the engine that evaluates Lisp programs.

Before proceeding, let us state that we will assume that the reader is familiar with functional programming at the level presented in Abelson and Sussman [2]. It is not necessary to understand any existing language in its entirety because MT is a small subset of most functional languages. A description of the MT language can be found in Appendix A. Appendix A also presents some implementation details that may be of interest to most readers.

1.2.1 The MT System

We studied several implementations of Lisp and Scheme [2, 6, 24, 23, 27, 36, 48, 69] to determine the basic components of a functional system. At a very high-level of abstraction these are:

- **The Heap**
- **The Stack**
- **The Function Table**
- **The Garbage Collector**
- **The Evaluator**

The precise definition of each of these varies among implementations, but these data structures are common to the implementations studied. In addition to searching for the components needed by a functional engine on a Von Neuman architecture [1], we also searched for the reasons functional languages are considered slow in the hope of applying parallelism to the problem.

The literature on why Lisp is slow clearly suggests that it is due to poor interaction with virtual memory [8, 25, 56, 64]. Poor virtual memory performance is due to garbage collection, to the lack of live-data density, and to the lack of intra-list locality.

[1] We did not consider other architectures as the Data Flow Machine because they are not readily available.

Garbage collectors exhibit poor virtual memory performance, because they usually must traverse the entire virtual address space in order to determine what is and what is not garbage and . Since functional languages usually box objects (i.e. use a pointer to an object instead of the object itself), objects are prematurely allocated in the heap after being computed. This leads, for example, to lots of garbage being intermingled between adjacent members of a list. Finally, poor virtual memory performance can be expected if most pointers in a list do not point to the same page. This arises when the address space is divided up by the type of object that can be held, when garbage density is high, and when assignment is supported.

Our review of both the sequential and parallel functional languages literature, has lead us to conclude that a functional system can derive performance benefits from a distributed virtual memory fine-tuned to the memory needs of functional languages [58]. The design is based on the components of a functional system and not on the types that are supported by the system. That is, the virtual address space is divided among the components needed to evaluate functional programs. The current design only includes those components that are common to all implementations we studied and is by no means considered complete. As the system grows and features are added to MT, new components may be added to the architecture or the current components may be redefined. For example, when first class functions and closures are introduced it may be desirable to introduce a closure network to manage the backing store of

closure space. This will be determined by studying the use of closures and their memory demands.

1.2.2 Definition of MT Components and Expected Advantages of MT

The five basic MT components have very simple definitions that may or may not conform to the definitions used in the implementations of sequential functional languages. In many instances our use of the components is compatible with the use given to them in other implementations. The current implementation of MT was developed on a network of T805 transputers using the Occam 2 language. The choice of platform for implementation is not relevant to the results we are reporting since any platform would have produced the same results. That is, our results are independent of the platform chosen for implementation. For example, once we have chosen the heap page size, the number of heap frames at the evaluator, and the evaluation algorithm, the number of faults produced by LRU will be the same whether on a network of transputers or a network of Sun-stations. That is, the platform that MT is implemented on does not influence the fault rate.

The current MT design has only one (applicative-order) evaluator. We decided on this based on the observation that it is little understood how a functional language employs virtual memory. For both sequential and parallel functional languages, virtual memory has been identified as a problem but little has been done outside of

improving garbage collection to improve virtual memory performance. If n is the number of evaluators in the system, the goal is to first have a virtual memory friendly evaluator for $n = 1$ before n grows any larger. Others in the field of functional languages have agreed with this goal [42].

The current MT heap is used to hold dynamically allocated list data. The MT heap is used exclusively to store lists. It is important to note that heap and stack space are completely discrete. That is, the stack is not heap allocated. The MT stack is used for argument passing and flow control. The stack stores for each function called its arguments and its return information. The MT function/code space was not focused on in this study. In this study, the function space is simply a function table that holds all the user defined functions and that fits entirely in the evaluator's memory at all times. This decision makes the comparison to studies that include code accesses a bit difficult since we must make sure that other factors, like total number of accesses, are the same in order to have some basis for comparison. On the other hand, we provide statistics on how specific language components are used which can serve as the basis of future design

The MT garbage collector was not implemented for this study. We do, however, outline in chapter 7 a parallel garbage collection algorithm for MT based on the work in [57]. This is consistent with our belief that the collector should not be an afterthought of the design. As knowledge is gained on how virtual memory is used

the collector can be further fine-tuned until it is implemented.

Of course, the question still remains as to what can we expect to gain from MT? MT proposes an all-software based distributed virtual memory (DVM) fine-tuned to the needs of a functional programming system. All-software DVM's have been implemented in the past, producing competitive results [43]. Instead of just having data sit on a disk in backing store or creating a unified address space in a distributed system, MT proposes that backing store can be intelligent and do more than just store data and provide a common address space. Regardless of the implementation platform, here are some of the advantages the intelligence at the nodes should provide MT:

1. Multiple views of memory. Paging, segmentation, and list-structured memory can all be combined in one system. Since no hardware support is assumed any view of memory can be used or combined. The only restriction is efficiency. The MT design does not rule out any view of memory due to hardware limitations.

2. Dynamic Prefetching. Prefetching of pages and segments can become effective if the calculations of what to prefetch is done by the MT components. Although not implemented, we propose in chapter 7 a prefetching algorithm for heap pages.

3. Multiple page replacement policies. Each component should be able to use

the page/segemnt replacement policy that is best suited for it. For our current implementation of MT, we argue for the use of two different replacement algorithms.

4. Parallel resolution of faults. Given that the MT evaluator and MT garbage collector will run in parallel and will not share the same physical memory space, collector faults and evaluator can be serviced by the MT DVM in parallel.

5. Fewer faults at the evaluator. The advancement of the main computation will endure fewer faults. Parallel garbage collection and prefetching will significantly cut on the number of faults. Furthermore, a smart allocation algorithm should help the evaluator promote locality without the interference of a garbage collector.

6. Parallel Collection of Objects. The garbage collector is not restricted to collecting one list-based structure at a time. Instead, several objects should be collectable in tandem.

1.3 Summary of Results and Thesis Organization

As a reasonable starting point in the design and development of MT, we have decided to focus on the evaluator, the MT heap, and the MT stack. The MT language is a small and pure functional language. Based on this small language, we have gained

considerable insight into how virtual memory is used by functional languages. The results that we present are applicable to any list-processing language.

In chapter two, we explore how list-based structures are allocated in the heap. A classical Lisp system where all items are boxed is compared with the results of using the MT allocation algorithm. The MT allocation algorithm, in essence, unboxes all data. That is, the stack contains all literals, not pointers to literals. A natural question is to ask how are lists unboxed. When a list is passed to a function, a literal cons-cell is pushed onto the stack. The gains of using the MT allocation algorithm are established via three simple experiments. The chapter is concluded with theorems that establish that garbage is not intermingled with live data within a list, that lists contain a certain amount of linearization built into them, and that complex lists are level-linearized.

Chapter three marks the beginning of our study of virtual memory performance. We start by demonstrating that both FIFO and LRU can beat each other as page replacement policies. One of our goals was to empirically establish that LRU is the best policy for heap pages and a functional language in general. Our interest was sparked by [8] and [27], given that they seem to accept LRU *a priori* as the best page replacement algorithm for Lisp. We discovered, however, that FIFO's performance was very close of that of LRU's. Our results are presented through a case study that uses insertion-sorting. We begin to discover that virtual memory is

poorly understood by establishing that page fault rates alone are not enough to draw conclusions. It turns out that our classical allocator achieved better fault rates than MT allocation algorithm. This can be explained when one realizes that the classical system is performing many more accesses to memory than the system using the MT allocator. Thus, for the classical system each page fault carries a smaller penalty even though many more page faults are incurred. This clearly establishes that page fault rates alone are unreliable means to a conclusion, yet most virtual memory studies use them to draw conclusions. In this chapter, we also observe that the difference between FIFO and LRU is very small. The chapter ends with theorems that establish the nearly equivalent virtual memory performance of LRU and FIFO for the high-level "memory operations" of list creation, large list traversal, and the repetitive creation and traversal of simple lists. This is significant, because these operations are common in functional programming. This suggest that functional programs spend a lot of time performing memory activities for which FIFO performs as well as LRU. In addition, if one considers the overhead associated with LRU, FIFO becomes a very attractive page replacement policy.

FIFO is traditionally considered inferior to LRU and suffers from Belady's anomaly[2] [7, 65]. For these reasons, it is not implemented in systems today. Pure LRU, due to the per access overhead associated with it also is not implemented. Instead, approx-

[2]Belady found that for some access patterns under FIFO increasing the size of memory also increased the number of faults.

imations to LRU are implemented. These approximations, however, do not perform as well as LRU. This tells us that for MT the performance of FIFO will be even better when compared to these approximations of LRU. Therefore, the only objection to FIFO is its anomaly in behavior. This behavior, however, was not observed in any of our benchmarks and cannot arise during the high-level memory operations we identify. Therefore, we conclude that FIFO should be used if it is found to be competitive.

Chapter 4 starts by presenting virtual memory performance numbers for Interlisp as reported in [8]. The comparison to MT is difficult because [8] presents raw global data which includes code faults. Our numbers refer strictly to heap and stack accesses. We proceed to present our benchmarks and present an argument which justifies our separation of concerns between the heap and stack. We empirically establish that for a global pool of frames for the stack and heap results in a worse effective access time than in MT. Surprisingly, there is very little in the literature on this matter beyond the initial studies done in the the late 1960s and early 1970s. Our study is a step forward in better comprehending the nature of the relationship of virtual memory and functional languages.

Having established that MT's separation of concerns is justified, Chapter 5 performs an in depth analysis of the MT heap. We present an argument in favor of FIFO over LRU for heap pages. The argument is based on how closely the performance of

FIFO follows that of LRU. In fact, there is a significant number of instances where FIFO simply does better than LRU. In closing, we also offer evidence that a prefetcher may play a roll in heap virtual memory performance.

Chapter six is an in-depth study of the MT stack. We prove that LRU is optimal for replacing the MT stack pages. We also empirically show that FIFO's performance is not as tightly coupled with LRU's as was with the MT heap. The MT stack page replacement algorithm is presented which mimics LRU but avoids all the overhead associated with LRU. In closing, evidence is presented that prefetching may also play a role for stack pages.

Our views on future work are presented in chapter seven. In addition, we outline a prefetching algorithm and a garbage collection algorithm for MT.

Chapter 2

List Layout and Virtual Memory

2.1 List Layout

Many pure functional languages depend on list structures for computation. The performance of these languages depends, in part, on how lists are stored in memory at runtime. For instance, virtual memory performance degrades as lists become too thinly spread throughout the virtual address space [11, 25, 56, 70]. The underlying reason is that the number of page faults caused by the lack of intra-list locality [1] is significantly higher than those caused by a list that is compactly held in memory. It follows that designing a system which fosters intra-list locality is of primary importance.

Several schemes have been proposed to increase intra-list locality and virtual memory performance. These proposals can be divided into two camps: those that suggest that the garbage collector should be responsible for building locality into list-based

[1] This term refers to a list that is both compactly held in memory and has a majority of its pointers referring to an object on the same page.

objects, and those that suggest using one of many list-encoding algorithms to build intra-list locality.

2.1.1 Garbage Collection as a Locality Builder

Garbage collection (GC) can build intra-list locality and it is primarily for this reason and not for the illusion of infinite memory that GC is needed in modern functional languages [11, 25, 56, 70]. In order to build locality a copying collector must be used. These collectors, however, may cause unacceptably long interruptions of the evaluator and may cause the machine to thrash if lists are spread-out over many pages [56, 70]. Furthermore, the use of a collector may be too little too late. The evaluator must wait until the collector is done before the benefits of its locality building efforts can be used. To overcome these problems GCs that work in phases have been developed; examples are incremental and generational collectors [5, 56, 64, 70]. These collectors improve performance, but still interfere with the evaluator.

Generational collectors, for example, can be used to recycle cache memory without recycling all of RAM as in SML-NJ [3]. In this manner, the memory requests made by the evaluator are mostly to the cache. The technique is based on the observation that most objects do not survive the first couple of collections. Keeping memory requests to cache held addresses is a method of fostering locality, but the garbage collector still must suspend the operation of the evaluator. On sequential machines caches are relatively small in relation to main memory and, therefore, interruptions

by the collector will be short (if not infrequent). We point out, however, that functional languages are still regarded as slow, despite years of work to improve garbage collecting.

2.1.2 Building Intra-List Locality During Evaluation

Given that frequent collections are undesirable, it is preferable to build locality into lists when they are created [11]. Building locality into lists without the interference of a garbage collector has been studied in the past [10, 11, 16, 17, 34, 51]. The earliest study by Borbrow and Murphy [11] suggests linearizing lists by altering **cons**. In their system a new list element is placed preferentially on the same page where the **cdr** or the **car** resides. The authors reported that this scheme minimizes cross page references, and limit structures to a few pages (i.e. created intra-list locality). This was accomplished by keeping a free-list for each page, which is likely to be very expensive.

More recent techniques compact lists by cdr-coding them [10, 16, 17, 34] in order to enhance VM performance. This coding strategy exploits the observation that most of the time the **cdr** is a pointer to the rest of the list. The idea is to eliminate the need to allocate space for the cdr-pointer by encoding in the **car** where the cdr-value is located. For example, the MIT Lisp machine used two bits to encode information about the cdr,[2] eliminating the need to always allocate space for the cdr pointer. The

[2]00 if the **cdr** was **nil**, 01 if the **car** of the **cdr** was in the next word, 10 if the next word contained

cdr-coding scheme has two effects: i) it reduces the amount of storage needed for a list by up to 50% and ii) it builds intra-list locality. It is clear that this scheme will enhance VM performance when lists are created one at a time.

If two or more lists are created simultaneously (so-called "parallel cons-ing"), then the performance of cdr-coding methods is significantly degraded [51]. For example, under these conditions the coding used in the MIT Lisp machine produces no compaction. The resulting lists can always be cdr-coded by a garbage collector, but this obviously results in an enormous increase in overhead. In order to solve this shortcoming, Li and Hudak [51] suggest a compaction algorithm that provides each list with its own sub-heap. Their idea is to allocate a block of cells when a list is created, instead of just allocating one cons-cell. In this manner, future memory requests in the construction of a list can be satisfied from the block of cells previously allocated. Simulations demonstrate that this technique compacts lists as well as or better than the scheme used by the MIT Lisp machine.

The coding schemes described above can achieve a significant compaction when the list is a *simple* list. In a simple list the **car** is always an atom and the **cdr** is **nil** or points to the rest of the list. There is, however, an overhead associated with these coding schemes. The operations of **cons**, **car**, and **cdr** are now more complex than in a system that does not employ encoding. For example, in Li and Hudak's

the value of the **cdr**, and **11** if the next word contained a pointer to the **cdr**.

method **cons** must execute at least two comparisons before deciding how allocation is to occur. In a system that does not use encoding there is no need for comparisons. Similarly, **car** and **cdr** are augmented by at least one comparison (sometimes two). To the knowledge of the authors there is no empirical evidence that establishes that this overhead is justified.

2.2 Heap Allocation in MT

The MT system enhances a small Lisp-like language with an all-software distributed virtual memory fine-tuned to it. The current implementation serves as a test-bed for different techniques utilized to improve virtual memory performance. In this section we will first describe the MT components that affect allocation and compare them to previous implementations. Then we will describe three simple experiments designed to demonstrate the improvements achieved by MT's allocation algorithm. Appendix A contains implementation details of MT.

2.2.1 Data Representation in Lisp

In many Lisp implementations all data objects are heap allocated [3] [24, 21, 27, 36, 37, 64]. An object that is heap-allocated and is in use by the program (i.e. live) has one or more references (i.e. pointers) to it. These references are utilized anywhere the

[3] The heap is the part of memory that is dynamically allocated as needed by the program. In order to give the illusion of an infinite memory this space must be garbage collected to recycle memory.

object is needed. For example, references are used to pass arguments to a function and to build bigger objects (e.g. a list structure).

The advantage of representing objects using references or pointers is that all objects are generic. The size of the object, its content or value, and its type are all abstracted away. All objects, regardless of their type, are manipulated the same way. Furthermore, only one copy of any object is necessary since a reference to the object can be used everywhere the object is needed. The fact that every object is unique allows the testing for equality of two objects to be accomplished by testing for pointer equality.

A downside to this approach is that it generates a lot of memory traffic. In order to reference a value one must chase a pointer. This is inefficient. For example, every time two integers of small magnitude are to be added or compared two pointers must be chased. In order to reduce memory traffic and improve space utilization the space used for a reference can be used to hold *self-contained* data [64]. The data that is self-contained can be encoded or immediate and is identified by its type. Data can be self-contained when the size of a pointer is large enough to hold both the tag and the datum itself. This approach has been successfully used in the representation of characters and *fixnums*. For example, MacLisp used self-contained data to optimize arithmetic operations [27, 46].

The use of self-contained data can easily lead to multiple copies of objects existing

at the same time. This leads to the breakdown of the classical concept of equality. It is no longer the case that it suffices to compare pointers since, for example, two different copies of 31 are considered equal but do not share the same space. Furthermore, the use of self-contained data may undermine portability of Lisp code from one machine to another. For example, the size of the set of *fixnums* may differ from architecture to architecture. Thus, on some architectures (eq 2^{25} 2^{25}) may be true while on others false [4]. This loss of referential transparency is unfortunate since one of the properties of Lisp and functional programming in general is that the architecture (or implementation) should be abstracted away as much as possible. The problem is further complicated if, for example, *bignums* are supported. These numbers are not limited in magnitude by the underlying word-size of the architecture. In this case, (eq (factorial 100) (factorial 100)) will always be false[5] [64]. Some Lisp dialects, like Common Lisp [44], allow multiple copies of objects to co-exist because of "tremendous performance improvements" for common operations. In order to regain some abstraction the predicate eql has been introduced. Thus, eq determines if two objects refer to the same object (i.e. memory location) while eql determines if two objects have the same value. That is, eq determines if two references point to the same object while eql determines if two objects are equivalent. The relationship between the two

[4]For example, in the MIT CADR machine fixnums are 24-bits while in Spice Lisp they are 27 bits [27]. Therefore, the result of eq will vary.

[5]The example here presupposes the use of an interpreter or a compiler that does not optimize the code by eliminating one of the calls to factorial

can be made clearer by observing that eq is sufficient for eql and that eql is necessary (not sufficient) for eq.

2.2.2 Data Representation in MT

All objects in MT are represented with 72 bits: an 8-bit tag and a 64-bit data field. That is, unlike most Lisp systems MT uses tagged data instead of tagged references. The object dimensions were based on the fact that Occam (the language MT is implemented in) has bytes and 64-bit integers as primitive types. This design allows the system to exploit the primitive Occam operators associated with these types.

Unlike its Lisp ancestors, all objects in MT are self-contained data or "unboxed". That is, the data field of an object always contains an integer, an index into the symbol table, a Boolean value, **nil**, or a **cons**-cell (i.e. the actual **car** and **cdr** 32-bit pointers). Pointers are only used within linked objects. Wherever an object is needed a self-contained datum is utilized. For example, the MT data stack is not a stack of pointers to objects in the heap. Instead, it is a stack of values. Thus, when an argument is pushed onto the stack it is an actual value that is being pushed (i.e. an integer, an index into the symbol table, **nil**, or a Boolean value). In the case of a **cons**-cell two pointers are pushed: one for the **car** and one for the **cdr**. Since all objects in MT have the same structure, pushing a **cons**-cell onto the stack takes only one push instruction as an integer. Executing (**cons** a b) requires heap-allocating a and b and returning on the stack a **cons**-cell whose **car** points to a and whose **cdr**

points to b.

The decision to make every object self-contained was based on the observation that the use of references forces objects to be prematurely allocated in the heap, as we shall show in the next section. The objects referred to are usually stored far away from their references. Such a situation does not promote intra-list locality which is clearly essential for good virtual memory performance. Our initial observations were confirmed by the experiments we describe.

In the remainder of this chapter we describe experiments that were designed to shed light on the heap layout of functional languages. Based on the insights gained from knowing how lists are stored in memory by the MT allocator we may then be able to make predictions about other more interesting programs.

2.3 Experiment I: List of Fibonacci Numbers Using References to Objects

Experiments I and II[6] used as a test bed the construction and traversal of the first fifteen Fibonacci Numbers. The naive algorithm for computing Fibonacci numbers was used (see 2.1). This algorithm was chosen for two reasons. The first is that it would force a great deal more heap allocation under the classical allocator than other list-building programs such as the construction and traversal of the list containing the

[6]For these experiments we used a page size of 512 S-expressions and the garbage collector was disabled. The number of 512 S-expressions was chosen because it results in a page size of approximately 4K bytes which is typical of current computer systems.

```
(define fiblist (n)
  (if (= n 0)
  '()
  (cons (fib n) (fiblist (- n 1)))))
(define fib (n)
  (if (< n 3)
   1
   (+ (fib (- n 1)) (fib (- n 2)))))
```

Figure 2.1: Code used for Experiments I and II

first n numbers, thus providing real insight into the severity of the intra-list locality problem. The second was that the algorithm is easy to understand, facilitating the explanation of how memory was utilized.

The list is constructed by prepending to **nil** the Fibonacci Numbers in descending order. All n numbers are computed before cons-ing them takes place. The needed results are kept in the stack. Clearly, the process produces a simple list as defined above (see 2.1.2). The layout of the list is exhibited in Figure 2.2 The memory space between elements of the list is garbage. That is, it is memory that was dynamically allocated during the computation of a Fibonacci number for storage of intermediate results. All this garbage is intermingled with live data because results to be held in the list are heap-allocated before the list is ready to be constructed. For example, the fifteenth Fibonacci number is heap allocated before the fourteenth Fibonacci number is computed. Thus, the memory between these two adjacent elements was used to compute the fourteenth member of the list. A similar situation holds for the other

adjacent members of the list: they are separated by memory used during computation. The only reason that there is more than one element on a single page is because the computation of small Fibonacci numbers can fit on a page of size 512. Clearly, this is not true of all computations.

A total of 24,794 heap cells were allocated requiring forty-nine pages. The list is thinly spread throughout the virtual address space on nine pages despite the fact that it can easily be allocated on one page. Of the nine pages that contain the list seven contain a single element, one page contains two elements, and one page holds six elements and the list's backbone. This means that 60% of the **car** pointers in the referred list are to a different page, meaning that this percentage of elements can cause a fault at traversal. One can observe that if the length of the list were increased by n then the number of pages that contain one element would increase by n, because each of the new elements would be held on a page where they are the only live-data. Thus, increasing the percentage of pointers to different pages and the potential for faults at traversal time.

The massive amount of heap allocation evidenced by this experiment suggests that a substantial amount of virtual memory traffic will be generated by a classical allocator. Since there is no effort to separate garbage from data or to minimize allocation one can expect many heap pages to be brought in for allocation and many to be swapped out to make room for new pages. Much of this work is done to manage

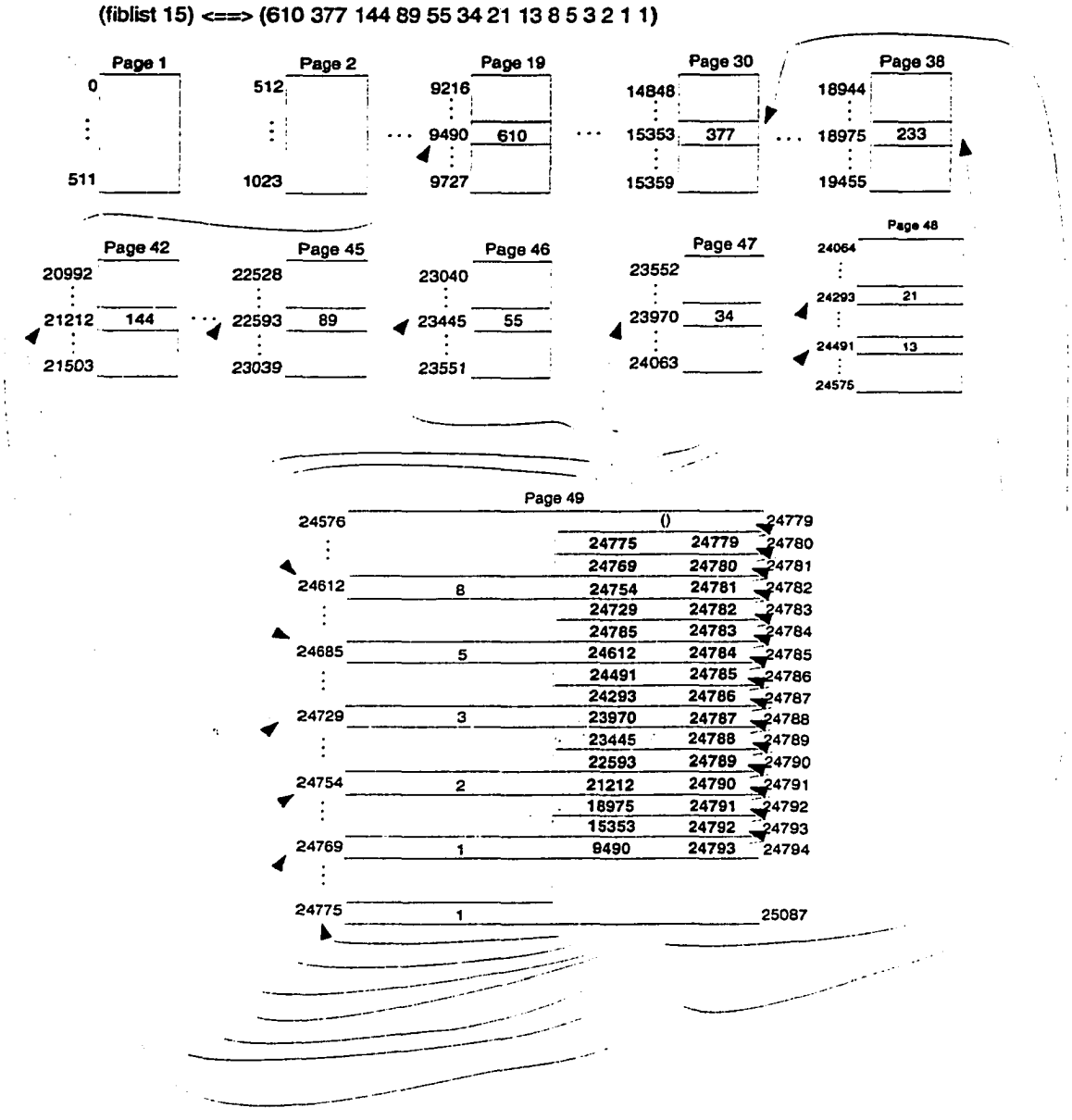

Figure 2.2: The list of the first 15 Fibonacci Numbers under classical heap allocation

garbage. The situation is not unlike that of a system that employs a generational collector. In order to minimize virtual memory traffic pages are recycled in RAM. This means that one delay is chosen over another. We show in our next experiment that by managing live data differently massive allocation can be avoided and virtual memory traffic minimized without calling on a collector.

The above observations make a compelling case for not allocating list members until the list is ready to be built. By holding list elements on the stack the intermingling of garbage and live data is avoided for atomic data (i.e. integer and symbols). So, for simple lists we expect to see no garbage intermingled with data.

2.4 Experiment II: Simple Lists in MT

Our second experiment used the same code as Experiment I, but used the MT allocation algorithm. All objects were self-contained as described in section 2.2.2. The allocation algorithm was designed to enhance intra-list locality, to reduce the amount garbage intermingled with live data, and to restrict the heap to list data. We first will outline the MT allocation algorithm and then proceed to describe the results of the experiment.

2.4.1 The MT Allocation Algorithm

The main characteristics of the MT allocation algorithm are:

1. During list construction heap space to execute a **cons** is not allocated until both arguments to **cons** are evaluated.

2. Values returned by function application are returned on the top of the stack. No heap allocation takes place for these objects. In the case when a list is returned, the whole list is heap-allocated except for the first cons-cell of the list that is returned on the top of the stack.

3. The stack is not heap-allocated. This means that stack space is automatically recycled without garbage collecting

4. Arithmetic is done on the stack. That is, no heap space is allocated for arguments or results.

5. Heap cells are allocated linearly from the first to the last.

6. Only S-expressions (i.e. objects) that are part of a list are heap-allocated.

Delaying the allocation of heap space until the arguments to **cons** are evaluated guarantees that there will be no garbage between adjacent simple-list elements. This follows from observing that any heap cells that are allocated for the computation of the car and cdr objects will not be intermingled between them when they are allocated (See Theorem 2.1 later in this section).

Returning values from function applications on the stack means that objects will not be prematurely allocated in the heap. This follows from observing that the needed

value is stored on the stack and not in the heap until the cons-ing takes place. All the garbage that may be generated by the computation of each list element does not separate adjacent list members since they are not stored in the heap. Allocation takes place when the cons-ing begins and all that is garbage is already heap allocated. This means that list elements are not allocated before the garbage is out of the way. Furthermore, if returned objects are not to be part of a list-based object MT will never allocate heap space for them. The savings in heap allocation are furthered by doing all arithmetic on the stack. This also speeds up these operations by eliminating the pointer dereferencing needed by some systems.

2.4.2 Results of Experiment II

Based on the MT allocation algorithm we expect simple lists to be compactly allocated in a continuous section of the heap. To test this hypothesis we ran the same program from Experiment I to compute the list of the first n Fibonacci Numbers. The results are in shown in Figure 2.3[7].

As expected under the MT allocation algorithm, the list was continuously allocated on one page. In fact, the computation did not even allocate one full page of heap space. There is no garbage intermingled with the list elements and all pointers (100%) are to the same page. In general for simple-lists that span more than one

[7]The reason for changing the length of the list was solely due to aesthetic and graphic creation concerns

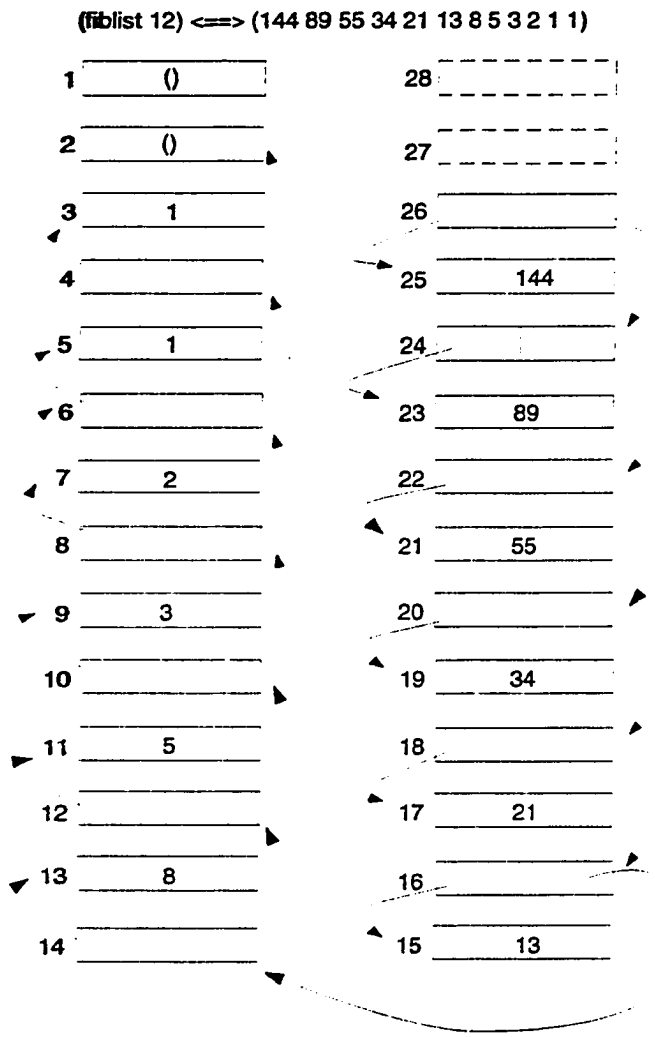

Figure 2.3: The list of the first 12 Fibonacci numbers under the MT Allocation Algorithm.

page, it will be the case that at most two pointers are not to the same page.

A closer look at the layout suggests that simple-lists are linearized under the MT allocation algorithm. The proof follows from an induction on the number of **cons**-cells in the list. Before proceeding to the proof, we first summarize the main ideas of the results. Without loss of generality assume heap cells are allocated from low addresses to high addresses. A cons-cell, L, with address l has its car pointing to at the heap cell at $l-1$ and its **cdr** to the heap cell $l-2$. In general, ($\mathbf{cdr}^n$ L) is $l-2\cdot n$ and the ($\mathbf{car}^n$ L) is $l-2\cdot n-1$. The arguments and results assume that the list is not created in parallel with other lists as in the case of the list of Fibonacci Numbers.

2.4.3 Layout Properties for Simple-Lists under MT Allocation

Theorem 2.1. *In the absence of parallel cons-ing and under the MT allocation algorithm simple lists are linearized as follows: The* **car** *and the* **cdr** *of a heap-allocated* **cons**-*cell with address l point to cells at $l-1$ and $l-2$.*

Proof: The proof follows by induction on the number, n, of heap allocated **cons**-cells that are heap allocated. The argument presented is valid for both recursive and tail-recursive constructions of simple-lists. The base is established for $n=1$ by observing the process by which a **cons**-cell is allocated. Recall that according to the MT allocation algorithm **cons** instructions are delayed until both arguments are evaluated and are held on the top of the stack. Also observe that during the construction of a simple list a **cons**-cell is not heap allocated until the second **cons**

```
(define fiblist (n l1 l2...lk)
  (cond ((test1) (recursive call with cons of an atom to l1))
    ((test2) (recursive call with cons of an atom to l2))
    .
    .
    .
    (else (return one of l1...lk))))
```

Figure 2.4: Pseudo-code for the tail-recursive construction of k lists

instruction is executed. When the first **cons** is executed its two arguments are heap-allocated and the resulting **cons**-cell is returned on the top of the stack for a recursive process and as an argument for a tail-recursive process. Notice that the arguments to the **cons** will be allocated contiguously; allocating the **cdr** argument first and then the **car**. Now the second **cons** can be executed and it will allocate the stack-resident **cons**-cell as its **cdr** value. Since no other heap allocation (i.e. no parallel cons-ing) has occurred, the **car** and the **cdr** of this allocated **cons**-cell, at address l, point to $l-1$ and $l-2$ respectively.

If we assume that the theorem holds for the first k allocated **cons**-cell it will suffice to show that it holds for the $(k+1)^{st}$ heap allocated **cons**-cell. When the $(k+2)^{nd}$ **cons** is executed its **cdr** argument is the $(k+1)^{st}$ **cons**-cell pointing to the sub-list containing k allocated **cons**-cell for which the theorem holds. Since no heap allocation takes place between one **cons** and another and **cdrs** are allocated before **cars**, the allocation of the $k+1$ **cons**-cell at location l will have the **car** and the **cdr** point to locations $l-1$ and $l-2$ respectively. **Q.E.D**

Parallel **cons**-ing, however, may yield non-continuous lists. Consider how heap cells are allocated by the process generated by the pseudo-code displayed in Figure 2.4. In this process, heap cells belonging to each of the k lists will be intermingled with each other. Furthermore, if any of the Boolean tests of the **cond** require heap allocation or any of the k lists becomes garbage then the resulting lists will be intermingled with garbage. However, if the allocation is restricted to the construction of the lists, all the lists constructed are returned, and the cons-ing is randomly distributed among the lists under construction we have the following theorem.

Theorem 2.2. *The expected distance between* **cons**-*cells of the same list when allocation only takes place for k simple-lists created in parallel is $2 \cdot k$.*

Proof: The probability to **cons** an atom to list i is $\frac{1}{k}$ every time the function building the lists in parallel is called. On average, each list will be augmented with an element before any list is repeated. This implies that we can expect list i to gain an element every k calls. Since every execution of **cons** allocates 2 heap cells, we expect to have $2 \cdot k$ heap cells allocated between adjacent **cons** cells. **Q.E.D**

The above theorem suggests that parallel construction of lists also creates linearized structures. However, the situation here exhibits a more general form of linearization than exposed before. When a simple list is created in the absence of parallel cons-ing the distance between **cons** cells is two. This is clearly because only one list is being created. In the case of parallel list construction the linearization occurs in memory blocks of roughly size k for each list.

The layout of simple lists as described above can be exploited for effective list compaction. This type of layout is achieved without modifying **cons**, **car**, and **cdr**. When such a layout exists some of the systems described above change the way these list primitives are implemented to achieve compaction. The exploration of whether it is efficient to create a coding scheme for simple lists in MT is left for future work. It is unclear that making the list primitives more complex will make Lisp faster despite compaction. The increase in complexity may not be justified if the gains in virtual memory performance are small.

2.5 Experiment III: A List of Simple Lists

The goal of this experiment was to gain insight onto how the properties of simple lists may be inherited by more complex lists. The idea we pursued was simple: determine the layout of a list of simple lists in memory under the MT allocation algorithm. We expected to find a structure that is linearized and allocated continuously in memory. The experiment evaluated the expression that constructed a list from cons-ing together three copies of the list of the first five Fibonacci numbers. The results are shown in Figure 2.5.

The three simple lists that make up this list are allocated exactly as described above. The first sub-list is held in locations 5–14, the second in locations 15–24, and the third in locations 25–34. In this case, however, the head of each list is not stored

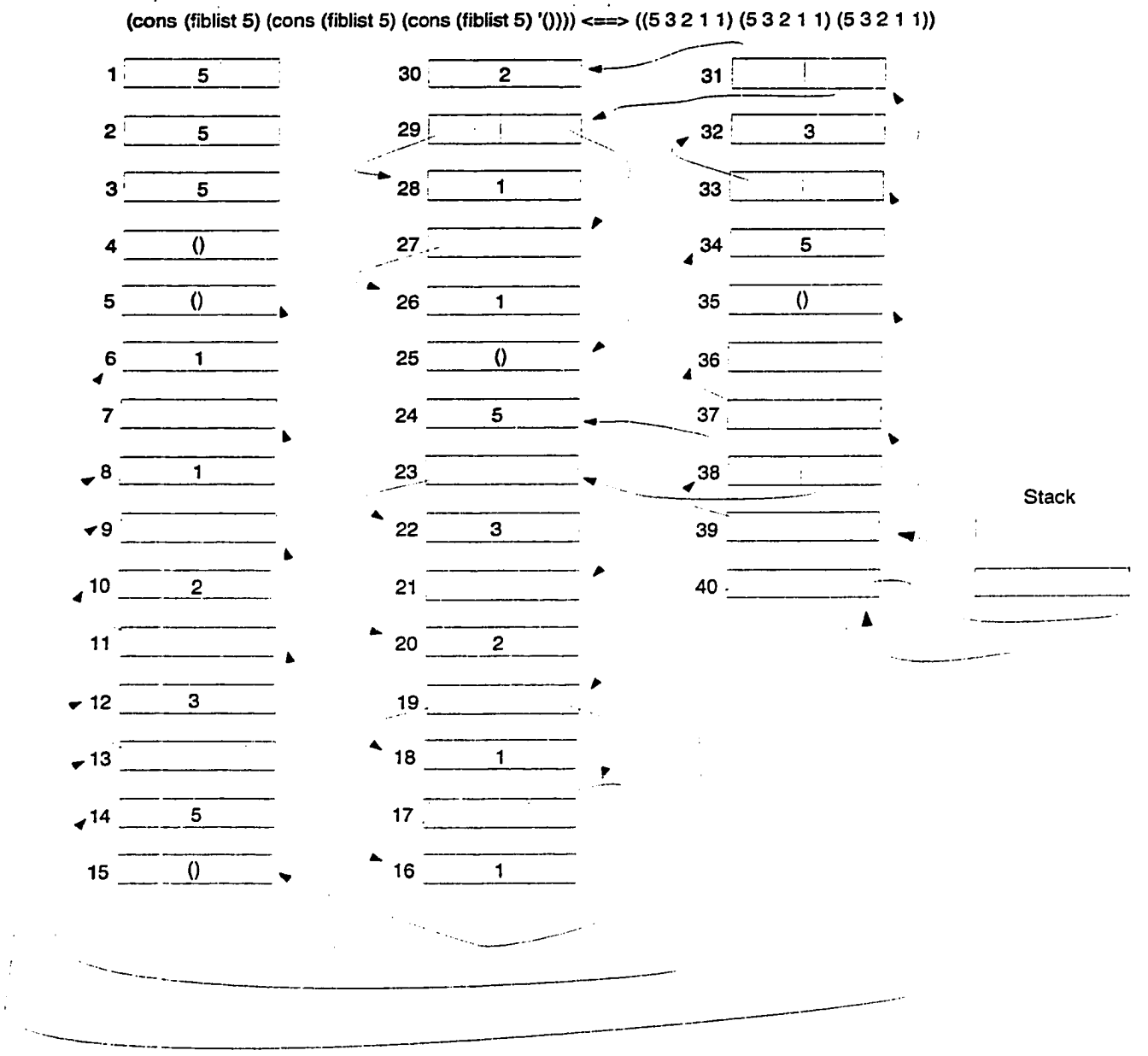

Figure 2.5: The list resulting from (**cons** (fiblist 5) (**cons** (fiblist 5) (**cons** (fiblist 5) '())))

with the rest of the list. Instead, they are stored with the backbone structure that glues the three simple lists together into a complex list. This backbone structure is stored in locations 35–40. The head of the overall list structure is returned on the stack and is not heap allocated.

The layout reflects the design choices made for MT. The head of each sub-list is not stored with the list itself, because the head is what is used to manipulate a list as an immediate value. This suggests that many copies of the head of a list may exist. This, however, does not introduce any ambiguity as to whether two lists are the same list. Testing two cons-cells for equality is all that is needed.

The layout also exhibits what we have coined *level-linearization.* That is, if you view a list-based structure as a tree you will find that the elements at each level of the tree are linearized as described in the previous section. The following theorem is suggested by the layout displayed by this experiment.

Theorem 2.3. *Complex lists in MT are level-linearized.*

Proof: The proof follows by induction on, n, the number of levels of the tree of the list-based structure. When the tree only has one level Theorem 2.1 establishes the linearity of the structure. Assuming that a k-level tree is level-linearized it will suffice to show that a $(k+1)$-level tree will be level-linearized. The $(k+1)^{st}$ level will be the backbone that glues together two or more list-structures with each having a height $\leq k$. Since the MT allocation algorithm does not allocate the arguments to a cons

until they are evaluated, it follows that the list-based structures to be glued together by the $(k+1)^{st}$ level will be allocated before the $(k+1)^{st}$ level is allocated. By the induction hypothesis these structures will be level-linearized. The $(k+1)^{st}$ level will be linearized since it is nothing more than the allocation of a simple list. The **cars** will point to the heads of each of the structures being glued together while the **cdrs** will point to the next **cons**-cell of the $k+1$ backbone. **Q.E.D.**

2.6 Summarizing Remarks

In this chapter we have demonstrated that the use of pointers to objects instead of the literal objects leads list elements to be prematurely allocated in the heap. This leads to lists being thinly spread throughout the virtual address space creating the potential of poor virtual memory performance due to lack of intra-list locality.

MT proposes the elimination of the use of pointers except where absolutely necessary as in lists. Values that can not be manipulated literally should be represented in a manner that allows them to be manipulated as literals. For example, pushing a list onto the stack is done by pushing a **cons**-cell. This is possible because all objects are represented the same way.

Using literal values allowed the development of a simple definition for the heap and the implementation of a simple allocation algorithm. The MT heap only contains list data and heap space is only allocated to hold list elements. Furthermore, allocation

for the arguments to a cons does not take place until both arguments are evaluated. This lead to the linearized layout of simple lists which contains a high degree of intra-list locality. This linearization property extended to complex lists by endowing them with level-linearization. That is, when a list-based structure is viewed as a tree each level is linearized. This result is suggested by realizing that under the MT allocation algorithm all lists at level k are simple lists. Thus, at each level of the structure there will be a high degree of intra-list locality.

Chapter 3

Virtual Memory Performance and Simple Lists

3.1 Virtual Memory

The term *virtual memory* (VM) is used to describe a two-level (or more) storage system that is used by most modern computers [1]. The secondary level or backing store provides the system with a large but slow memory level. This is traditionally implemented with a hard disk. The primary level is much smaller than the secondary level and traditionally is the computer's RAM. At any moment during the execution of a program a subset of the information in backing store is held in primary memory. The goal is to give the illusion that the computer has a large fast memory.

In order for a VM system to work efficiently the addresses generated by the computer should be found in primary memory as often as possible. This means that the stream of addresses must be known in advance or that the programs running on

[1] There are notable exceptions- the transputer, for example, has one-level memory [39]

the computer exhibit *locality of reference.* Locality of reference is the term used to describe the fact that the address stream of a program over a short period of time refers to a small subset of the virtual memory space [35]. Locality of reference is exhibited by most but not all programs (e.g. garbage collection). It has been observed that the subset of pages being accessed by a program changes slowly [20], and hence maintaining it in primary memory yields the intended illusion. The majority of references will be to primary memory with an occasional reference to an item that must be imported from backing store.

3.1.1 Virtual Memory Organization

Most VM systems are implemented by dividing primary memory into *frames* and secondary memory into *pages.* Frames and pages are fixed regions of primary and secondary memory respectively which are of exactly the same size. The pages currently in use by the program (i.e. in the time interval from t_i to t_{i+1}) are called the current working set and are kept in primary memory. The VM subsystem manages the flow of pages to and from primary memory trying to keep the pages that will not be accessed for the longest period of time in backing store.

As a computation progresses a reference to a page that is not in primary memory may occur, causing a *page fault.* When a page fault occurs the page that contains the address required by the program has to be imported from backing store. If primary memory is not saturated the page is brought into an empty frame. Otherwise, a page

must be selected for eviction. The policy by which a page is chosen for eviction is called the *page replacement policy.* The page replacement policy attempts to evict a page that will not be used in the future or that will not be used for the longest amount of time.

3.1.2 VM Performance

The above description correctly suggests that all memory accesses do not take the same amount time. Accesses that cause a page fault will take orders of magnitude more time than accesses that do not cause a page fault. For example, servicing an average page fault requires twenty-five milliseconds, while the main memory access time is only ten to two hundred nanoseconds depending on the system.

Since all memory requests do not take the same amount of time to be serviced, we measure *the effective access time*, that is the amortized time it takes to service a memory request. The effective access time clearly depends on how often a page faults occur. When a page is found in primary memory it is called a hit and, otherwise, it is called a miss. The probability of a hit (or hit rate) is defined as:

$$h = \frac{Accesses that do not cause a fault}{Total number of accesses}$$

while the miss rate is defined as $m = 1 - h$. We can define the effective access

time, *eft*, as follows:

$$eft = h * mat + m * pft,$$

where *mat* is the memory access time and *pft* is the page fault time. The goal is to make the *effective access time* as close as possible to the memory access time [65].

Clearly, the closer to 1 the hit rate is the closer the effective access time will be to the memory access time. The question now becomes how low can the hit rate become before performance is compromised too much. For example, CRL, an all-software distributed shared memory system developed at MIT [43], reports that its performance is within 12% of a hardware-supported shared memory. If we take 12% as an acceptable level of degradation we need:

$$112 * mat >= h * mat + m * pft$$

$$112 * mat >= h * mat + (1 - h) * pft$$

$$112 * mat - pft/mat - pft <= h$$

Therefore, if we take a memory access time of thirty-three nanoseconds and a

page fault time of twenty-five milliseconds we find that it must be the case that $h >= 0.99853$. Approximately, only one in every 700 accesses can cause a fault to keep the effective access time within twelve percent of the memory access time. Thus, the page replacement policy must keep in memory the needed addresses over ninety-nine percent of the time. Otherwise, execution time will be dominated by virtual memory management.

3.1.3 Page Replacement Policies

The commonly implemented page replacement policies are *first-in first-out* **FIFO** and *least recently used* **LRU** [35]. FIFO selects for eviction the page that has been resident in primary memory for the longest amount of time. The advantages of FIFO are that it is easily implemented and requires little overhead during program execution. LRU selects for eviction the page that has not been accessed in the longest amount of time. The advantage of LRU is that frequently used pages are not evicted just because they have been memory resident for a long time, which may happen under FIFO. LRU is more difficult to implement, however, since every memory access incurs the overhead of updating statistics on accessed pages. LRU is usually implemented with hardware assistance and is usually approximated to reduce the overhead associated with *pure* LRU. The performance of both FIFO and LRU are used in chapters 4, 5, and 6 to compute effective access times for MT and to argue for the separation of heap and stack spaces, for FIFO as the heap-page replacement algorithm, and for the MT

stack-page replacement algorithm as the best policy for stack pages.

As suggested above, the performance of page replacement policies is measured by computing the hit and miss ratios. Swapping out the page that will not be used for the longest period of time was demonstrated to be optimal [7]. Unfortunately, this policy requires knowing in advance the stream of addresses to be accessed. Therefore, we are forced to rely on approximation algorithms like LRU and FIFO.

To a system builder the question that must be answered is what page replacement algorithm is best for the system being built. Despite the popular belief that LRU is superior to FIFO there is, surprisingly, no proof of this. This belief stems from comments and/or statistics (derived from empirical evidence) that are not thoroughly explained. For example, Hayes states that *"the page hit ratio of LRU is quite close to that of OPT, a property that seems to be generally true"* after a small example where FIFO produces the worst hit ratio [35]. We find in the VM literature the same problem Wilson et. al. [71] find in the literature on memory allocation: paging is poorly understood. In fact, paging is much less understood than popularly believed. As recently as 1995 Wilson et. al. state that even locality, the basis of modern paging algorithms, is poorly understood [71]. A review of the literature reveals that little has changed since then. These observations extend to paging under functional programming languages. The Symbolics 3600, a machine with hardware support for Lisp, assumes LRU as the default page replacement policy [27]. There is, however, no

LRU Ordering of Pages in Memory	Sequence	FIFO Ordering of Pages in Memory
1 2 3 4 5	$P_1P_2P_3P_4P_5$	1 2 3 4 5
5 4 3 2 1	$P_4P_3P_2P_1$	1 2 3 4 5
4 3 2 1 6 *	P_6	2 3 4 5 6 *
3 2 1 6 5 *	P_5	2 3 4 5 6
2 1 6 5 4 *	P_4	2 3 4 5 6
1 6 5 4 3 *	P_3	2 3 4 5 6
6 5 4 3 2 *	P_2	2 3 4 5 6
5 4 3 2 1 *	P_1	3 4 5 6 1 *
5 4 3 2 1 ******	$P_6P_5P_4P_3P_2P_1$	4 5 6 1 2
Total faults: 12		Total Faults: 3

Table 3.1: FIFO beats LRU

evidence provided to justify this design choice. LRU outperforms FIFO for general access streams, but not for LISP. Our measurements on MT, presented in chapters 4 and 5, clearly show that it is not uncommon for FIFO to outperform LRU.

The difficulty with identifying the best page replacement policy (other than the optimal one) is that they all can beat each other by attacking the assumptions each makes. In the following examples, let P_i be one or more accesses to the i^{th} page, let n be the size of main memory in pages, and let s be an arbitrary positive integer.

Example 3.1.1 The access string $P_1P_2.....P_{n-1}.....P_1(P_{n+1}P_n.....P_1)^s$ causes more faults under LRU than it does under FIFO. This is established by the trace shown in Table 3.1 where each represents a page fault, $n = 5$, and $s = 2$. Without loss of generality we assume that the initial state (i.e. ordering) of the memory resident pages is 1 2 3 4 5. The ordering notation means that the left most listed page will be the next victim and the right most listed page is the youngest for the given

LRU Ordering of Pages in Memory	Sequence	FIFO Ordering of Pages in Memory
1 2 3 4 5	$P_1P_2P_3P_4P_5$	1 2 3 4 5
5 4 3 2 1	$P_4P_3P_2P_1$	1 2 3 4 5
4 3 2 1 6 *	P_6	2 3 4 5 6 *
4 3 2 6 1	P_1	3 4 5 6 1 *
4 3 6 1 2	P_4	3 4 5 6 1 *
4 6 1 2 3	P_3	4 5 6 1 2 *
6 1 2 3 4	P_2	5 6 1 2 3 *
1 2 3 4 5 *	P_1	1 2 3 4 5 *
4 3 2 1 6 *	$P_6P_5P_4P_3P_2P_1$	2 3 4 5 6 *
Total faults: 3		Total Faults: 7

Table 3.2: LRU beats FIFO

algorithms. If a page is not listed it means that it is not memory resident. The table clearly establishes that LRU will fault every time a different page is accessed within the repeated sequence. That is, the number of page faults under LRU is O(ns). In contrast, FIFO faults twice the first time through the repeated sequence and only once every time through the repeated sequence afterwards. Thus, in this case the number of page faults under FIFO is O(s).

Example 3.1.2 Similarly, there exist access streams for which FIFO will cause more faults than LRU. Such an access string is $(P_1P_2P_3P_4P_5P_4P_3P_2P_1P_6)^s$. A trace of page faults for this access string is exhibited in table Table 3.2 for $n = 5$ and $s = 2$. As in the previous example we assume that the starting state of memory resident pages is 1 2 3 4 5. The trace demonstrates that LRU performs better than FIFO. As s goes to infinity LRU will fault twice after every pass through the string while FIFO will fault six times. In this example, it is FIFO that causes the number of faults to

be proportional to the number of different pages accessed.

The above examples clearly establish that both LRU and FIFO can perform better than the other. They further establish, however, that both can be a poor choice of replacement algorithm. This is particularly true when the number of page faults is proportional to the number of pages accessed. One would expect a random replacement algorithm to be a better choice in this case. Interestingly enough, both of these algorithms can be optimal for some access strings. For instance, FIFO is optimal for the example used in Table 3.1. We believe this establishes that the choice of a page replacement algorithm is a critical design issue. Whenever possible, the decision should be guided by the domain of programs that are expected to be evaluated. In general, this may not be possible but we believe that it may possible for specific domains. In particular, list-processing languages may provide a domain where one is able to determine the best on-line page replacement algorithm through empirical research and thorough explanations for the observed results.

Before proceeding with the questions of which page replacement algorithm, LRU or FIFO, is better for MT and which allocation algorithm, classical or MT, should be used, the question as to whether there exist for LISP-like languages programs that can defeat LRU and FIFO arises. More generally, can LISP code be generated to defeat any on-line page replacement algorithm? After a little thought, it should be clear that Lisp can defeat any on-line page replacement algorithm by strategic use of

conditional statements and knowledge of system parameters such as page size, main memory size, and the allocation algorithm. The coder just needs to create a number of lists (i.e. the number of pages in main memory + 1) such that each list is stored on a different page and then access each one in the desired order using **cond**.

3.1.4 Empirical Data

There is no proof that LRU is generally superior to FIFO or vice versa. In fact, Belady's seminal study as well as studies done by Denning showed no clear winner among several page replacement policies [7, 20]. There is, however, a popular belief that LRU is the best choice for a on-line page replacement algorithm. This belief is in all likelihood based on empirical studies that have suggested that LRU is superior and the fact that FIFO suffers of Belady's anomaly [7, 65].

Many studies have focused on presenting measurements obtained from different systems, different programs, and/or different system parameters. There is a need in the literature to explain why, for example, the performance of LRU is superior or why FIFO performs as well as LRU or very poorly against LRU. Baer states that no analytical result is known besides the optimality of OPT but that very strong experimental evidence shows that on the average for a program $A\ FIFO_{faults} > LRU_{faults} > OPT_{faults}$ [4]. This seems to be consistent with others that have concluded that FIFO is less effective than LRU [68, 15].

Margolin et. al. [68] conducted paging experiments on an IBM 360/40. They were interested in determining if the page fault count was a good metric for comparing page replacement algorithms. In their study they point out that it is impossible to determine if a high fault count is due to a heavy workload or a poor page replacement algorithm. Bearing this in mind they propose the activity count (AC), which is the percentage of core resident pages accessed between page faults, as a metric (which we use in chapters 5 and 6 to determine if a prefetcher can improve VM performance in MT). The page replacement algorithms studied were **LRU**, **FIFO**, and **Random**. They used as test programs the compilation of a small, a medium, and a large FORTRAN program. The page size was of 4096 bytes and the size of main memory was varied from 24 to 20 to 16 pages. Among some of their conclusions are that the AC is a better metric than the page fault count and that LRU performed better than both **FIFO** and **Random**. Also noteworthy is the fact that **FIFO** performed better than **random**.

Given the lack of empirical data reported in the literature and the significant contradictions in the record, a true sense of the real difference in performance between LRU, OPT, and FIFO can not be obtained from a literature survey. One of the earliest and few empirical studies found that LRU yields a performance within thirty percent (30%) to forty percent (40%) of OPT [18]. This statement directly contradicts the assertion quoted above from Hayes. One can hardly consider a 30%-40% difference

0.53	0.40	0.47	0.52	0.60	0.38	0.44	0.52	0.26
0.89	0.65	0.66	0.46	0.62	0.36	0.70	0.81	0.26
0.64	0.78	0.60	0.75	0.40	0.35	0.67	0.76	0.33

Table 3.3: Relative Difference of FIFO and LRU Page Faults Based on Margolin et. al.'s Reported Data

to be quite close. This seems to be representative of the state of empirical work in the field and, therefore, there is no clear conclusion to be drawn from the numbers reported.

The study conducted by Margolin et. al. [68] presents much more of its data. They do not report page fault rates but do present the number of page faults caused by **LRU**, **FIFO**, and **Random**. Based on their data we can compute the relative difference between the number of **FIFO** and **LRU** faults [2]. The relative difference in page faults for their twenty-seven FIFO-LRU experiments are displayed in Table 3.3. The table reveals that **FIFO** causes anywhere between twenty-six percent (26%) and eighty-nine percent (89%) more faults than **LRU** for the programs in their study. The average of the table is around fifty-five percent (55%). If the programs in their study are representative, we can expect the number of page faults and the page fault rate produced by **FIFO** to be substantially higher than those produced by **LRU**. Indeed, a fifty-five percent difference in performance on average would make **FIFO** a poor choice for any system at all with the exception of those that are expected only to run programs that produce a small number of faults.

[2] That is, $(FIFO_{Faults} - LRU_{Faults})/LRU_{Faults}$

It is unlikely that in the near future a proof will be forthcoming that establishes which is the best page replacement policy for functional languages if one exists. Nonetheless, we are still faced with the need to choose the paging policy for MT. We know that the decision will be influenced by the allocation algorithm that MT uses. So, before choosing a page replacement policy, it is necessary to choose an allocation algorithm and to understand the consequences of this choice on how lists are laid out in memory. In the remainder of this chapter, we will make the case for the MT allocation algorithm and establish some properties of how list-based structures are laid in memory that affect VM performance via a close study of Insertion-sort.

3.2 Case Study: Insertion-sort

We now turn to the task of determining if the MT allocation algorithm makes a difference in VM performance. It certainly would make no sense to advocate such a radical departure from classical implementations unless the benefits were empirically confirmed. Our goal in this section is to illustrate the benefits gained from the MT allocation algorithm. The aim is not to simply present our empirical results in the ways of numbers, graphs and charts, but also to precisely explain why the observed behavior occurs and to extrapolate observations that may help in explaining the behavior of more complex programs.

In order to gain insight into heap paging behavior we will use Insertion-sort as

```
(define insert (x L)
  (cond ((null? L) (list x))
    ((> x (car L)) (cons (car L) (insert x (cdr L))))
    (else (cons x L))))
(define in-sort (L)
  (if (null? L)
    '()
    (insert (car L) (in-sort (cdr L)))))
(define mklist (n)
  (if (= n0)
    '()
    (cons (random 1000000000) (mklist (-n1)))))
```

Figure 3.1: Definitions for Insertion-sort

a case study. The code used for each this study is displayed in Figure 3.1. This algorithm was chosen because it is generally familiar to a broad audience, easy to understand, and only requires the use of simple lists (as defined in Chapter 2). We shall first present the results obtained from both our classical Lisp simulator and our MT system. Then we will attempt to explain why the observed results occur. As we go along we will point out what we believe to be common heap reference operations and prove some properties about these common operations on simple-list and their influence on virtual memory performance.

3.2.1 The MT Allocator vs. a Classical Allocator

The experiments were conducted on a list of random numbers generated by **random**, MT's static random number generator [3]. Insertion-sort was applied to a list of 300

[3]The static random number generator always produces the same stream of numbers when the system is restarted

elements. In all experiments the heap was large enough to avoid a call to the garbage collector and each page could hold 512 MT objects. Each MT object is 72-bits long as described in Appendix A. Thus, each page can hold 4608 bytes. The decision to limit each page to 512 objects is based on a conclusion in [18] that states that virtual memory performance is substantially increased by increasing the number of pages in main memory instead of increasing the page size. Thus, page sizes of 1024, 2048, or 4096 were eliminated since we were seeking the best performance. The page size of 512 S-expressions was agreed upon since it is the smallest page size used by modern computers. In our case, a word has 72-bits which is the size of all MT objects.

Under the classical implementation, insertion-sort allocated 71,508 heap cells for a total of use of 140 heap pages. Under MT, 47,863 heap cells were allocated for a total use of 94 heap pages. Our classical simulator accessed the heap 713,443 times while the MT system accessed the heap 238,704.

For Insertion-sort we will present tables with the total number of page faults produced by FIFO and LRU under both our Classical Implementation and MT, with the respective fault rates, and with the relative difference between Classical and MT. The discussion for Insertion-sort will be followed by some results describing operations on lists that yield FIFO equivalent to LRU. Each table will start with a column labeled Main Memory Size which contains the percentage of the total number of pages used by the sample run represented by each row. Since MT allocates less heap space than

	Classical	Classical	MT	MT
Main Memory Size	FIFO	LRU	FIFO	LRU
10 (14,9)	0.0002131	0.0001906	0.0004189	0.0003979
20 (28,19)	0.0001724	0.0001626	0.0003351	0.0003267
30 (42,28)	0.0001444	0.0001388	0.0002848	0.0002806
40 (56,38)	0.0001233	0.0001191	0.0002429	0.0002387
50 (70,47)	0.0001009	0.0000981	0.0002010	0.0001968
60 (84,56)	0.0000813	0.0000785	0.0001633	0.0001591
70 (98,66)	0.0000617	0.0000589	0.0001214	0.0001173
80 (112,75)	0.0000420	0.0000392	0.0000795	0.0000795
90 (126,85)	0.0000210	0.0000196	0.0000377	0.0000377

Table 3.4: Fault Rates for Insertion-sort: FIFO vs LRU and Classic vs MT

the classical implementation (CI) it is important to realize that $n\%$ under the classical implementation is greater than $n\%$ under MT. That is, the numbers are presented in relation to the number of total pages used by each. For those wanting to compare numbers for equal sizes of main memory the size of memory in pages is provided in parenthesis. The first is the size for the classical implementation and the second for the MT allocation. In order to compare numbers with equal size memories, the columns must be aligned to roughly match the size in pages.

The page fault rates for insertion-sort are presented in Table 3.4. The table immediately reveals that insertion-sort under both the classical and the MT systems can yield a good page fault rate. Based on the example at the beginning of this chapter this occurs after 50% in the classical case and after 70% in the MT case. Furthermore, you can observe that the fault rate for MT is approximately twice as high (i.e. a relative difference >= 1.0) as the one for the classical system. This

	Classical	Classical	MT	MT
Main Memory Size	FIFO	LRU	FIFO	LRU
10 (14,9)	152	136	100	95
20 (28,19)	123	116	80	78
30 (42,28)	103	99	68	67
40 (56,38)	88	85	58	57
50 (70,47)	72	70	48	47
60 (84,56)	58	56	39	38
70 (98,66)	44	42	29	28
80 (112,75)	30	28	19	19
90 (126,85)	15	14	9	9

Table 3.5: Heap Page Faults for Insertion-sort: FIFO vs LRU and Classic vs MT

would seem to suggest that the classical implementation outperforms the MT system. This observation, however, is wrong. The problem is that fault rate alone can be a deceptive metric since the fault rate evenly distributes the cost of faults over all accesses. This occurs regardless of the amount of work being done. In other words, the fault rate abstracts away the details of how many faults and how many accesses really occur. This unjustly favors processes that execute more accesses to do the same work. The classical system accesses the heap approximately 50% more often than the MT system (a relative difference of 0.49). Therefore, the cost of a page fault in the classical system is lower per access than in MT even though MT incurs fewer faults as shown below.

The number of actual heap page faults incurred by insertion-sort under LRU and FIFO for both MT and CI can be found in Table 3.5. The table clearly establishes that the MT allocation algorithm yields better virtual memory performance than does

Main Memory Size	Classical	MT
10	0.12	0.05
20	0.06	0.03
30	0.04	0.01
40	0.04	0.02
50	0.03	0.02
60	0.04	0.03
70	0.05	0.04
80	0.07	0.00
90	0.07	0.00

Table 3.6: Relative Page Fault Difference between FIFO and LRU for Insertion-sort

CI. Unlike the page fault rates in Table 3.4 that suggested that FIFO in the classical system yields better performance than either page replacement algorithm under MT, we can observe that it is MT that yields the best performance. Fewer page faults, allocations, and accesses to the heap mean a more virtual memory friendly heap. It is safe to make this assumption since code accesses remain the same and the increase in stack accesses does not cause more faults [4].

The above discussion clearly establishes that as a metric, page fault rates (or hit rates) do not suffice to determine the effectiveness of a paging policy. The use of page fault rates must be accompanied by the use of non-relativistic measurements like fault counts, number of accesses, and number of allocations.

The relative difference between FIFO and LRU for Insertion-sort is presented in Table 3.6. The numbers in the table suggest that the difference in performance between FIFO and LRU is a lot tighter in MT than suggested by the numbers from

[4]See Chapter 6 for more on this topic

Table 3.3. In fact, the difference is less than 12%. These numbers establish that FIFO can be as good as LRU in practice. However, it could be that the results are the exception and not the norm. Perhaps the locality of insertion-sort is so great that there is little opportunity for the performance of FIFO and LRU to diverge. The numbers for the CI suggest that the results are a property of insertion-sort since the difference with MT is small. If the results were not partly due to the locality inherent in insertion-sort the difference between MT and CI would be larger. The difference then would be due to differences in allocation. Nonetheless, it may be the case that the apparent equivalence of FIFO and LRU is due to a property of Lisp. To determine the answers to these questions it is necessary to understand how the heap is being accessed by the insertion-sort program. We shall see that insertion-sort performs several memory operations that we consider common to most list programs and that these memory operations leave little room for FIFO and LRU to diverge.

3.2.2 Understanding Insertion-sort

In order to understand the paging behavior of insertion-sort we will describe what is occurring at the memory level during program evaluation. We will trace the actions generated by the code displayed in Figure 3.1. Our attention will focus on heap accesses, but actions taking place on the stack will be described in order to make the description clear. Descriptions of all other accesses (e.g. code) will be omitted since they do not affect the performance of the virtual memory that implements the

heap in MT. The reader is reminded that the MT evaluator employs applicative-order evaluation, and as a consequence, all arguments must be evaluated before a function can be applied.

The heap access pattern can be summarized by observing what the code is doing. First, all the list elements are pushed on the stack and then cons-ed together to form the list. The consing of the of the elements is a linear traversal of unallocated heap pages. Then starting from the empty list as the sorted list so far an element is inserted into the list one at a time. This means traversing the sorted list so far and then creating a new list of sorted elements containing the newly added element. This repetitive cycle of list creation followed by list traversal yields $FIFO \approx LRU$.

Evaluation would start with an expression like:

```
(in-sort (mklist n))
```

where (mklist n) is evaluated before the evaluation of in-sort commences. The function mklist pushes n numbers on the stack before executing a single cons. When n reaches zero all the elements of the list have been stacked and the delayed conses are executed. For each cons executed, two heap cells are allocated and a cons-cell is pushed onto the stack. When mklist concludes, the list of random numbers is contained in the heap and the head of the list is on the stack. This process is simply

the one described in Chapter 2 as the construction of simple lists. For this part of the evaluation the performance of FIFO equals that of LRU (see Theorem 3.1).

After the list is constructed, in-sort traverses the list and pushes all its elements on the stack. Traversing a simple list is simply a sequential traversal of the pages that hold the list given the way simple lists are linearized under MT. This means that the performance of FIFO equals that of LRU. Finally, a series of traverse-create operations take place to construct the sorted list by taking one element from the stack at a time. The traverse-create operation first traverses a list (in this case up to the point the element is to be inserted) and then creates a new front of list from the element being inserted and the elements smaller than it. The former is a linear traversal of allocated pages and the latter is a traversal of unallocated pages For every element, a_i, on the stack we can expect to traverse half of the sorted list in order to add a_i to the sorted set. These pages, with the possible exception of the last one, immediately become garbage. That is, if the insertion of a_i traverses pages P_f to P_l all pages but P_l become garbage (if all the elements in P_l are traversed then P_l itself also becomes garbage). After inserting a_i, the pages in the front of the list (i.e. the pages containing the elements just traversed) will have the same ordering under FIFO and LRU. Recall that heap space is allocated for a list after popping the stack. Creating the new front of list requires the use of previously unallocated heap pages which are linearly traversed. Under MT, the front of the list will be stored

contiguously in memory. This means that for the next traversal of the front of the list the virtual memory ordering of pages will be the same for both page replacement algorithms. The creation followed by a traversal theorem 3.2 below establishes this fact.

As the sorted lists grows, we expect the gap between LRU and FIFO to be tighter since more elements are being traversed with roughly the same performance. If the list is short then it can be completely sorted in memory, and we have that, performance-wise, $FIFO = LRU$. If the list is very large (much larger than main memory) then the middle of the list will be in backing store most of the time. Traversing this list for the insertion point of the new element is simply traversing a series of pages in the reverse order of creation forcing both replacement policies to fault for the same pages held in backing store. The creation of the new front of list only requires popping the stack and traversing unallocated heap pages. Thus we have once again that performance-wise $FIFO = LRU$. Recall that the pages traversed immediately become garbage so all core resident pages become garbage if the list up to the middle element does not completely fit in memory. So, where does the small difference in performance come from?

The order in which garbage pages are evicted makes no difference as far as virtual memory performance is concerned. It does not matter which garbage page is evicted. Thus, the differences can only arise with medium-sized lists. That is, lists that can

fit from the beginning up to its middle element in main memory. As observed above, after a traversal all traversed pages become garbage with the possible exception of P_l. P_l is made young by LRU but not FIFO. Thus, LRU will evict garbage pages P_f through P_{l-1} before P_l while FIFO may swap P_l out once before the garbage pages. In other words, the inversion between P_l and the new garbage pages may cause under FIFO a single fault that LRU does not incur. Clearly, P_l will not always cause that extra fault under FIFO given that on occasion an element will be placed past P_l in the already sorted list, or multiple elements will be placed at the beginning of the this list forcing LRU to also swap it out before it is needed again. As the list of sorted elements grows we expect the former to be the more common case. In the former case P_l becomes garbage before causing a fault and in the latter case both replacement algorithms swap P_l eliminating any difference P_l could contribute.

3.2.3 Lessons from Insertion-sort

It is probably impossible to breakdown large complex programs in the manner we have done so for insertion-sort. Besides infeasibility, it is unlikely that any true insights can be gained from such an arduous task. It is the careful analysis of simple programs that we believe can reveal "higher-order" memory operations that are shared with complex programs. Insertion-sort revealed 3 memory operations that we believe are common to most heap intensive programs. Unlike others in the past that have studied the frequency of Lisp commands [16, 64], we are talking about higher level operations

like list-based structure creation, the traversal of simple lists, and repeated creation-traversal operation. If these "operations" are truly common and have distinctive virtual memory properties, it is not unreasonable to expect programs that employ such operations to inherit their virtual memory properties. We start by establishing that for the above mentioned "higher-order operations" FIFO performs as well as LRU.

Our first theorem relates to the creation of list-based structures. List creation is likely to be an important part of heap intensive programs, and thus, determining the virtual memory properties of creation is of vital importance. We focus our attention on list-based structures that are constructed without extracting their elements from preexisting structures. These structures are built, shall we say, from scratch and include the creation of items such as a list a random numbers, a list of the integers in $a..b$, or the creation of a matrix.

Theorem 3.1. *Under MT, FIFO induces the same number of page faults as LRU for the process of creating list-based structures that do not depend on preexisting structures.*

Proof The proof follows from an induction on the height of the tree representation of the list-based structure. As a base we have the case where there is only one level. In this case the structure is a simple list. As the elements of the list are encountered in the input expression or are generated (e.g. using random) they are placed on the stack. After all the elements are stacked they are popped-off one at the time.

Pages containing unallocated heap space are accessed linearly. Thus, $FIFO = LRU$. Furthermore, the ordering of the pages containing the list will be the same for both LRU and FIFO. Now, we assume the theorem holds for a $k - level$ structure. It will suffice to show that the theorem holds for a structure with $k+1$ levels. All sub-trees with k or less levels must be allocated before the backbone that points to them at the $k+1$ level is allocated. By assumption we know that the theorem holds for the creation of the structures that are pointed to by the $k+1$ level. Due to the MT allocation algorithm we know that the $k+1$ level is a simple list. By our base case we know that the theorem holds for the creation of a simple list. Thus, it follows that it holds for the creation of a $k+1$ level. **Q.E.D**

We next establish that the traversal of a simple list yields $FIFO = LRU$ when the memory resident pages that hold the list have the same memory ordering in relation to each other and to other memory resident pages under both FIFO and LRU. The memory state that we wish to capture here is that which exists after creating a simple list, or after traversing a simple list that is too large to fit in memory, or when both LRU and FIFO have the same consecutive list pages resident when their traversal begins. Given that we expect the MT system to be used with large lists this result is important. Furthermore, the traversal of large list-based structures should inherit this property since in MT they are level-linearized.

Theorem 3.2. *Under MT and starting in a state where pages holding a simple list, L, have the same ordering under FIFO and LRU and have the same relative ordering to garbage pages, the traversal of L yields FIFO=LRU.*

Proof The key to the proof is to establish what takes place when memory resident pages are traversed. The traversal of non-resident pages will cause faults whenever they are requested. So, for these pages $FIFO = LRU$. The differences can only arise due to the ordering of memory resident pages when the traversal of L starts. The ordering of garbage pages in relation to each other is irrelevant since swapping out a garbage page will never cause a fault later in the traversal. However, the ordering of live and garbage pages and of live pages amongst themselves is important because any inversions in this regard may cause a difference in performance. If live memory resident pages have the same relative ordering suggested above the traversal of the beginning of L will yield the same performance for FIFO and LRU even if any live pages to be were to be swapped out. Both LRU and FIFO will swap out the same page because live pages have the same ordering and the relative ordering with garbage pages is the same. Thus, when these swapped live pages need to be traversed both FIFO and LRU will exhibit the same behavior. The traversal of the, possible non-memory resident, end of L will also yield $FIFO = LRU$. Thus, the traversal of L yields that $FIFO = LRU$. **Q.E.D.**

The relevance of the above theorem may appear minimal at first glance because of the strong conditions needed for it to apply. However, the traversal of lists that occupy more space than that provided for the heap at the evaluating node fulfills the necessary conditions and we expect these lists to be common in MT. The following

theorem also establishes that the repeated creation and traversal of a simple list renders $FIFO = LRU$. We focus on the creation of simple lists built from "scratch" (i.e. built inductively from the empty list).

Theorem 3.3. *Under MT, for the creating and traversing cycle of a simple list that does not depend on preexisting non-atomic structures yields* $FIFO = LRU$.

Proof Theorem 3.1 establishes that the creation of simple lists yields $FIFO = LRU$. The question now reduces to determining the ordering of the pages that hold the list after creation. Clearly, if the list fits in memory then both page replacement algorithms have the same performance. The interesting case is the one where the list does not fit in memory. In this case, the pages left in memory after creation are those containing the front of the list (recall that elements are popped off the stack in reversed order). Furthermore, these pages have the same ordering under FIFO and LRU when the traversal begins. Therefore, the traversal of the list (which is simply a linear traversal of pages) will cause the same beginning pages to be swapped out by both replacement algorithms. Leaving memory in the same state (i.e. ordering of pages) with the ending pages of the list in main memory. Thus, the next creation and traversal cycle will find memory in a state that the ordering of pages by LRU and FIFO are the same **Q.E.D.**

This third theorem sheds light on why FIFO and LRU are so close for insertion-sort. The process generated by insertion-sort is simply the repeated creation and traversal of different lists. The front of the sorted list is always a new simple list

that is immediately traversed after creation. The pages containing the end of the list (i.e. those pages containing elements after the last page traversed by the previous traversal) will have the same ordering under FIFO and LRU. Thus there is only one page that may make a difference which is the last page accessed by the previous traversal.

It would appear that insertion-sort is a FIFO friendly process. In the next chapter we will explore the possibility that the operations described by the above theorems are common enough to yield the conclusion itself that MT is FIFO friendly. Informally, we conjecture that this is likely, since traversing large complex list-based structures usually occurs immediately after their creation. Furthermore, traversal of any level of a list-based structure is similar to traversing a simple list and inter-level traversals are likely to be a linear traversal of pages. This, of course, is not the whole story, but it does suggest that empirical measurements be taken to determine if MT is FIFO friendly (i.e., that for MT, FIFO performs approximately as well as LRU).

3.3 Summarizing Remarks

Despite contradictions in the record, we have extrapolated what we believe are common numbers establishing the relative difference between FIFO and LRU. According to the numbers presented by Margolin et al. this difference lies between 26% and 89%. This apparently justifies the popular belief that in general LRU is better than

FIFO.

We have presented a case study using insertion-sort that yielded a much tighter performance picture for the MT language. This raises the question of whether this is a property of insertion-sort or of MT. Perhaps, the answer is that it is a property of both, if the operations performed in insertion-sort are common to list-based programs. These operations are list-creation, list-traversal, and repeated creation-traversal of lists. We presented theorems establishing the approximate equivalence of these operations under FIFO and LRU. The discussion of insertion-sort also pointed out where the differences between the page replacement algorithms can occur.

This motivates the next chapter where we further compare MT virtual memory performance to that presented in the literature. We also begin to explore how well FIFO performs for the MT heap and stack.

Chapter 4

Virtual Memory Performance and Program Evaluation

The previous chapter suggested that the MT allocation algorithm is capable of producing competitive virtual memory performance and that the MT allocation algorithm tightens the gap between FIFO and LRU. Of course, conclusions can not be drawn on the basis of a single program. Nonetheless, the discussion established that the MT allocation algorithm is expected to produce competitive numbers as far as general virtual memory performance is concerned.

In this chapter, we will compare MT's virtual memory performance to data obtained from a study performed on INTERLISP by Bobrow and Grignetti [8]. Unlike most virtual memory studies on Lisp that focus on garbage collection (e.g. [56, 64, 70, 72]), this study focuses on the virtual memory behavior during evaluation. In fact, the numbers presented are the result of evaluation without calling a garbage collector. These numbers are of particular interest to us, because we are in-

terested in the virtual memory performance of the evaluator without interference by a garbage collector. Before presenting the INTERLISP results and the experiments on MT, we will briefly describe some conclusions reached in other studies.

In this chapter, we also justify the separation of stack and heap spaces. That is, we present an argument in favor of providing the heap and the stack with separate memories. The alternative would be to let them compete for frames from a global pool of frames at the evaluator node. It turns out that using a global pool of frames produces slightly fewer faults than allowing each component to have its own discrete pool of frames. However, the performance gap between LRU and FIFO is so tight that when one considers the effective access time of each system it is better to separate concerns. The performance gap between LRU and FIFO is also compared to that predicted by Margolin [68].

4.1 VM and Lisp

4.1.1 Early Studies

A study on the dynamic use of lists partly focused on the effects of linearization on paging [16]. The linearization described is similar to that achieved by the MT allocation algorithm. However, Clark's linearization was achieved by using a garbage collector. As noted during the discussion of our allocation algorithm (see Chapter 2), their linearization compacted lists to the point that fewer pages were used and intra-

list locality was built. On the two programs used in this study, linearization caused a reduction of 20% and 2% respectively in the number of pages storing lists. Clark, however, concluded that the effects on paging were small. For one program he found a page fault rate of 0.005 and 0.001 without and with linearization respectively. For his second program the rates were 0.0018 and 0.001. Based on the first two experiments in Chapter 2, the MT allocator caused an 88% reduction in the number of pages storing the list. The heap fault rate was reduced from 0.00014 to 0.00. It is difficult to compare Clark's numbers to ours, because he uses a garbage collector to linearize lists. Therefore, it is not clear how much eval's heap faulting has been reduced. In other words, it is impossible to distinguish how much of the faulting improvement is due to the activities of the garbage collection.

The results of a very early study on the virtual memory performance of Lisp based on a new allocation algorithm are reported by Bobrow and Murphy [9]. They report that under their allocation algorithm 2.5% of the references made to lists required accessing backing store. Due to this their system only spent 10% of its time waiting for faults to be serviced. It should be noted that the experiments were done on a machine that did not have hardware support for address translation. This is of interest to us since MT employs an all-software distributed virtual memory. MT, however, achieves much more favorable miss rates for heap and stack pages (see Chapters 5 and 6).

4.1.2 Fault Rates for INTERLISP

The most extensive study on LISP and virtual memory performance was performed by Bobrow and Grignetti [8]. Their study was based on measurements taken on a PDP-10 simulator. The simulator assumed the job of computing effective memory addresses and substituted in for the PDP-10 hardware that calculates these addresses. This allowed the authors to record memory reference patterns on which their study was based.

One of their goals was to study the expected number of page faults in relation to the number of pages in main memory the process owned. They presented numbers for memory sets ranging from 40 to 320 pages with the upper bound depending on the total number of pages used by the session. They assumed the use of the LRU page replacement algorithm. When a process started it faulted until it has brought in the maximum number of pages allowed. After this initial stage the least recently used page was chosen as a victim to make room for a page faulted on.

They present results for seven programs from which we were able to extrapolate page fault rates for six of them. We shall refer to these six programs as P1-P6. The miss rates for P1-P6 are displayed in Figure 4.1. The number of memory accesses for these six programs are similar in magnitude to the number of accesses for our benchmarks. Memory accesses for P1-P6 range from 1,204,224 to 8,392,704. Our benchmarks range from 4,094,428 to 6,136,099 (excluding **eval** with 22,430,082 ac-

cesses).

There are also differences since we only measure heap and stack accesses while they report all accesses. We report only heap and stack accesses, because these are the only structures that are distributed in the current version of MT. All other structures, such as the code and function table, fit entirely in the evaluator's RAM. The authors state that for the programs measured, lists take only 10-15% of the space, all atoms take over 20%, and over 50% is taken up by code (macros and compiled). Once again, differences in allocation are apparent which make these numbers relevant to our study. It is space allocation policy that we argue makes MT yield better fault rates.

We remind the reader that the INTERLISP data was taken from a machine where there was no distinction made between, for example, heap and stack pages as in MT. The different components must compete for frames from a global pool of frames. In the next section, therefore, we will present data on MT with a global pool of frames and with discrete pools of frames.

4.2 Virtual Memory Performance in MT

In this section we will present empirical measurements taken on MT. First, we will describe the programs that were chosen as benchmarks and present some basic measurements for each that are the same for every experiment (e.g. number of accesses).

After describing the benchmarks, we present measurements taken from a version of MT where at the evaluator node there is a global pool of frames for both stack and heap pages. By global we mean that the heap and stack are competing for the same frames. These measurements allow us to make a better comparison between MT and the numbers reported in the literature and allow us to develop a series of statistics that can serve as a baseline when assessing the effectiveness of discretizing heap and stack memory space. The former is achieved by comparing our global-pool numbers to the reported numbers in the literature. All of the measurements reported in the literature are taken from machines that use a global pool of frames that every language component competes for. We approximate this scenario by having the stack and heap compete for the same pool of frames. The latter will be achieved by comparing the data obtained from the global-pool system and the system that has separate pools of frames for the stack and the heap. That is, these numbers will be used to establish that the separation of (heap and stack) concerns is justified.

4.2.1 Program Selection for MT Measurements

We chose seven programs to run as benchmarks. Our goal was to select programs that were representative of Lisp programs in general. Our selection favored programs that heavily used list-based structures since lists are the only compound type that is currently supported in MT. We attempted to select algorithms that are widely known (i.e. familiar to most computer scientists), that use different list-based structures, and

that exhibited different access patterns. Furthermore, and unlike the benchmarks used in [27], all of our programs are pure (i.e. do not employ assignment). Programs were also chosen to allow for comparison between different algorithms that do the same thing (e.g. sorting) and between algorithms that use the same data structure (e.g. two graph algorithms).

All of our benchmarks are pure, that is, contain no assignment. Benchmarking of Lisp systems by others [8, 27], for example, have not restricted themselves to pure programs. Gabriel, however, includes several pure programs in his benchmarks. In addition, he includes specific benchmarks to profile destructive programs. This suggest that pure programs are not uncommon among functional programmers and that assignment is used infrequently. Therefore, it is valid to compare the performance of our benchmarks against others in the literature, despite the fact that they may be impure programs. Given that the number of accesses made by our benchmarks is similar to those made by programs in other studies, we further believe that comparison of our data with other data in the literature can give an indication of the effectiveness of our techniques.

The code for each of the seven algorithms can be found in Appendix II. The following is a brief description of the programs used:

- **ins**: This is an implementation of insertion-sort. Starting from with the empty list elements are added one at a time keeping the added elements sorted. The

program was tested with a randomly generated list of three hundred positive integers. In this program the original list of numbers is traversed once while the list of sorted numbers is repeatedly created and traversed until all numbers have been added to the list of sorted numbers. For our sorting of three hundred numbers the heap was accessed 238,704 and the stack was accessed 3,855,724 yielding a total of 4,094,428 accesses. The sort allocated 47,864 heap cells and accessed 94 different heap pages. The maximum stack height was 2,409 needing to accesses five different stack pages.

- **qs**: This is a version of quicksort that visits every sublist of unsorted numbers twice. One pass is made to extract 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 quick-sorting of the smaller elements. This program ran on a list of one thousand randomly generated positive integers. The heap was accessed 231,746 while the stack was accessed 4,419,143 for a total of 4650889. The number of heap cells allocated were 39092 requiring 77 heap pages. The stack reached a height of 7647 requiring 15 stack pages.

- MM: This is matrix multiplication. Each row of the matrix is represented by a simple list. A matrix is a list of rows. The resulting matrix is built one row at the time. This means that if we are computing AxB then each column of B

is extracted multiple times (i.e. once for each row of A). After creating each matrix this program spends most of its time traversing list-based structures. The program was used to multiply two randomly generated 20x20 matrices. The total number of accesses was 6,136,099 of which 327,242 were to the heap and 5,808,857 were to the stack. A total of 58,200 heap cells were allocated using 114 heap pages. The maximum stack height reached 3,188 requiring 7 stack pages.

- **fp**: This program finds a path from a to b in an undirected and unweighted graph G. G is represented using adjacency lists. Starting with 'a' all the unvisited neighbors 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. G used contained twenty-six nodes and ninety-eight edges. The fp program spends most of its time traversing different list-based structures like the graph, adjacency lists, and a queue. This program does not necessarily traverse the whole graph structure. G had 26 nodes and 71 edges. Finding a path from node 'a' to node 'x' required 260,452 heap accesses and 4,134,935 stack accesses for a total of 4,395,387 accesses. A total of 55,538 heap cells were allocated using a total of 109 heap pages. The stack reached a height of 3,117 requiring a total of 7 stack pages.

- **bfs**: This program performs a breadth-first search of the same graph, G, used

in fp. In this program, G 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 in MT. It takes as inputs an expression to be reduced to normal form, 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. As stated above, we ran **bfs** in **eval**. These experiments were the most memory intensive of the whole set. The total number of accesses 23,306,192. Of these, 876,110 were heap accesses while 22,430,082 were stack accesses. A total of 37,506 heap cells were allocated using 74 different pages. The stack reached a maximum height of 2590 requiring at most 6 pages at any given time during the computation.

The data that is presented in the remainder of this chapter (and dissertation) was obtained from experiments in which the page size was always kept at a size of 512 S-expression. As mentioned above, the size of main memory in frames was varied as a percentage of the total number of pages used. The code and input used for the experiments was always the same. For example, for every experiment involving quicksort the same input and the same code was always used.

4.2.2 Virtual Memory Performance for the MT Language Under a Global Pool of Frames

The MT measurements presented in this section were taken from a system employing a global pool of frames for the stack and heap. By global we mean that a heap fault can displace a stack page from main memory and vice versa.

Heap and stack pages are contained in a single pool of frames. When a reference is made to either one the page table is searched for the desired page number. Upon locating the desired page number the VM system must verify that the right type of page (i.e. heap or stack) has been found. Otherwise, the search must continue. If the desired page is not memory resident a request for it from backing store is made while a victim is chosen to be swapped out.

The results are shown in Tables 4.2-4.11. Each benchmark was tested for different sizes of main memory with the same data. The size of memory was varied by the percentages (10% to 90%) of the total number of pages used. This yielded nine experiments per benchmark for a total of forty-five experiments. Each experiment is presented by the percentage of memory used (i.e. 10% to 90%). In parenthesis, next to the percentage number, you will find the actual number, h, of frames initially assigned to the heap, and the actual number, s, of frames initially assigned to the stack.

The even numbered tables present the number of page faults incurred by both

FIFO and LRU. In addition they also display the relative difference between FIFO and LRU [1] The odd numbered tables present the fault rates under FIFO and LRU for the same experiments.

The fault rates reported for INTERLISP under LRU in table 4.1 range from 3.44e-5 to 3.20e-3. For the MT language using a global pool of frames the fault rates range from 4e-7 to 5.96e-5 under FIFO and from 3e-7 to 3.28e-5 under LRU. This means that, overall, MT under a global pool of frames (which is the default for INTERLISP) we have better fault rates. Given that we have the same magnitude of accesses as argued in 4.2.1, we can only conclude that INTERLISP is causing more faults. Therefore, we can conclude that for our experiments, both FIFO and LRU perform better than LRU for INTERLISP. This conclusion may be mitigated by the fact that Bobrow and Grignetti had code paged; we are currently engaged in estimating the impact of paging code in MT.

The extrapolation of direct conclusions from this comparison is difficult since the benchmark programs are different. There are, however, strong similarities between the benchmarks used by Bobrow et. al. [8] and our benchmarks for MT. For example, the total number of accesses in both studies are of similar magnitude [2] This suggests, as stated above, that the raw number of faults in INTERLISP is much larger than in MT given that there is one or two orders of magnitude difference in the fault rates.

[1] Note that the same result is obtained if fault rates are used instead of faults.
[2] The exception being **eval** which accesses memory much more than any of the other programs.

These observations, however, must be taken in context. In the Bobrow study most of main memory was taken up by code which in the current version of MT is always memory resident. We did not include code accesses in our numbers for this reason. Nonetheless, we believe that we can conclude that MT achieves competitive page fault rates for a system with a global pool of frames.

Recall from Table 3.3 that for the numbers reported by Margolin the relative difference between FIFO and LRU ranged from 26% to 89% with the average being 55%. For MT language using a global pool of frames we have that the relative difference between FIFO and LRU ranges from -88% to 1.01%. A negative value for the relative difference indicates that FIFO caused fewer faults than LRU. A value greater than one (1) indicates that FIFO caused more than twice as many faults as LRU. The latter only occurred under the special case when the stack was initialized with only one page in main memory. If one ignores the cases where the number of stack pages is one (1) the the relative difference between LRU and FIFO ranges from -88% to 25% with an average of -17%. This is the first evidence that we see that FIFO may play a prominent role in the MT system. Also noteworthy is how much tighter the LRU and FIFO numbers are when compared to those from Table 3.3. An average of -17% clearly indicated that in the MT (or functional) domain LRU is not outperforming FIFO as one would expect. In fact, the data suggests that FIFO should be used as a page replacement policy. We will continue to discuss this in

section 4.3.

The page fault rates for insertion-sort range from $2.2e-5$ to $4.42e-5$ for FIFO and from $2.2e-5$ to $3.10e-5$ for LRU. LRU does perform better than FIFO in all 9 experiments, but if we ignore the cases where the number of stack pages is initially one the maximum relative difference is 12%. The average of the relative differences (ignoring the data for the number of initial stack pages equal to one) is 6.57%. The small loss in VM performance incurred by using FIFO can probably be absorbed by the savings gained from not having to implement LRU in software.

The page fault rates for quicksort range from 1.9e-6 to 3.01e-5 for FIFO and from 1.46e-5 to 2.90e-5 for LRU. The performance of LRU and FIFO is very close with FIFO actually performing better than LRU in two experiments (with memory sizes at 40% and 90%). In themselves the results are interesting since this is the first time we see FIFO performing better than LRU in a "real" program. We have all seen examples of access streams for which FIFO beats LRU in an Operating Systems book, but this is our first observing the phenomena with a "naturally" occurring program. In fact, accepting the result was difficult, because all recent results suggest that in practice LRU is truly better than FIFO [40, 14].

The average of the relative differences is -8.89%. If the list of numbers sorted by these nine experiments is representative of the average list of numbers then FIFO should be used. On average it will perform better than LRU and will not incur a

MMSIZE	P1	P2	P3	P4	P5	P6
20	3.2112e-3	1.4820e-3	2.3652e-3	1.2654e-3	1.1964e-3	1.7842e-3
30	1.2880e-3	6.968e-4	8.955e-4	5.911e-4	4.244e-4	5.014e-4
40	6.851e-4	5.109e-4	4.449e-4	4.207e-4	2.153e-4	1.656e-4
50	4.767e-4	3.911e-4	2.973e-4	3.227e-4	1.271e-4	6.95e-5
60	3.297e-4	1.826e-4	2.102e-4	1.618e-4	8.61e-5	4.84e-5
70	2.782e-4	1.511e-4	1.117e-4	1.144e-4	6.23e-5	3.90e-5
80	2.375e-4	1.418e-4	9.85e-5	1.024e-4	4.72e-5	3.51e-5
90	2.059e-4	1.188e-4	8.93e-5	8.47e-5	4.12e-5	3.44e-5

Table 4.1: Page Fault Rates for INTERLISP

Memory Size (h,s)	FIFO	LRU	Rel. Diff.
10 (9,1)	181	127	0.43
20 (19,1)	131	106	0.24
30 (28,2)	114	102	0.12
40 (38,2)	109	99	0.10
50 (47,3)	105	97	0.08
60 (57,3)	102	97	0.05
70 (66,4)	101	95	0.06
80 (76,4)	100	95	0.05
90 (85,5)	9	9	0

Table 4.2: Page Faults for Insertion-Sort in MT Using a Global Pool of Frames

Memory Size	FIFO	LRU
10	4.42e-5	3.10e-5
20	3.20e-5	2.59e-5
30	2.78e-5	2.49e-5
40	2.66e-5	2.42e-5
50	2.56e-5	2.37e-5
60	2.49e-5	2.37e-5
70	2.47e-5	2.32e-5
80	2.44e-5	1.32e-5
90	2.20e-5	2.20e-5

Table 4.3: Page Fault Rates for Insertion-Sort in MT Using a Global Pool of Frames

Memory Size (h,s)	FIFO	LRU	Rel. Diff.
10 (8,2)	140	135	0.04
20 (15,3)	106	105	0.01
30 (23,5)	93	91	0.02
40 (32,6)	85	87	-0.02
50 (40,8)	85	84	0.01
60 (47,9)	85	84	0.01
70 (55,11)	81	81	0.00
80 (62,12)	79	78	0.01
90 (70,14)	9	68	-0.87

Table 4.4: Page Faults for Quicksort in MT Using a Global Pool of Frames

Memory Size	FIFO	LRU
10	3.01e-5	2.90e-5
20	2.28e-5	2.26e-5
30	2.00e-5	1.96e-5
40	1.83e-5	1.87e-5
50	1.83e-5	1.81e-5
60	1.83e-5	1.81e-5
70	1.74e-5	1.74e-5
80	1.74e-5	1.68e-5
90	1.9e-6	1.46e-5

Table 4.5: Page Fault Rates for Quicksort in MT Using a Global Pool of Frames

heavy penalty for implementing it in software. The small loss in performance incurred by some runs when using FIFO can probably be absorbed by the savings gained from not having to implement LRU in software.

Matrix multiplication is the second most memory intensive of our benchmarks. Ignoring the case where the number of stack pages is initialized to one (1), we can observe that the relative difference between LRU and FIFO ranges from 3% to 31%. The figure of 31% appears to be an aberration since it oddly sticks out from the rest of the data. In fact, this is a perfect example to illustrate that when the the number of page faults is low the relative difference can be high suggesting a real difference in performance where none exists. The relative difference of 31% occurs when main memory has a size of 90% of the working-set. The absolute difference between LRU and FIFO is only four (4) page faults. After 5.8 million accesses to memory this difference in page faults is negligible and we can see that relative difference alone can not be relied on to judge virtual memory performance. A quick glance at the numbers at 40% drives the point home. The absolute difference in page faults is twice as big but the relative difference is only 8% which is significantly less than 31%. Thus, if we are allowed to ignore the 31% figure we have that the range of the relative difference is from 3% to 9%. This suggest that, indeed, there is no significant difference between FIFO and LRU for matrix multiplication.

The fault rates range from 2.8e-6 to 3.34e-5 for FIFO and from 2.1e-6 to 2.20e-5

Memory Size (h,s)	FIFO	LRU	Rel. Diff.
10 (11,1)	205	135	0.52
20 (23,1)	146	129	0.13
30 (34,2)	119	110	0.08
40 (46,3?)	98	90	0.089
50 (57,4?)	80	76	0.053
60 (68,4)	63	61	0.033
70 (80,5?)	46	44	0.05
80 (91,6?)	32	30	0.07
90 (103,6?)	17	13	0.31

Table 4.6: Page Faults for Matrix Multiplication in MT Using a Global Pool of Frames

Memory Size	FIFO	LRU
10	3.34e-5	2.20e-5
20	2.38e-5	2.10e-5
30	1.94e-5	1.79e-5
40	1.60e-5	1.47e-5
50	1.30e-5	1.24e-5
60	1.02e-5	9.9e-6
70	7.5e-6	7.2e-6
80	5.2e-6	4.9e-6
90	2.8e-6	2.1e-6

Table 4.7: Page Fault Rates for Matrix Multiplication in MT Using a Global Pool of Frames

for LRU. These numbers are one order of magnitude better than those reported for INTERLISP. In fact, FIFO achieves fault rates in MT that were not achieved by LRU in INTERLISP.

The results for **fp** are the most unexpected up to now. For seven (7) of the nine (9) experiments FIFO performed as well as or better than LRU. The average of the relative differences is -7.38%. This table clearly establishes once and for all that FIFO can not be dismissed out of hand as a viable page replacement policy for the MT language. Once again, the absolute differences are small suggesting that overall FIFO and LRU performance is the same. The page fault rates are competitive with both LRU and FIFO exhibiting lower miss rates than those reported for INTERLISP.

The results for **eval**, the most memory intensive of our benchmarks, suggest that the difference between LRU and FIFO becomes less significant as the number of accesses increases. In other words, as a program spends more time in its steady state we find that in practical terms FIFO performs as well as LRU. The fault rates become extremely competitive beating the virtual memory performance reported for INTERLISP and all the other benchmarks in this MT study.

The experiments reported here establish that MT and its allocation algorithm produce competitive virtual memory performance that is in line with the performance reported for other functional systems. The MT allocation algorithm renders FIFO a page replacement policy that must be seriously considered for implementation in the

Memory Size (h,s)	FIFO	LRU	Rel. Diff.
10 (11,1)	262	144	0.82
20 (22,2)	159	129	0.23
30 (34,2)	123	125	-0.02
40 (45,3)	99	109	-0.09
50 (56,4)	80	85	-0.06
60 (66,4)	61	80	-0.24
70 (77,5)	41	47	-0.13
80 (88,6)	26	28	-0.07
90 (98,6)	16	16	0

Table 4.8: Page Faults for Find Path in MT Using a Global Pool of Frames

Memory Size	FIFO	LRU
10	5.96e-5	3.28e-5
20	3.62e-5	2.93e-5
30	2.80e-5	2.84e-5
40	2.25e-5	2.48e-5
50	1.82e-5	1.93e-5
60	1.39e-5	1.82e-5
70	9.3e-6	1.07e-5
80	5.9e-6	6.3e-6
90	3.6e-6	3.6e-6

Table 4.9: Page Fault Rates for Find Path in MT Using a Global Pool of Frames

Memory Size (h,s)	FIFO	LRU	Rel. Diff.
10 (7,1)	291	145	1.01
20 (15,1)	88	79	0.11
30 (22,2)	67	65	0.03
40 (30,2)	56	65	-0.14
50 (37,3)	47	42	0.12
60 (44,4)	33	33	0.00
70 (52,4)	26	25	0.04
80 (59,5)	18	17	0.06
90 (67,5)	10	8	0.25

Table 4.10: Page Faults for **eval** in MT Using a Global Pool of Frames

MT system. The close performance between FIFO and LRU can only be attributed to the properties list have when allocated. As with the simple lists used by Insertion-sort in chapter 2, the data structures used by the benchmarks are not intermingled with garbage. This is certainly the case for the simple lists created and used by **ins** and **qs**. The data structures created and used by **MM**, **fp**, and **eval** are level-linearized and, therefore, contain no garbage within a list at any level. This means that garbage is not likely to be making a difference in VM performance.

The lack of garbage interference allows FIFO to perform as well as or even better than LRU. Since simple lists are only allocated after all the members of the list are known (in the absence of parallel cons-*ing* as in the benchmarks) all the garbage generated during creation is "older" than the list itself according to FIFO. The same is true for LRU, but one can note that LRU will note make this garbage young again. Thus, this garbage can not cause a difference between FIFO and LRU when working with the list created. The garbage must be swapped out before the list by both algorithms. The exception to this observation will be when live data is caught on the same page with garbage. This occurrence becomes the exception and not the norm as data structures grow larger. Thus, the gap between LRU and FIFO is expected to become smaller as data structures become larger.

The lack of intermingled garbage is only part of the story. After all, we have established that compaction is only a necessary condition, but not sufficient, for good

VM performance. The pattern of accesses is also of critical importance. As suggested in chapter 3, the creation and traversal of lists are the pivotal bases on which list-based algorithms rely on. The sorting algorithms repeatedly create and traverse simple lists. Matrix multiplication and finding path traverse a list of simple lists. Finally, MT in MT (i.e. eval) traverses several list-based structures that are level-linearized like the function-table, the expression to be evaluated and the chain of activation records. As established in Chapter 3 these operations yield FIFO as good as LRU.

In the next section we present the numbers obtained from separating the heap and the stack memory. We will explore if the possibility of separating heap and stack memory produces better virtual memory performance.

4.2.3 MT's Virtual Memory Performance with Discrete Heap and Stack Memory

In this section, we present the results of experiments where the underlying structure of the MT memory system was in place. Stack and heap pages no longer have access to a global pool of pages. Instead, they each draw from a private pool of pages. In essence, there are two (2) virtual memory managers; one for each component. In this subsection, we will try to layout the empirical ground work that will allow us to conclude in the next subsection that this division of concerns among components is justified. In the following chapters (5 and 6) we will then try to further the design of each of these components.

As in the previous subsection, the size of main memory in pages, n, was varied from one experiment to the next as a percentage of the total number of pages used. For any given experiment at $d\%$ n is approximately $d\%$ of the total number of heap pages, h, used plus $d\%$ of the total number of stack pages, s, used. That is, $n \approx d\% * n + d\% * s$. The actual number of heap and stack pages can be found in parenthesis in the memory size column of the tables containing the number of page faults incurred by FIFO and LRU, and the relative difference between the two. The first number in parenthesis is h, the number of heap pages, and the second is s, the number of stack pages.

A global inspection of the tables reveals that the page fault rate for FIFO varied from 5e-7 to 1.346e-4 and from 4e-7 to 9.50e-5 for LRU. These ranges ignore the special case where there is only 1 stack frame allocated. The lowerbound of the ranges is better than those reported for INTERLISP. The upperbound, however, is slightly better for INTERLISP. The upperbounds for the MT language using a global pool of frames are also slightly better. A closer look at the tables reveals that these upperbounds occur when the stack has only 2 frames in main memory. Experiments with a stack that has 3 or more frames in main memory yield a better virtual memory performance than INTERLISP. This suggests that the stack must be allowed to hold a minimum of three frames at the evaluator node.

The MT language using a global pool of frames produces fewer faults than the MT system under FIFO. That is, a global pool of frames under FIFO yields a better VM

Memory Size	FIFO	LRU
10	1.24e-5	6.2e-6
20	3.7e-6	3.3e-6
30	2.8e-6	2.7e-6
40	2.4e-6	2.7e-6
50	2.0e-6	1.8e-6
60	1.4e-6	1.4e-6
70	1.1e-6	1.0e-6
80	7e-7	7e-7
90	4e-7	3e-7

Table 4.11: Page Fault Rates for **eval** in MT Using a Global Pool of Frames

Memory Size (h,s)	FIFO	LRU	Rel. Diff.
10 (9,1)	8168	8163	0.00
20 (19,1)	8148	8146	0.00
30 (28,2)	551	389	0.39
40 (38,2)	541	379	0.43
50 (47,3)	217	153	0.42
60 (56,3)	208	144	0.44
70 (66,4)	112	47	1.38
80 (75,4)	102	38	1.68
90 (85,4)	92	28	2.29

Table 4.12: Page Faults for **ins** in MT

10	1.9949e-3	1.9937e-3
20	1.9900e-3	1.9895e-3
30	1.346e-4	9.50e-5
40	1.321e-4	9.26e-5
50	5.30e-5	3.74e-5
60	5.08e-5	3.52e-5
70	2.74e-5	1.15e-5
80	2.49e-5	9.3e-6
90	2.25e-5	6.8e-6

Table 4.13: Page Fault Rates for Insertion-Sort in MT

performance than a system where there is a separation of heap and stack concerns under FIFO. This behavior occurred in all but one of the forty-five (45) experiments (qs at a memory size of 80%). Under LRU, we can observe that that the system with the global pool of frames outperforms the MT system most of the time. For some experiments, MT under LRU performed better than the system with a global pool of frames. For these experiments, however, the difference between the systems is very small. On the other hand, the system with the global pool of frames can outperform the MT system by a significant amount.

One of the reasons the system with the global pool of frames outperforms the system with discrete memories is the flexibility with which a frame can be part of the heap or the stack. When few stack frames are allocated, for example, the global pool of frames allows the stack to "steal" frames that were initialized to the heap. We can observe that the use of a global pool of frames disallows many page faults the discrete memory system can not avoid since the stack cannot "steal" heap frames.

Studying page fault rates will not help in answering this question since they do not carry the same information as the number of page faults. Fault rates are probably a better metric to measure different systems with each other. In this case, however, the rate is just the number of page faults divided by a constant. This constant is the same for both LRU and FIFO and for both systems (global and discrete heaps) which is the number of accesses. This number does not change regardless of the page

Memory Size (h,s)	FIFO	LRU	Rel. Diff.
10 (8,2)	570	411	0.37
20 (15,3)	375	325	0.15
30 (23,5)	256	223	0.15
40 (31,6)	218	179	0.22
50 (39,8)	164	121	0.36
60 (46,9)	146	94	0.55
70 (56,11)	95	49	0.94
80 (62,12)	59	28	1.11
90 (69,14)	22	11	1.00

Table 4.14: Page Faults for Quicksort in MT

Memory Size	FIFO	LRU
10	1.226e-4	8.84e-5
20	8.06e-5	6.99e-5
30	5.50e-5	4.79e-5
40	4.69e-5	3.85e-5
50	3.53e-5	2.60e-5
60	3.14e-5	2.02e-5
70	2.04e-5	1.05e-5
80	1.27e-5	6.0e-6
90	4.7e-6	2.4e-6

Table 4.15: Page Fault Rates for Quicksort in MT

replacement algorithm or the allocation of frames at the evaluating node.

If we were designing a hardware based distributed virtual memory where all page faults took a constant amount of time to be serviced and where the LRU overhead was small, then the data may suggest implementing LRU. However, in an all-software based distributed virtual memory page faults do not take a constant amount of time to be serviced. In fact, the memory access time is not constant. This, of course, is system, hardware, and implementation dependent. In a traditional virtual memory system there exist a page table that provides the base address of a page if it is memory resident or signals the need to call the page fault manager.

The efficiency of the system hinges on how the page table is implemented. If the page table is small it is held in a set of registers that can be searched fast for base addresses. This, however, will never be the case in MT since we can expect to have a very large number of pages available. When the number of pages is large it becomes too expensive to have a set of registers for the page table. In this case, the page table is broken up into pages, but the search for a page is aided by a translation-lookaside buffer (TLB) and/or two level mappings (e.g. segments broken into pages) [66]. Both of these techniques require hardware to make the construction of real addresses fast. Neither, however, eliminates the fact that the system can fault on a page-table page. Given that a dedicated cache for TLBs is not readily available for our use on every system and that the overhead in software for two level mapping

Memory Size (h,s)	FIFO	LRU	Rel. Diff.
10 (11,1)	13,452	13,404	0.00
20 (23,1)	13,410	13,392	0.00
30 (34,2)	124	108	0.15
40 (46,3)	100	91	0.10
50 (57,4)	84	75	0.12
60 (68,4)	67	62	0.08
70 (80,5)	50	45	0.11
80 (91,6)	35	30	0.17
90 (103,6)	20	15	0.33

Table 4.16: Page Faults for Matrix Multiplication in MT

Memory Size	FIFO	LRU
10	2.1993e-3	2.1844e-3
20	2.1854e-3	2.1825e-3
30	2.02e-5	1.76e-5
40	1.63e-5	1.48e-5
50	1.37e-5	1.22e-5
60	1.09e-5	1.01e-5
70	8.1e-6	7.3e-6
80	5.7e-6	4.9e-6
90	3.3e-6	2.4e-6

Table 4.17: Page Fault Rates for Matrix Multiplication in MT

makes it impractical, neither technique is appropriate for an all-software distributed virtual memory system. Furthermore, the overhead of managing a huge page table would significantly slow down the MT evaluator.

Adding hardware support, however, does not invalidate the idea of separating concerns for at least three reasons. First, the page table becomes smaller and easier to manage. Second, the entire virtual address space does not have to be searched for a page. Third, nodes of backing store can be added dynamically to a network depending on the demands put on the system.

The addition of hardware, also, does not automatically invalidate the arguments for in favor of FIFO. The answer would depend on how much faster could FIFO circuitry be over LRU circuitry. The gap between LRU and FIFO may or may not be enough to justify one over another.

Based on these observations, it was decided that in MT the page table would be kept small by only keeping track of the logical pages are being held by the real frames at the evaluator node. In fact, the traditional virtual memory page table was substituted by a table that only contained an entry for the current pages held at the evaluator node. That is, the MT page table has an entry for each $frame_i$ that contains the $logical-page_j$ being held at $frame_i$. Since any page can be held in any frame, this means that to access a $page_i$ the table needs to be searched linearly to find the location of a page or to signal a fault. If the required address is in memory

Memory Size (h,s)	FIFO	LRU	Rel. Diff.
10 (11,1)	9,643	9605	0.00
20 (22,1)	9,605	9591	0.00
30 (33,2)	770	540	0.43
40 (44,3)	404	249	0.62
50 (55,4)	89	80	0.11
60 (65,4)	73	66	0.11
70 (76,5)	45	46	-0.02
80 (87,6)	29	30	-0.03
90 (98,6)	18	16	0.13

Table 4.18: Page Faults for **fp** in MT

Memory Size	FIFO	LRU
10	3.75e-5	2.89e-5
20	2.89e-5	2.57e-5
30	2.43e-5	2.23e-5
40	2.02e-5	1.93e-5
50	1.64e-5	1.59e-5
60	1.27e-5	1.27e-5
70	9.1e-6	9.6e-6
80	5.9e-6	6.4e-6
90	3.4e-6	3.2e-6

Table 4.19: Page Fault Rates for **fp** in MT

its content is returned. Otherwise, a fault is signaled by a call to the fault manager. When the fault is serviced the desired contents is returned after swapping out a victim, swapping in the needed page, and updating the MT page table. Clearly, the memory access time is not constant. Requests for pages near the beginning point of the search will be serviced faster than those that are further away. This approach may be a far departure from anything implemented on a sequential system, but it does eliminate faults on the page table, and the excessive memory and processing overhead of maintaining a large page table.

In the current implementation of MT, the search for a page in backing store is done in constant time, because we only have one level of backing store implemented with one node. If multiple levels and multiple nodes are at work for each component then the network that owns the page must be searched. For example, a heap page would be searched for in the heap network. We expect the searches to proceed no further than one hop from the evaluator since the MT prefetcher will be responsible for keeping pages that can be faulted on near the evaluator (see Chapter 7 for a description of the proposed MT prefetcher). Bearing this in mind and the high hit rates reported, we do not expect that multiple nodes and levels of backing store will affect the effective access times calculations presented in the next section. Keeping faults low and potential fault pages one hop away from the evaluator suggest that the weight of the contribution made by searches over multiple hops is negligible.

Memory Size (h,s)	FIFO	LRU	Rel. Diff.
10 (7,1)	57,327	57,271	0.00
20 (15,1)	57,241	57,230	0.00
30 (22,2)	85	70	0.21
40 (30,2)	75	62	0.21
50 (37,3)	48	43	0.12
60 (44,4)	37	34	0.09
70 (52,4)	29	26	0.18
80 (59,5)	20	17	0.18
90 (67,5)	12	9	0.33

Table 4.20: Page Faults for **eval** in MT

Memory Size	FIFO	LRU
10	2.4597e-3	2.4573e-3
20	2.4560e-3	2.4556e-3
30	3.6e-6	3.0e-6
40	3.2e-6	2.7e-6
50	2.1e-6	1.8e-6
60	1.6e-6	1.5e-6
70	1.2e-6	1.1e-6
80	8e-7	7e-7
90	5e-7	4e-7

Table 4.21: Page Fault Rates for **eval** in MT

The data presented, thus far, clearly establishes that the time it takes to run a MT program will not be dominated by the number of faults or the fault service time. MT's fault rates clearly establish that the allocation algorithm has rendered a virtual memory friendly system. If one ignores the special cases where the stack frames are less than 3 there is no evidence that the system thrashes. So, what will determine system performance? Should we use a global pool of frames or the MT architecture? Since the total number of faults and the fault rates can answer these questions for MT we must find another metric that is capable of truly discriminating between the different design choices. Given that memory access time is not constant it appears that the effective access time of the different design choices will yield the greatest insight.

4.3 The Case for Discretizing MT's Heap and Stack

The system with a global pool of frames consists of K_G frames that can hold either stack or heap pages at any time. The MT system consists of two discrete pools of frames where the heap pool has K_H frames and the stack pool has K_S frames. In order to measure the effective access time we decided to count the number of cycles needed for a non-fault memory access and for a page fault memory access. Counting cycles, instead of nanoseconds, abstracts away from the current platform MT has been implemented on. Thus, the effective access time on any machine executing

the same assembly code can quickly be computed. It also provides an easy way to compare the performance of implementations on different platforms by normalizing what a cycle represents. The important facts to remember are that memory accesses do not take a constant amount of time and depend on K, whether using a global pool of frames or sets of discrete pools of frames and that there is an implementation overhead per memory access for LRU that does not exist for FIFO. Both of these are implementation and platform independent. Therefore, the effective access times that we present below are specific to our implementation but are presented to illustrate a point that would hold true for any implementation platform. The specific number of cycles will vary from implementation to implementation, but the results will remain the same.

The cycle counts that we present were obtained from translating our Occam code for heap and stack accesses to the native assembly code of the transputer. The translation was based and guided by The Transputer Compiler's Writer Guide [53]. The translation was mostly a straight forward exercise of finding the translation of different Occam primitives to transputer assembly. After the translation to assembly, The Transputer Data Book [52] was used to look-up the cycle times for each assembly instruction. The translation was a very conservative one in which the worst choice was always taken. For example, the expressions found for page fault time assume that pages are always swapped out to backing store when this is not always case. It

is not always necessary to copy a heap page out to backing store. Nonetheless, we assumed that they were to err on the side of caution. Notice that these conservative decisions do not influence the relative difference between the different systems we have implemented since they all execute the same code.

The translation to assembly and the counting of cycles yielded the following basic equations for the number of cycles per memory access under FIFO, ma_f, and LRU, ma_l. These are the cost for accesses that do not cause a fault. K is the number of frames allocated at the evaluator node:

$$ma_f = 23 \cdot K + 69$$

$$ma_l = 23 \cdot K + 124$$

The difference between the two is due to the overhead associated with LRU. After every access, LRU must time stamp the accessed page and increment the clock. The K term is associated with the search for the frame that has the needed page. On average, we expect to traverse half the page table which takes $23 \cdot K$ cycles. The number of cycles required by a fault access, fa, is defined as follows:

$$fa_f = 46 \cdot K + 7,260 \qquad fa_l = \frac{125}{2} \cdot K + 7,373$$

Both FIFO and LRU must scan the entire page-to-frame table to determine that a

MMSIZE	INS	QS	MM	FP	EVAL
10	354.24	354.22	400.17	400.25	308.05
20	584.21	538.18	676.17	676.24	492.03
30	814.21	768.16	952.16	952.25	676.02
40	1044.21	998.16	1274.14	1228.23	860.02
50	1274.22	1228.17	1550.12	1504.19	1044.02
60	1504.23	1412.17	1780.10	1734.18	1228.01
70	1734.23	1642.17	2102.04	2010.11	1412.01
80	1964.14	1826.17	2378.05	2286.07	1596.01
90	2194.24	2056.15	2654.02	2516.04	1780.00

Table 4.22: Effective Access Times for MT Using a Global Pool of Frames Under LRU

MMSIZE	INS	QS	MM	FP	EVAL
10	299.33	299.22	345.25	345.45	253.09
20	529.25	483.17	621.18	621.29	437.03
30	759.22	713.16	897.16	897.22	621.02
40	989.22	943.15	1219.13	1173.19	805.02
50	1219.22	1173.15	1495.11	1449.16	989.02
60	1449.22	1357.16	1725.09	1679.12	1184.61
70	1679.22	1587.15	2047.07	1955.08	1357.01
80	1909.24	1771.16	2323.05	2231.06	1541.01
90	2139.20	2000.99	2599.03	2461.03	1725.00

Table 4.23: Effective Access Times for MT Using a Global Pool of Frames Under FIFO

fault has occurred. Thus, we have a factor of at least $46 * K$ in each equation. For LRU the search for a victim makes the constant on the K larger. FIFO needs no such search because a pointer to the next FIFO victim is always maintained. Therefore, the FIFO page fault manager only needs to return the current value of the pointer and increase it by 1 (modulo K) to the next page. The differences with the constants being added are due to LRU's time stamping, clock increasing, and victim searching. The magnitude of these constants have can be attributed to the copying of a page to backing store, the swapping in of the needed page, the search time for the page, and the updating of the MT page table. As suggested earlier, if multiple levels and nodes are implementing backing store terms will be added to both fa to reflect the searching and communication overhead. The contributions of these quantities depend on the probability of finding $page_i$ at level k. As discussed earlier, we expect their contributions to the effective access time be small based on the observed fault rates and the running of a prefetcher.

By taking the miss rate, mr, from the tables for the global pool of frames system above we can now define the effective access time, eft, for the system with the global pool of frames as:

$$eft_{gf} = (1 - mr_{gf}) * ma_f + mr_{gf} * fa_f$$

$$eft_{gf} = (1 - mr_{gf}) * (23 * K_G + 69) + mr_{gf} * ((125/2) * K_G + 7,373)$$

The results for the system using a global pool of frames are exhibited in Tables 4.23 and 4.22. Both the results for LRU and FIFO are dominated by the *ma* factors. This is clearly due to the low miss rates exhibited by the systems with a global pool of frames. This also explains why the effective access time increases as the size of main memory increases despite the fact that the fault rate decreases. You can observe that ma_f is slightly better than ma_l. In most instances the difference is exactly $mal - maf$ which is fifty-five (55) cycles.

The effective access time of the MT system depends on both the heap and stack pools of frames. In order to determine the contribution of each to the effective access time of MT it is necessary to know the probability of a heap access, p_h and the probability of a stack access, p_s. These quantities are easily computed from the statistics reported in the description of each benchmark. Thus, if we define eft_h and eft_s as the effective access times of the heap and stack respectively, we have that the MT effective access times for FIFO and LRU are:

$eft_{MTf} = p_h * eftMTfh + p_s * eft_{MTfs}$ and

$eft_{MTl} = p_h * eftMTlh + p_s * eft_{MTls}$ and

Using the tables shown in chapters 5 and 6 to define the MT miss rates for the

stack and heap, mr_{MTfs} and mr_{MTfh} for FIFO, and mr_{MTls} and mr_{MTlh} for LRU, we have eft_{MTfh} and eft_{MTfs}, and eft_{MTlh} and eft_{MTls} defined as follows:

$$eft_{MTfh} = (1 - mr_{MTfh}) * maK_H + mr_{MTfh} * faK_H$$

$$eft_{MTfs} = (1 - mr_{MTfs}) * ma_{K_S} + mr_{MTfs} * faK_S$$

$$eft_{MTlh} = (1 - mr_{MTlh}) * ma_{K_H} + mr_{MTlh} * faK_H$$

$$eft_{MTls} = (1 - mr_{MTls}) * ma_{K_S} + mr_{MTls} * faK_S$$

Before presenting the data we observe that it is clear that the dominating factor in the MT effective access time should be the effective access time of the heap. This follows from the fact that under the MT allocation algorithm the overwhelming majority of the memory accesses are to the heap. In our experiments, p_s was always over 0.90. It is also noteworthy that the number of frames allocated to the heap is much larger than the number allocated to the stack. Thus, the MT effective access time should be smaller than that of the system with a global pool of frames. Since $K_S << K_G$ the effective access time should not grow as fast for MT.

The results of our experiments are displayed in Tables 4.25 and 4.24. As predicted, the MT effective access time is much smaller than the effective access time of the

system with the global pool of frames.

The results suggest that it is indeed a good idea to separate heap and stack spaces. In this manner, the MT stack is allowed to dominate memory accesses without interference from the heap. By no interference we mean two things. The first is that the heap can not force a stack fault by swapping out a stack page. The second is that there is no need to search through heap pages to resolve a logical stack address. The latter is of particular importance if efficiency is important. Making the the size of the stack at the evaluator too small will cause the system to thrash as evidenced by the data presented for MT with a stack of size 1 or 2. Making the stack too large would cause ma_{K_S} to grow unnecessarily large significantly degrading the effective access time.

4.4 Concluding Remarks

This chapter has established that the MT system achieves competitive page fault rates. There is no evidence that the system thrashes unless the number of stack frames at the evaluator node is less than 3. The rates achieved can be attributed to the MT allocation that creates linearized simple lists and level-linearized complex list-based structures.

For an all-software DVM system the traditional metrics for performance like number of page faults and fault rates do not suffice to establish the best performance.

MMSIZE	INS	QS	MM	FP	EVAL
10	172.26	177.53	175.19	176.57	170.08
20	185.65	207.28	189.92	191.54	176.98
30	205.58	260.00	209.40	213.18	187.32
40	218.97	290.95	245.88	249.33	194.23
50	252.29	343.74	281.12	285.67	222.41
60	264.35	373.57	294.60	299.28	250.60
70	299.28	428.66	331.07	355.87	257.51
80	311.28	457.36	366.30	372.47	285.70
90	324.67	509.06	381.01	387.42	292.61

Table 4.24: Effective Access Times for MT Under LRU

MMSIZE	INS	QS	MM	FP	EVAL
10	117.23	122.77	120.09	121.46	114.91
20	130.50	152.34	134.76	136.39	121.80
30	150.81	205.04	154.41	158.43	132.32
40	164.20	236.00	190.88	194.56	139.23
50	197.40	288.79	226.12	230.67	167.41
60	209.45	318.64	239.60	244.27	195.60
70	222.84	373.73	276.07	280.85	202.51
80	256.39	402.40	311.31	317.45	230.70
90	269.78	454.07	326.01	332.42	237.61

Table 4.25: Effective Access Times for MT Under FIFO

As evidenced in this chapter, other metrics like effective access time are necessary to establish the best performance among different design choices. The effective access time proved pivotal in establishing that the discrete separation of heap and stack space was the best choice for the MT language. The separation of concerns allows for each to be separately designed and optimized.

A parameter that can be used to tune the system is the size of the heap, K_H, at the evaluator node. The size of the stack is not important beyond noting that it should be greater than 3 as noted above. This is because in Chapter 6 we present an optimal page replacement algorithm for the MT stack that is not proportional to K_S. The ideal value for K_H may be found by graphing page faults against memory sizes. This type of graph that you will see is a parachor curve which will reveal the point at which increases in memory will have little effect on page fault rates [54, 66]. That is, these graphs have a point called a knee after which increases in memory only have a small effect on paging. Thus, increasing heap size beyond this point can only degrade the effective access time. The closer the knee is approximated on a typical load of programs the better performance the system will provide.

Chapter 5

The MT Heap

In this chapter we present the virtual memory performance numbers collected from forty-five (45) experiments performed on the MT heap. This heap has its own address space that is shared with no other data structure. We first present the page faults and page fault rates observed on our five benchmarks. We then proceed to present an argument in favor of FIFO as the MT Heap page replacement policy. We end the chapter by using a metric called activity count to determine if a prefetcher has a role to play at the evaluator node. This metric was first introduced by Margolin et. al. [68] to study the efficiency of virtual memory. Before proceeding, it is important to note that the separation of the heap and the stack spaces is not the same as a technique called BIBOP [45, 27, 22]. BIBOP is a method of organizing the heap address space by the type of object being stored. Objects of the same type of all stored together in the same space. The simplest technique designates whole pages to hold objects of a certain type. In this manner the type of an object is embedded in its address. To

determine the type of an object one only needs to look up its page number in a table that maps pages to objects. A page, however, may too large an amount to allocate for a single type of object. In this case, a page can be broken-up into segments that hold different types. Unlike pages, the size of a segment is not predetermined in advance. This scheme requires that a segment table be associated with each page. The BIBOP scheme is in use by several languages including Chez Scheme [22]. The MT Architecture divides the address space among the data structures needed to evaluate a program. For example, the MT heap only contains lists and their elements while the heap only contains activation records. Furthermore, every object has a a generic representation allowing the different spaces to hold all types of objects. For example, MT integers are represented with 64 bits plus an 8-bit tag which can reside in either the heap or the stack. That same space could be occupied by a list that is represented by two 32-bit pointers plus an 8-bit tag. The MT separation of spaces does not require page and/or segment translation tables to determine the type of an object. Types are determined by accessing the tag associated with each object. Also unlike BIBOP, the MT separation of spaces allows the evolution of the computation to group related objects next to each other in the virtual address space instead of artificially imposing a structure on memory to hold objects by type. For example, in MT the members of a list, regardless of type, are stored on the same page. This fosters intra-list locality. The same list, under a BIBOP policy, would not have all of

its members stored in list space.

5.1 Heap Virtual Memory Performance

This section presents in Tables 5.1 through 5.10 the VM performance measurements taken on forty-five experiments performed on the MT heap. The odd numbered tables present the raw number of page faults and the relative difference between FIFO and LRU while the even numbered tables present the page fault rates for both replacement algorithms. The data is presented in relation to the size of main memory that is increased in quantum increments of 10 percent of the total number of pages used. On the odd numbered tables, the memory size column contains the actual size of the heap in pages. Every experiment ran with a page size of 512 S-expressions. Each S-expression is represented with 72 bits for a total page size of a little over 4KB which is a typical page size in modern computers. A global inspection of the tables reveals that the page fault rates range from 1.03e-5 to 6.335e-4 for FIFO and from 8.0e-6 to 4.876e-4 for LRU. The relative difference between FIFO and LRU ranges from -0.11 to 0.39 with the average around 7.38. The small difference in performance must be attributed to the MT allocation algorithm which attempts to minimize the intermingling of garbage and data and to the basic list operations being performed like creation and traversal. The small relative difference in performance once again suggests that it may not pay to implement LRU. All of the benchmarks display similar

Heap Size	FIFO	LRU	Rel. Diff.
10 (9)	100	95	0.05
20 (19)	80	78	0.03
30 (28)	68	67	0.01
40 (38)	58	57	0.02
50 (47)	48	47	0.02
60 (56)	39	38	0.03
70 (66)	29	28	0.04
80 (75)	19	19	0
90 (85)	9	9	0

Table 5.1: Heap Page Faults for **ins** in MT

Heap Size	FIFO	LRU
10	4.189e-4	3.980e-4
20	3.351e-4	3.268e-4
30	2.849e-4	2.807e-4
40	2.430e-4	2.388e-4
50	2.011e-4	1.97e-4
60	1.634e-4	1.592e-4
70	1.215e-4	1.173e-4
80	7.96e-5	7.96e-5
90	3.77e-5	3.77e-5

Table 5.2: Heap Page Fault Rates for **ins** in MT

fault rates with the exception of **eval** which is about an order of magnitude better than the other. For insertion-sort the relative difference between LRU and FIFO is very small with the maximum difference reaching 5%. Given that this sorting algorithm is mostly executing create-traverse memory operations when accessing the heap, the paging performance can not be improved through any paging algorithm. As the size of the list to be sorted grows larger (beyond K_H), we expect the performance of LRU and FIFO to become even closer. This follows from the fact that the intermediate

Heap Size	FIFO	LRU	Rel. Diff.
10 (8)	105	101	0.04
20 (15)	81	79	0.03
30 (23)	66	67	-0.01
40 (31)	54	53	0.02
50 (39)	46	45	0.02
60 (46)	39	38	0.03
70 (56)	23	25	-0.08
80 (62)	17	16	0.06
90 (69)	8	9	-0.11

Table 5.3: Heap Page Faults for qs in MT

Heap Size	FIFO	LRU
10	4.531e-4	4.358e-4
20	3.495e-4	3.409e-4
30	2.848e-4	2.891e-4
40	2.33e-4	2.287e-4
50	1.985e-4	1.942e-4
60	1.683e-4	1.64e-4
70	9.92e-5	1.079e-4
80	7.34e-5	6.90e-5
90	3.45e-5	3.88e-5

Table 5.4: Heap Page Fault Rates for qs in MT

lists being produced will be longer. Thus, their creation and traversal means longer periods of execution in which FIFO performs as well as LRU. The performance of FIFO and LRU is virtually the same on quicksort. FIFO actually outperforms LRU a couple of times with memory sizes at 30% and 90%. As with insertion-sort, neither FIFO or LRU will improve performance. Quicksort spends much of its heap acessing time in create-traverse memory operations. Unlike insertionsort, however, some of the simple lists created are traversed twice. For the list-traverse memory

Heap Size	FIFO	LRU	Rel. Diff.
10(11)	172	124	0.39
20(23)	130	124	0.05
30(34)	109	98	0.11
40(46)	91	83	0.10
50(57)	77	69	0.12
60(68)	60	56	0.07
70(80)	45	41	0.10
80(91)	32	23	0.39
90(103)	17	13	0.31

Table 5.5: Heap Page Faults for MM in MT

Heap Size	FIFO	LRU
10 (11)	5.256e-4	3.789e-4
20 (22)	3.973e-4	3.789e-4
30 (33)	3.331e-4	2.995e-4
40 (44)	2.781e-4	2.536e-4
50 (55)	2.353e-4	2.109e-4
60 (65)	1.834e-4	1.711e-4
70 (76)	1.375e-4	1.253e-4
80 (87)	9.78e-5	7.03e-5
90 (98)	5.19e-5	3.97e-5

Table 5.6: Heap Page Fault Rates for MM in MT

operation FIFO performs as well as LRU. Therefore, we also expect the performance of FIFO and LRU to become tighter as the list to be sorted becomes longer. Longer lists mean longer traversals which mean the system will spend more time performing activities for which FIFO performs as well as LRU. Matrix multiplication displays a performance in which the gap between FIFO and LRU is slightly looser, but not by much. Twice the relative difference reaches 0.39. At 10% it is clear that both algorithms are running with fewer pages than the minimum suggested by the knee

Heap Size	FIFO	LRU	Rel. Diff.
10	165	127	0.30
20	127	113	0.12
30	107	98	0.09
40	89	85	0.05
50	72	70	0.03
60	56	56	0.0
70	40	42	-0.05
80	26	28	-0.07
90	15	14	0.07

Table 5.7: Heap Page Faults for **fp** in MT

Heap Size	FIFO	LRU
10	6.335e-4	4.876e-4
20	4.876e-4	4.339e-4
30	4.108e-4	3.763e-4
40	3.417e-4	3.2636e-4
50	2.764e-4	2.688e-4
60	2.150e-4	2.150e-4
70	1.536e-4	1.613e-4
80	9.98e-5	1.08e-4
90	5.76e-5	5.38e-5

Table 5.8: Heap Page Fault Rates for **fp** in MT

of their parachor curves. Therefore, the problem here is that too few frames have been allocated to the heap at the evaluator node. At 80% the relative difference of 0.39 is not significant because the absolute difference at 9 is too small to affect performance. This exemplifies the fact that the significance of relative differences can not be ascertained without knowing the absolute difference. Just from the relative difference value it is impossible to tell if it is high due to a large absolute difference or due to the small magnitude of the amounts being compared. For find path the

Heap Size	FIFO	LRU	Rel. Diff.
10 (7)	157	151	0.04
20 (15)	71	60	0.18
30 (22)	58	52	0.12
40 (30)	48	44	0.09
50 (37)	41	37	0.11
60 (44)	32	30	0.07
70 (52)	24	22	0.09
80 (59)	17	15	0.13
90 (67)	9	7	0.29

Table 5.9: Heap Page Faults for **eval** in MT

Heap Size	FIFO	LRU
10	1.792e-4	1.153e-4
20	8.10e-5	6.85e-5
30	6.62e-5	5.94e-5
40	5.48e-5	5.02e-5
50	4.68e-5	4.22e-5
60	3.65e-5	3.42e-5
70	2.74e-5	2.51e-5
80	1.94e-5	1.71e-5
90	1.03e-5	8.0e-6

Table 5.10: Heap Page Fault Rates for **eval** in MT

tight performance between FIFO and LRU continues. A relative difference of 0.30 at 10% occurs before the knee of the curve generated by this data. Since the number of heap frames allocated at the evaluator node are too small for the experiment that generated this result we can disregard it. Once again, running the heap with fewer frames than those suggested by the knee of the curve hinders performance. It it also noteworthy that FIFO outperforms LRU at memory sizes of 70% and 80%. The experiments on **eval** suggest that the larger a program is the tighter LRU and FIFO

Memory Size	ins	qs	MM	fp	eval
10	334.02	311.30	379.91	380.74	285.87
20	563.61	471.67	656.36	633.52	469.54
30	770.34	655.36	908.57	886.21	630.48
40	1000.08	838.93	1184.29	1138.92	814.42
50	1206.79	1022.70	1437.00	1391.53	975.37
60	1413.50	1183.48	1689.69	1621.10	1136.31
70	1643.15	1413.02	1965.30	1873.65	1320.23
80	1849.81	1550.67	2217.76	2126.15	1481.16
90	2079.40	1711.39	2493.45	2378.60	1665.08

Table 5.11: Effective Access Times for the MT Heap under LRU

become. The absolute differences in page faults are so small that after 876,110 heap accesses one is forced to conclude that any differences are negligible.

5.2 The Argument for FIFO

The performance of FIFO and LRU is so close that based on the fault rates and the number of page faults it is impossible to conclude which page replacement policy should be used. Matters are really complicated by the fact that for a significant number of experiments performed FIFO does better than LRU or is within 10% of of LRU. Thus, we are forced to examine other metrics in order to determine the best heap page replacement algorithm for the MT system. As done to decide between MT and a system with a global pool of frames, we will look at the effective access times yielded by both LRU and FIFO. Tables 5.11 and 5.12 display the heap effective access times for FIFO and LRU respectively. The tables clearly establish that FIFO should be used as the heap page replacement policy for MT. The difference between FIFO and

LRU, however, seems to be very small for the heap. Certainly, the difference is smaller that the difference observed between MT system and the MT language with a global pool of frames from the previous chapter. In fact, the difference is almost constant at fifty-five (55) cycles. This is the difference between ma_{LRU} and ma_{FIFO} and should come as no surprise. Since the miss rates are so small the fault access time contribute little to the effective access time. Thus, we have that the effective access time mimics the memory access time so well. The above suggests two things. The first is that the MT allocation algorithm has solved the problem of heap locality for the current version of MT. That is, the MT heap exhibits good locality of reference and can no longer be considered a source of virtual memory inefficiency. The MT heap is virtual memory friendly. Page fault rates may still be improved via prefetching as suggested in the next section, but the gains will be small. Researching prefetching, however, is still a good idea because it remains to be seen if these rates can be maintained when, for example, first-class functions, closures, and continuations are introduced. It is unclear at this time if these objects can and/or will be incorporated to the MT heap or if they will be granted their own discrete space as the stack. Prefetching can also help to reduce page search time when multiple nodes and multiple levels of backing store are used by keeping pages that may be requested only one hop away from the evaluator. The second is that if one considers the heap locality problem solved for a sequential system then reducing the number of cycles it takes to service a hit is the

Memory Size	ins	qs	MM	fp	eval
10	279.10	256.34	325.91	326.72	231.32
20	508.56	416.63	601.07	578.75	414.61
30	715.23	600.20	853.66	831.27	575.51
40	944.96	783.84	1129.29	1083.80	759.43
50	1151.66	967.60	1382.00	1336.34	920.38
60	1358.39	1128.39	1634.61	1565.87	1081.30
70	1588.06	1357.84	1910.24	1818.37	1265.23
80	1794.71	1495.63	2162.91	2070.92	1426.17
90	2024.34	1656.30	2438.50	2323.54	1610.09

Table 5.12: Effective Access Times for the MT Heap under FIFO

only way to reduce the effective memory access time in an all-software system. Since memory access time is proportional to K_H reducing or eliminating it appears the most promising route to reducing effective access time. Given that temporal locality is also spatial locality for simple lists and other list-based structures built recursively from the empty list, hashing pages into a set of frames may pay big dividends. Hash tables have been used in the past in place of associative memories [59, 63]. Studying the impact of hashing pages in MT has been left for future research.

5.3 Heap Activity Counts

Another question that may be asked of a virtual memory system is how well is RAM being utilized. For MT, the question is how well is the set of heap frames being used at the evaluator node. For example, how many of the heap pages resident at the evaluator are accessed between faults? Efficient use of the space would show that most pages are accessed between faults. If too few pages are accessed then this signals

that the set of pages being held at the evaluator do nò hold the best approximation of the current working set or that too many frames have been allocated to the heap. If all pages are always accessed then it may be the case that the heap's working-set does not fit at the evaluator node. Margolin et al defined activity count as the average number of pages in main memory referenced at least once between successive page faults. To compute the activity count one needs to know the frequency distribution, $f(x)$, of how frequently are x pages accessed between faults. So, if the number of heap pages is K we have that the activity count is given by :

$$AC = \frac{100}{M} \cdot \frac{\sum x \cdot f(x)}{PageFaults}$$

where the sum is from $x = 0$ to K.

They proposed the AC metric to measure virtual memory performance. In their study they used three block-structured imperative programs and varied, among other things, the memory size and the paging algorithm. In their experiments with LRU, AC varied from $32.5to48.5$ with an average of 39.5. Their experiments with FIFO found that AC ranged from $24.5to41.5$ with an average of 34.4.

Tables 5.13 and 5.14 display the heap activity counts for LRU and FIFO respectively. As can easily be seen the percentage of pages accessed between faults is very low. Under LRU, AC varies from $0to70$ with an average of 13.4. Under FIFO, we have that AC ranges from $4to56$ with an average of 12.3. These numbers suggest that despite the high hit rates, heap memory at the evaluator node is not being used

Heap Size	ins	qs	MM	fp	eval
10 (9)	22	24	36	36	56
20 (19)	10	12	16	20	30
30 (28)	9	8	10	15	20
40 (38)	6	9	10	10	15
50 (47)	6	6	5	5	12
60 (56)	4	6	6	6	12
70 (66)	5	5	6	7	7
80 (75)	6	6	7	8	8
90 (85)	12	11	0	11	11

Table 5.13: Activity Counts for the MT Heap Using FIFO

in the most efficient manner. The MT allocation algorithm minimizes the amount of garbage intermingled with data and reduces the amount of garbage produced, but it does not prevent new list-based objects from becoming garbage quickly. These numbers, therefore, confirm what advocates of generational collectors have been stating all along. Most list-based objects become garbage soon, occupying valuable RAM space. To remedy this situation we do not advocate the use of a generational collector that will frequently interrupt the evaluator. Instead, we suggest that prefetching can play a role in MT. There is no reason not to substitute garbage pages that will never be accessed with pages that may be needed in the future. Chapter 7 will further discuss our ideas on prefetching.

5.4 Concluding Remarks

In this chapter we have studied the VM behavior of the heap. It has been established that for an all-software system the number of page faults and page fault rates do not

Heap Size	ins	qs	MM	fp	eval
10 (8)	22	24	45	45	70
20 (15)	10	12	20	20	36
30 (23)	9	8	10	15	24
40 (31)	6	9	10	6	18
50 (39)	6	6	6	8	12
60 (46)	4	6	6	11	14
70 (56)	5	4	7	6	0
80 (62)	6	6	8	8	9
90 (69)	12	10	0	11	14

Table 5.14: Activity Counts for the MT Heap Under LRU

suffice as metrics to choose a page replacement algorithm. It is necessary to look at the effective access time since memory access time is not constant. Our analysis revealed that for the MT heap FIFO should be used as the page replacement algorithm. This analysis is not valid only to our implementation on a network of transputers. The reason that FIFO was chosen was the small relative difference with LRU and the fact that FIFO took fewer cycles to service a hit. These observations would hold on any system MT were to be implemented on. If the performances of FIFO and LRU were to diverge and favor LRU much more then the story may be different. We have no evidence, however, to suggest that this will ever happen. Given that heap locality problem has been solved with the heap allocation algorithm, the important design parameters are, K_H, the size of the heap and/or the way pages are mapped to frames. In order to reduce memory access time K_H must be kept small or the page searching algorithm should not be proportional to it. We have suggested hashing pages as a way to reduce memory access time. Finally, we have briefly introduced activity counts as

a measure of how well heap frames at the evaluator node are being used. Despite the high hit rates, only a very low percentage of pages is accessed between faults. This, of course, is due to the high density of live data per page in MT. It does suggest, however, that prefetching may play an important role.

Chapter 6

The MT Stack

In this chapter we present the empirical data collected from the forty-five (45) experiments ran on the MT stack using our benchmarks. We first present data on the number of stack faults and stack fault rates for both FIFO and LRU along with the relative difference between the two. The second section presents an argument for the optimality of LRU for the MT stack. We then proceed to describe the MT stack page replacement policy that simulates LRU avoiding the overhead associated with LRU. Finally, in section three we present the activity count numbers for the MT stack.

As with the heap, each page could hold 512 S-expressions. Once again, this page size was chosen because it was a reasonable approximation to the 4KB page size used by many modern computers. In reality, there is no reason for the MT heap pages to be the same as the MT stack pages. Since the spaces are discrete and there is no hardware support, the heap and stack page sizes can differ as much as they like. Before running the experiments we present here we had no reason to make these sizes

different. Experimenting with different page sizes is left for future work.

6.1 Virtual Memory Performance

Tables 6.1 through 6.10 present the VM performance numbers for our benchmarks. The odd-numbered tables present the number of page faults and the relative difference between FIFO and LRU. The even-numbered tables present the page fault rates for both FIFO and LRU for each of the benchmarks.

The first observation that can be made from the tables is that when the number of frames allocated to the stack is too small the number of faults become excessive. This is collaborated by significantly high fault rates for memory sizes of 1 or 2 pages. Based on our data, we can safely conclude that the minimum number of frames that should be allocated to the stack is 3. This rule-of-thumb should be followed regardless of the page size. If the page size is reduced then the number of faults will increase if the number of stack frames is 1 or 2. If the page size is increased and the number of stack frame is less than 3, the fault rate will be reduced for small programs. These gains in fault reductions, however, will dissipate with programs that need more stack space are executed. As a the demand for more stack space increases 1 or 2 large pages will not suffice to avoid thrashing.

The performance gap between LRU and FIFO is much larger for the MT stack than for the MT heap. This is evidenced by the relative difference between the two

Memory Size (s)	FIFO	LRU	Rel. Diff.
10 (1)	8068	8068	0.00
20 (1)	8068	8068	0.00
30 (2)	483	322	0.50
40 (2)	483	322	0.50
50 (3)	169	106	0.59
60 (3)	169	106	0.59
70 (3)	169	106	0.59
80 (4)	83	19	3.37
90 (4)	83	19	3.37

Table 6.1: Stack Page Faults for **ins** in MT

Memory Size	FIFO	LRU
10 (1)	2.0925e-3	2.0925e-3
20 (1)	2.0925e-3	2.0925e-3
30 (2)	1.201e-4	8.35e-5
40 (2)	1.201e-4	8.35e-5
50 (3)	4.38e-5	2.75e-5
60 (3)	4.38e-5	2.75e-5
70 (3)	4.38e-5	2.75e-5
80 (4)	2.15e-5	4.9e-6
90 (4)	2.15e-5	4.9e-6

Table 6.2: Stack Page Fault Rates for **ins** in MT

page replacement algorithms. For insertion-sort FIFO produced up to 60% more faults than LRU. For quicksort the minimum relative difference is 50% while the maximum is 600%. By observing that the absolute difference for the different stack sizes is not always trivial, we can conclude that LRU is the policy to use for the MT stack while running quicksort.

For matrix multiplication the absolute difference between FIFO and LRU is extremely small. This difference is equal to one (1) for memory sizes from 40%*to*90%.

Memory Size (s)	FIFO	LRU	Rel. Diff.
10 (8,2)	465	310	0.5
20 (15,3)	294	246	0.20
30 (23,5)	190	156	0.22
40 (32,6)	164	126	0.30
50 (40,8)	118	76	0.55
60 (47,9)	107	56	0.91
70 (55,11)	72	24	2.0
80 (62,12)	42	12	2.5
90 (70,14)	14	2	6.0

Table 6.3: Stack Page Faults for qs in MT

Memory Size	FIFO	LRU
10	1.052e-4	7.01e-5
20	6.65e-5	5.57e-5
30	4.30e-5	3.53e-5
40	3.71e-5	2.85e-5
50	2.67e-5	1.72e-5
60	2.42e-5	1.27e-5
70	1.63e-5	5.4e-6
80	9.5e-6	2.7e-6
90	3.2e-6	5e-7

Table 6.4: Stack Page Fault Rates for qs in MT

The relative difference for the same memory sizes ranges from 13%*to*50%. Once again, we see that the relative difference can be a deceiving metric when studied out of context. For matrix multiplication, however, it is clear that the stack's VM performance is virtually the same for LRU and FIFO. It is only for matrix multiplication that the FIFO and LRU performance is so tight.

For fp the gap in performance between FIFO and LRU is bigger than for matrix multiplication or eval. As the size of main memory grows, however, the gap between

Memory Size	FIFO	LRU
10 (8,2)	50	100
20 (15,3)	66	66
30 (23,5)	40	40
40 (32,6)	32	32
50 (40,8)	24	36
60 (47,9)	22	44
70 (55,11)	27	54
80 (62,12)	24	56
90 (70,14)	21	98

Table 6.5: Stack Activity Counts for **qs** in MT

Memory Size (s)	FIFO	LRU	Rel. Diff.
10 (11,1)	13280	13280	0.00
20 (23,1)	13280	13280	0.00
30 (34,2)	15	10	0.50
40 (46,3)	9	8	0.13
50 (57,4)	7	6	0.17
60 (68,4)	7	6	0.17
70 (80,5)	5	4	0.25
80 (91,6)	3	2	0.50
90 (103,6)	3	2	0.50

Table 6.6: Stack Page Faults for MM in MT

FIFO and LRU dissipates. This type of behavior is what you would expect to see, but would miss if we only looked at fault rates. For example, for quicksort the relative difference between FIFO and LRU grows as main memory gets larger. How is this possible? The answer lies in observing that the rate at which LRU faults decreases is much larger than that of FIFO.

In closing, we can conclude that LRU's performance is better than that of FIFO's. Probably, the difference still is not large enough to use FIFO if the effective access

Memory Size	FIFO	LRU
10	2.2862e-3	2.2862e-3
20	2.2862e-3	2.2862e-3
30	2.6e-6	1.7e-6
40	1.5e-6	1.4e-6
50	1.2e-6	1.0e-6
60	1.2e-6	1.0e-6
70	9e-7	7e-7
80	5e-7	3e-7
90	5e-7	3e-7

Table 6.7: Stack Page Fault Rates for MM in MT

Memory Size	FIFO	LRU
10 (11,1)	100	100
20 (23,1)	100	100
30 (34,2)	50	100
40 (46,3)	66	66
50 (57,4)	50	50
60 (68,4)	50	50
70 (80,5)	60	60
80 (91,6)	64	96
90 (103,6)	64	96

Table 6.8: Stack Activity Counts for MM in MT

time is proportional to K_S. The difference, however, may be enough for a replacement policy equivalent to LRU that has a memory access time and an effective access time that is constant. The MT Stack Page Replacement Algorithm is described in the next section.

Memory Size (s)	FIFO	LRU	Rel. Diff.
10 (11,1)	9478	9478	0.0
20 (22,1)	9478	9478	0.0
30 (33,2)	663	442	0.50
40 (44,3)	315	164	0.92
50 (55,4)	17	10	0.70
60 (65,4)	17	10	0.70
70 (76,5)	5	4	0.25
80 (87,6)	3	2	0.50
90 (98,6)	3	2	0.50

Table 6.9: Stack Page Faults for **fp** in MT

Memory Size	FIFO	LRU
10 (11,1)	2.2922e-3	2.2922e-3
20 (22,1)	2.2922e-3	2.2922e-3
30 (33,2)	1.603e-4	1.068e-4
40 (44,3)	7.62e-5	3.966e-5
50 (55,4)	4.1e-6	2.4e-6
60 (65,4)	4.1e-6	2.4e-6
70 (76,5)	1.2e-6	10e-7
80 (87,6)	7e-7	5-7
90 (98,6)	7e-7	5e-7

Table 6.10: Stack Page Fault Rates for **fp** in MT

6.2 The MT Stack Page Replacement Algorithm

We start this section by describing the structure of activation records in MT in order to argue that LRU is optimal for the MT stack. We will then argue LRU's optimality. Finally, we will present the effective access time for FIFO and the MT page replacement algorithm.

Memory Size	FIFO	LRU
10 (11,1)	100	100
20 (22,1)	100	100
30 (33,2)	50	100
40 (44,3)	66	66
50 (55,4)	50	75
60 (65,4)	50	75
70 (76,5)	60	80
80 (87,6)	64	80
90 (98,6)	64	80

Table 6.11: Stack Activity Counts for **fp** in MT

Memory Size	FIFO	LRU	Rel. Diff.
10 (1)	57,170	57,170	0.00
20 (1)	57,170	57,170	0.00
30 (2)	27	18	0.50
40 (2)	27	18	0.50
50 (3)	7	6	0.17
60 (4)	5	4	0.25
70 (4)	5	4	0.25
80 (5)	3	2	0.50
90 (5)	3	2	0.50

Table 6.12: Stack Page Faults for **eval** in MT

6.2.1 Activation Records in MT

Activation records indicate the existence of and hold the state of a function that is being evaluated or has been interrupted to evaluate a recursive call or a call to another function. Every activation record contains control information, like the address of the previous activation record, the function that is being evaluated, and the program counter for the previously interrupted function, and the arguments for the function that it is associated with.

Memory Size	FIFO	LRU
10	2.5457e-3	2.5457e-3
20	2.5457e-3	2.5457e-3
30	1.2e-6	8.02e-7
40	1.2e-6	8.02e-7
50	3.12e-7	2.67e-7
60	2.22e-7	1.78e-7
70	2.22e-7	1.78e-7
80	1.33e-7	8.9e-8
90	1.33e-7	8.9e-8

Table 6.13: Stack Page Fault Rates for **eval** in MT

Memory Size	FIFO	LRU
10	100	100
20	100	100
30	50	100
40	50	100
50	66	66
60	50	75
70	50	75
80	60	100
90	60	100

Table 6.14: Stack Activity Counts for **eval** in MT

In MT, the stack grows from low addresses to high addresses. Every function that is called, whether user-defined or primitive, causes a new activation record to be stacked above the previously stacked activation records. Within an activation record, the address of the previous activation record is the first item to be stored and always occupies the lowest address in the record. When a function terminates the address of the previous activation record is read, the activation record is popped, and the result is pushed onto the stack. The result object occupies the same space the address of

the previous activation occupied. It will be important to know that it is the lowest part of an activation record that is last accessed.

6.2.2 LRU is optimal for the MT Stack

All the information a function needs is stored within its activation record. It follows, therefore, that a function need only access its own activation record for the information it requires to execute or to build the next activation record. Within an activation record there may be random access but there is no random access between them. In fact, the MT stack follows a strict last-in first-out discipline. This means that if $|largestactivationrecord|$ [1] $< STACKPAGESIZE$, then the LRU is an optimal page replacement policy [2]

Theorem 6.1. *Given that* $|largestactivationrecord| < STACKPAGESIZE$ *LRU is an optimal page replacement policy for the MT stack.*

Proof As the stack grows, stack memory is allocated linearly to hold each new activation record. This means that as the stack grows (towards higher addresses) higher numbered pages are being accessed. 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 page, respectively, held by the evaluator node at any given time. Since allocation and deallocation is always linear we have that all stack pages from $SP_{low}toSP_{high}$ are contained in the stack frames. Therefore,

[1] The parallel bars —— are used to represent the cardinality of its argument

[2] The optimal page replacement policy always picks as its victim a page that will never be referenced again or the page that will not be referenced again for the longest period of time [7].

there are only two cases under which a fault may occur: a request for page $low - 1$ or a request for page $high + 1$. Faulting on $high + 1$ means the stack is growing and that the least recently used page is the one numbered *low*. This page is also the memory resident page that will not be used for the longest period of time. All pages from *high* down to $low + 1$ must be accessed at least once before accessing the page numbered *low* because all activation records are smaller than a page. Faulting on $low - 1$ means that the stack is shrinking and that the least recently used page is the one numbered *high* This page is also the page that will not be accessed for the longest period of time. Even if the stack started growing immediately, all pages from *low* to $high - 1$ must be accessed at least once before accessing the page numbered *high*. **Q.E.D**

6.2.3 The MT Stack Page Replacement Algorithm

The proof that LRU is optimal for the MT stack suggests a page replacement algorithm that yields a fault access time that is not proportional to K_S. Instead, the access time is constant. Furthermore, it also suggests an algorithm that make memory access time also constant. That is, the stack effective access time does not have to be proportional to K_S. This allows us to make the number of stack frames as large as necessary without worrying that effective access time may deteriorate. The algorithm will also establish that it is no longer to time stamp stack pages at all.

The algorithm requires four variables:

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

The memory and faulting algorithm can be described as follows:

1. Load evaluator stack frames with pages $0..NUM-OF-STACK-FRAMES$. Set slow and flow to 0 and shigh and fhigh to $NUM - OF - STACK - FRAMES - 1$.
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 $flow+(requested-page-slow)$ by the appropriate displacement. Goto 2.
4. If $requestedpage > shigh$
   (a) Victim is slow in flow.
   (b) Swap slow out to backing store
   (c) Swap requested page in from backing store into flow.
   (d) Increase all four variables by $1 MOD NUM-OF-STACK-FRAMES$.

(e) Access the requested page and goto 2.

5. If *requestedpage* < *slow*

(a) Victim is *shigh* in frame *fhigh*.

(b) Swap *shigh* out to backing store.

(c) Swap requested page in from backing store into *fhigh*.

(d) Decrease all four variables by 1. If either *flow* or *fhigh* becomes negative set it to $NUM - OF - STACK - FRAMES - 1$.

(e) Access the requested page and goto 2.

6.3 Activity Count for the MT Stack

The next two tables present the activity count measurements taken on the MT stack. Both FIFO and LRU perform better on the stack than on the heap. LRU, however, is superior to FIFO in this regard. Nonetheless, there is work here for a prefetcher to do like to make sure that the pages at the evaluator are being accessed between faults. Algorithms with tree-like recursion such as quicksort will benefit the most according to the activity count metric.

Heap Size	ins	qs	MM	fp	eval
10	100	50	100	100	100
20	100	66	100	100	100
30	50	40	50	50	50
40	50	32	66	66	50
50	66	24	50	50	66
60	66	22	50	50	50
70	66	27	60	60	50
80	50	24	64	64	60
90	50	21	64	64	60

Table 6.15: Activity Counts for the MT Stack Using FIFO

Heap Size	ins	qs	MM	fp	eval
10	100	100	100	100	100
20	100	66	100	100	100
30	100	40	100	100	100
40	100	32	66	66	100
50	66	36	50	75	66
60	66	44	50	75	75
70	66	54	60	80	75
80	100	56	96	80	100
90	100	98	96	80	100

Table 6.16: Activity Counts for the MT Stack Using LRU

6.4 Summarizing Remarks

In this chapter we have presented empirical evidence that for the MT stack, the performance gap between LRU and FIFO is larger than the gap found for the heap. LRU is a better page replacement policy for the MT stack. In fact, we have proven that LRU is optimal for the MT stack.

The proof of LRU's optimality suggested a memory access algorithm and a page replacement algorithm that were not proportional to K_S. In fact, both memory

accesses and fault accesses can be performed in a constant amount of time. This means that the number of stack frames allocated at the evaluator node can grow without degrading performance. Furthermore, it means that the MT system requires at least two different page replacement algorithms, FIFO for the heap and the MT stack-page replacement algorithm for the stack. FIFO should not be used for the stack, because memory access time would still be proportional to K_S. This follows from observing that FIFO does not keep its pages ordered like LRU, thus, a memory resident page must me searched for before an access.

Finally, the activity count metric suggest that there is some role for a prefetcher to play in relation to the stack. The importance of this role, however, may not be as big as the one it can play with the heap.

Chapter 7

Future Work

Where do we go from here? At the beginning of this project, the author envisioned a full MT system for a small language. Ergo, we present work done on a prefetching algorithm and on a garbage collector algorithm. These algorithms have not been implemented. Their implementation may be held back until the MT language has been extended with, for example, first-class functions, closures, and continuations. The reason for this being that we consider the current version of MT virtual memory friendly. The impact a prefetcher and garbage collector can have is small and , therefore, the priority these components once held in our minds has dropped.

7.1 Adding First-Class Functions, Closures, and Continuations to MT

All modern functional languages support first-class functions. This means that functions can be passed in as arguments and returned as values from functions. Ma-

nipulating functions as data requires that the function code be associated with a context in which it can be evaluated. This context is provided by an environment that holds bindings for variables. Together the function's code and environment form the function's closure.

In all likelihood, closures will be added to MT as a new component. This compound data structure requires two types of space: code space and frame (or stack) space. The MT representation of a closure could consist of 2 32-bit pointers, one pointer for the code and another for the environment. The environment could be allocated in the MT heap, but this is likely to be a poor choice since it will be more efficient to keep the components of a closure close together swapping them in and out at the same time.

7.2 Adding More Nodes and Levels of Memory

The present implementation of MT only uses one node for heap backing store and one node for stack backing store. Adding more nodes and levels of backing store raises the question of what the topology these backing store networks should be.

The stack network could be a bidirectional ring in which pages move clockwise when the stack is shrinking and counterclockwise when the stack is growing. As suggested by the MT stack page replacement algorithm, pages should all be kept in linear order around the ring. Searching for pages will be fast given that the capacity

of each node will be known and all pages will be linearly ordered.

The heap network should attempt to maximize the number of nodes between the evaluator and the GC while minimizing distances between the two. This suggest a hypercube that is shared by heap and the GC.

7.3 Proposed Algorithms

7.3.1 The MT Prefetcher

Prefetching is a technique that is used to speed-up the execution of programs that employ some type of memory hierarchy. The idea is to bring into main memory (or the fastest access memory like a cache) those objects that will be accessed before they cause a fault. In this manner, system designers hope to curtail the amount of time the processor is idle waiting for a fault to be resolved. In a traditional sequential system this is rather difficult to do since there exist random access to memory and many levels of address indirection, and prefetching calculations force the primary computation to be temporarily halted. Random access to memory and address indirection make the correct prediction of where in the virtual address space the next object to be accessed resides an impossibility. In an attempt to be able to predict where the next object to be accessed resides the system could keep a trace of its access patterns. In this manner, the system would use past accessing patterns to predict future accessing patterns. This may curtail the number of faults incurred, but keeping track of the

tracing can be an expensive operation. Another attempt to reduce faults entails using the working-set model in paged systems. This model exploits locality of reference to reduce page faults, but will not work well if memory accesses do not exhibit locality, as is the case of programs that may randomly access memory.

We speculate that in MT, prefetching can be used effectively without requiring special interruptions of the primary computation. Furthermore, the system will be able to predict the set of heap pages the next access will be from. This follows from the manner in which lists are traversed. Lists do not support random access. Instead, accessing the nt^{th} element of a list requires traversing the first $n-1$ elements.

The constrained, non-random access of heap pages suggests a potentially efficient and effective way to implement prefetching of heap pages. We know that heap page faults can only occur on non-evaluator resident heap pages pointed to by cons-cells that are either heap or stack allocated at the evaluator node. Nodes used as backing store for heap/stack pages can create for each page they hold a "pages pointed to vector" (PPV). PPV*s* can be computed in parallel with the advancement of the primary computation. The computation of PPV*s* for pages that are evaluator resident must wait until these pages are swapped out in order not to halt the program's computation. A page's PPV is computed by inspecting every cons-cell on the page and computing the set of pages cons-cells point to. This set of pointed to pages constitute the elements of the page's PPV. There are no repeated elements within a PPV and

once computed we now know what pages can be requested by the evaluator when examining a given page. For fully allocated heap pages, *PPVs* do not change after they have been computed given that MT does not supports assignment. Therefore, heap *PPVs* need to be computed only once for each page in the current semi-space being used [1]. After a page, P_i, is recycled by the garbage collector and re-used by the MT allocator, a new PPV_i will need to be computed for the given P_i. Stack *PPVs* must be recomputed every time a stack page travels from backing store to the evaluator and back. For all allocated stack pages $j > 0$, stack PPV_j always includes stack page P_{j-1} since that is where the the previous activation records reside, thus, being a page that can be immediately requested.

Now, define Φ to be the union of *PPVs* that belong to both stack and heap pages that are resident at the evaluator. The only heap pages the evaluator can possibly request access to are those pages contained in Φ, a new unallocated heap page, or a page that should be in a stack PPV that has not been added yet (remember a stack page's PPV is only updated when it is swapped out). Φ can be used as an approximation of pages to be preftech.

Faults will not be completely eliminated since the set Φ may never be complete. The necessary pages may never be swapped to backing store. Nonetheless, for large list-based data structures Φ, complete or incomplete, can be used to prefetch pages

[1] This discussion assumes a semi-space garbage collector as described in the next section

when a fault occurs. The payoffs, although not significant for the effective access time, may almost eliminate faults given the activity counts presented in Chapter 5. The AC numbers suggest that lots of the garbage generated by the evaluator can be swapped out for live pages in backing store. The set Φ can further be used to reduce page search time by keeping the pages in Φ close (e.g. one hop away) to the evaluator.

7.3.2 The MT Garbage Collector

In this section, we present the proposed MT garbage collection algorithm. This algorithm has not been yet implemented. Instead of presenting a detailed specification and algorithm design, we will only present some of the salient characteristics of our proposal. The MT garbage collector (GC) is based on the sequential semi-space collector[2]. We chose the semi-space collector for two reasons. The first is that a compactifying collector which will collect objects in a manner that furthers the locality built into list-based structures by the MT allocator. The second is the separation of semi-spaces meant that there could be some degree of autonomy between the GC and the evaluator with each working in different semi-spaces.

We propose a mostly parallel GC algorithm which takes advantage of the fact that a DVM allows for easy management of replicated pages in an assignment-free system, the parallel servicing of GC and evaluator page faults, and the execution of the GC algorithm in parallel with the advancement of the computation. We use the term "mostly parallel" as done by Boehm et. al. [12] to mean that a small part of

the time the GC may have to operate in a stop-the-world fashion. We conjecture, as [12], that the duration of the stop-the-world phase will not take as long as would a stop-the-world collector.

In order to reduce contention for pages between the evaluator and the collector, pages will be replicated. This will allow each to have their own copy of the spaces that need to be collected. If the replication of a heap page P_i is delayed until it has been fully allocated then the GC can assume that it always has a faithful copy of P_i. This replication will not interfere with the evaluator since replication will only occur in backing store. In contrast, stack pages can not always be considered faithful. As long as, a stack page in backing store is not swapped into the evaluator node we can consider the page to be a faithful copy. This can easily kept be kept track of in MT's stack network (introduced in Chapter 1). This replication strategy will give the GC and the evaluator parallel access to the same data with no contention.

The stop-the-world phase could be eliminated if the semi-spaces are switched when stacktop drops below a certain threshold. That is, when the stack falls below the collected level switch the semi-spaces. This, however, may be too little to late since it is impossible to tell how long it will be before the stack shrinks to the desired point. Instead, the semi-spaces can be switched and allow the evaluator to temporarily have pointers to both semi-spaces. Any pointers into old-space can be updated as the pages containing old-space pointers are swapped-out to backing store.

7.4 Future Work and Final Commentary

This dissertation has proposed an architecture for an all-software distributed virtual memory tailored for a list-processing system. In MT, memory is divided according to the components needed to evaluate programs written in a functional language. This dissertation has focused on two of these components, the heap and the stack. Our initial thoughts were to continue these efforts by adding a garbage collector and a prefetcher to MT. The results presented in this study, however, have made us reconsider our goals. Given the fault rates that we have observed the importance of a prefetcher and a GC has been reduced. There is room for virtual memory improvement as suggested by the activity counts of both the heap and the stack, but the impact of these improvements on the overall virtual memory performance would be small.

Future efforts may focus on efficiently extending the MT language with first-class functions, closures, and continuations. The idea here is to create a more expressive language by efficiently adding components to the MT system. For example, how do we efficiently introduce a continuation handling mechanism given the current definitions of heap and stack in MT? Do the current data structures need to be redefined or will a new "continuation network" suffice? Similar questions can be asked of the introduction of closures and first-class functions.

The expansion of the MT principles to other higher-order languages such as Prolog,

or ML is also being considered. The transition to these languages, however, must be preceded by studying how they use memory. The programming system developer should not assume that dynamically allocated memory in these languages follow the same patterns found in MT. List creation, list traversal, and the repetitive creation-traversal cycle may not be as common as in functional languages.

After almost 40 years of research and effort, it is our sincerest hope that we have touched upon topics that will finally liberate functional languages from the stigma of inefficient. Furthermore, we hope that some clarity and understanding has been brought to the rather incompletely understood field of virtual memory. It may be the case that the best way to study virtual memory is to study the high-level memory access patterns of the components of the languages that are to be used for program evaluation.

Appendix A

The MT Language-Implementation Description

The MT language is a pure subset of Lisp implemented in Occam [28] on a transputer-based system [39]. MT includes the arithmetic operators (+, -, * and /), the relational operators (<, <=, >, >= and =), the boolean operators **and**, **or** and **not**, the list construction operator **cons**, the selectors **car** and **cdr** and the predicates null?, atom?, list? and symbol?. In addition, MT has a static and a dynamic number generator called **random** and **drand** respectively. They both take a positive integer n as input and return a random number in the range from zero (0) to $n-1$. The static random number generator will always produce the same stream of numbers every time the MT system is started. The dynamic random number generator is hooked to an Occam clock (i.e. TIMER) and is designed not to repeat the same stream of numbers even after restarting the MT system. We chose a small subset of Lisp in order to gain insight into how virtual memory is used by Lisp. Once a virtual memory efficient

Lisp has been implemented for this small subset, 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 [13, 38, 62].

We will describe the MT language by independently describing each important feature and comparing these features to what has been done in the implementation of other Lisp systems. The reader is warned that there is no "standard" implementation for Lisp. What some considered feasible others have completely disregarded. MT shares features with its ancestors, but as them does not completely fit any of the implementation molds that have been created. This will become clearer as the reader proceeds through this appendix.

A.0.1 Primitive Types

MT currently supports the following primitive types:

- Integers in the range $-2^{63}..2^{63}-1$
- Symbols an index into the symbol table [1]
- Boolean TRUE and FALSE
- NIL (special end of list symbol)

[1]The current version of the symbol table only stores information about the name of the symbol. One can interchangeably use the term string instead for this version of MT.

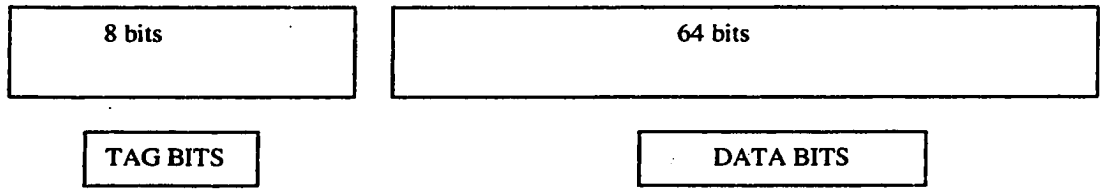

Figure A.1: Data Representation

- Lists (i.e. cons-cells)

The decision to initially only support these types was based on the observation that these types are the ones most frequently tested for (67%-96% of all tag tests) according to Shaw [64]. Efficiently implementing a system that supports these types becomes a priority given the reported numbers. Other types that may be considered important are not used as frequently. For example, *bignums* tend not to be used and arrays are tested for less than 1% of the time.

A.0.2 Representation and Tagging

All objects in MT are represented with 72 bits: 64 bits for the data and 8 bits for the tag. The data part is used to store a literal in the case of an integer or a boolean, an offset from the beginning of the symbol table in the case of a symbol, garbage in case of NIL and two 32-bit pointers (i.e. car and cdr) in case of a list. The car is held in the top 32 bits and the cdr in the lower 32 bits. The car and the cdr are always pointers to other objects that are already heap allocated.

As discussed in Chapter 2, in MT (primitive) objects are considered immediate

data and not references to objects. MT makes copies of an object instead of duplicating references to an object. This allows MT to exploit native Occam commands for object manipulation without having to dereference a pointer. This approach can only be successful if multiple copies of an object are considered the same. Certainly, two copies of an integer are the same as well as two symbols with the same name (as long as the symbol table does not have duplicate entries). Thus, equality can easily be checked with "=" (Occam's equality operator). Notice that there is no need to access the symbol table which is similar to testing pointer equality. Given MT's representation list equality can be tested in the same manner. Two lists are the same if the 64-bit integers that represents them are the same. The cost of comparing two integers is the same as comparing two pointers in classical Lisp representations. The MT advantage, however, is that it always has access to the car and cdr of a list without having to dereference a pointer.

The decision to use 8-bit tags was based on the observations that many more types will be supported in the future as the system grows and that Occam has built-in operators for characters. Thus, assigning a type is a simple character assignment in Occam. Tags are always kept with the data. That is, MT supports tagged data objects. A similar strategy was used in the MIT CADR system [27]. Unlike MT, MIT CADR combines the data and the tag into a single word (32 bits: 24 data bits and 8 tag bits). This lead, for example, to nonstandard number formats. In MT, the

separation of tags and data allows the direct use of the 64-bit integer operators native to Occam. Similarly, the storing of two 32-bit pointers into a 64-bit integer allows the direct exploitation of Occam's bit-manipulating operators for pointer extraction.

Many Lisp systems encode the type of an object in the pointer to that object (e.g. S-1 Lisp, NIL, Spice Lisp, Portable Standard Lisp and Xerox D-Machine InterLisp). In these systems the pointer must be extracted in order to reference the object. Similarly, the tag must be extracted to know the type of the object. In the case of immediate data some optimization is possible if, for example, tags are chosen judiciously to allow arithmetic operations without separating out the tag. MT evades these concerns by keeping the tag separate from the data. As mentioned above, this allows the use of primitive Occam operations on the tag and data directly. The use of data structures that can easily be used by the architecture of the machine is important since hardware support makes execution faster. This observation is consistent with the ones made by Henderson and Shaw [37, 64]. Furthermore, including the tag in a pointer to an object reduces the amount of virtual memory that is addressable.

The strict separation of tag and pointer can be removed if the tag is made part of the pointer. That is, a subset of address/pointer bits are used also as tags. These bits are those that identify the page number of a reference in the BIBOP scheme (used in MacLisp, Franz Lisp, Vax Common Lisp, Data General Common Lisp, NIL and Spice Lisp). The type of an object pointed to can be determined by looking

up the page number in a type-table. Although elegant, this scheme may not be virtual memory friendly [64]. Memory fragmentation across objects may cause lots of unneeded or unused page space to be memory resident. In order to overcome this shortcoming, pages can be divided into segments and each segment can only hold data of a certain type [22]. This requires an even bigger type table, but will reduce fragmentation. There is, however, no empirical evidence establishing the effectiveness of this solution. While designing MT we chose not to implement a BIBOP scheme because it does not allow objects of a different type to be intermingled in memory. Furthermore, the problem of fragmentation is unsolved and the size and management of type-tables may grow to become prohibitive. There is no clear way of deciding how big to make a segment or page to keep a reasonable balance between the size of the table and the amount of fragmentation.

A.0.3 Functions and Function Calling

Functions are stored in a function table. The details of how the function table is implemented are irrelevant since this initial study excludes focusing on function space. Functions are not first-class and there are no local function definitions. This means that closures are beyond the limits of this initial study. Function bodies are stored as strings with repeated blanks deleted and variables substituted by their parameter ranking. For example,

(define fff (x y z) (f x y z))

is stored as:

(f #1 #2 #3)

in the function table. During evaluation the # indicates that the value of a parameter is needed while the number indicates the displacement of the needed value from the base of the activation record.

The MT function calling mechanism is similar to those used by MIT CADR, Spice Lisp and VAX Common Lisp. When a function is called the base address of the previous activation record and the index of the function to be evaluated are pushed on the stack. The base address of this new activation record is the address of the function's index. The arguments to the function are then pushed on the stack in the same order they are evaluated from left to right. Thus, we are able to lookup parameter bindings directly on the stack without having to access a symbol table or other binding holding mechanism as an environment by simply using the displacement that follows a #. After all arguments are evaluated the function is applied. The indicates that the current activation record should be popped, the previous activation record restored and the result should be returned on the top of the stack. Notice that the result will be stored in the correct place if it is an argument to a function.

Unlike MIT CADR, MT does not limit the size of an activation record, thus, functions can have as many parameters as they like. Function application is triggered

when the last closing parenthesis is found. Different strategies are used by other systems. For example, LMI Lambda pushes the last argument to a special location triggering function application.

As described in chapter two, the biggest difference between MT and classical Lisp implementations is that the MT stack does not contain pointers. Literals are pushed onto the stack instead of a pointer to the literal. Pushing a cons-cell means pushing a car pointer and a cdr pointer onto the stack. This can be done in one Occam operation since each pointer is 32 bits allowing two pointers to be held in the 64-bit data field. The only other system that we know of that does not allocate heap space for fixnums, for example, is S-1 Lisp. Their motivation was to conserve both space and time for common operations. MacLisp heap allocates fixnums to a predefined area. Thus, allowing arithmetic operations to be done through pointer arithmetic without accessing the number itself.

Bibliography

[1] S.E. Abdullahi, E.E. Miranda, and G.A. Ringwood. Collection schemes for distributed garbage. *Proceedings of the International Workshop on Memory Management 1992*, LNCS 637:43–81, 1992.

[2] H. Abelson and J. Sussman. *Structured Interpretation of Computer Programs.* McGraw-Hill, 1990.

[3] Andrew W. Appel and Zhong Shao. An empirical and analytical study of stack vs. heap cost for languages with closures. Department of Computer Science,CS-TR-450-94,Princeton University, 1994.

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

[5] H.G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 15:280–294, 1978.

[6] D. Bartley and J.C. Jensen. The implementation of pc scheme. *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 86–93, 1986.

[7] L.A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5:78–101, 1966.

[8] 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.

[9] Daniel G. Bobrow and Daniel L. Murphy. Structure of a lisp system using two-level storage. *Communications of the ACM*, 10:155–159, 1967.

[10] D.G. Bobrow and D.W. Clark. Compact encodings of list structure. *ACM Trans. Prog. Lang. and Systems*, 1:266–286, 1979.

[11] D.G. Bobrow and D.L. Murphy. The structure of a lisp system using two level storage. *Communications of the ACM*, 10:155–159, 1967.

[12] H. Boehm, A. Demers, and S. Shenker. Mostly parallel garbage collection. *SIGPLAN Notices*, 26:157–164, 1991.

[13] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. *ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, June 1996.

[14] M. Chorbak and J. Noga. LRU is better than FIFO. *Algorithmica*, 23:180–185, 1999.

[15] Takeshi Chusho and Toshihiro Hayashi. Performance analyses of paging algorithms for compilation of a highly modularized program. *IEEE Transactions on Software Engineering*, SE-7:248–254, 1981.

[16] Douglas W. Clark. Measurements of dynamic list structure use in lisp. *IEEE Transactions on Software Engineering*, SE-5:51–59, 1979.

[17] D.W. Clark and C.C. Green. An empirical study of list structure in lisp. *Communications of the ACM*, 20:78–87, 1977.

[18] E.G. Coffman and L.C. Varian. Further experimental data on the behavior of programs in a paging environment. *Communications of the ACM*, 11:471–474, 1968.

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

[20] Peter Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6:64–84, 1980.

[21] R. Kent Dybvig. *The SCHEME Programming Language.* PTR Prentice Hall, Inc., 1987.

[22] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages. Department of Computer Science,Technical Report 400,Indiana University, 1994.

[23] R.K. Dybvig. *Three Implementation Models for Scheme.* PhD thesis, Ph.D. Dissertation, The University of North Carolina at Chapel Hill, Department of Computer Science, Chapel Hill, NC, Available as Technical Report TR87-011, 1987.

[24] S.E. Fahlman and D.B. McDonald. Design considerations for cmu common lisp. *In Peter Lee, editor. Topics In Advanced Language Implementation, MIT Press,* pages 137–156, 1991.

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

[26] Daniel P. Friedman and David S. Wise. Applicative multiprogramming. Department of Computer Science,Technical Report No. 72, Indiana University, 1978.

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

[28] John Galletly. *Occam 2.* Pitman Publishing, 1990.

[29] J. Gaudiot and L. Lee. Multiprocessor systems programming in a high-level sata-flow language. *Parallel Architectures and Languages Europe, Volume 1: Parallel Architectures (PARLE 1987), LNCS 258*, pages 134–151, 1987.

[30] B. Goldberg. *Multiprocessor Execution of Functional Languages.* PhD thesis, Ph.D. Dissertation, Yale University, Department of Computer Science, Available as Research Report yaleu/dcs/rr-618, 1988.

[31] B. Goldberg and P. Hudak. Alfalfa: Distributed graph reduction on a hypercube multiprocessor. *Graph Reduction: Proceedings of a Workshop, LNCS 279*, pages 94–113, 1986.

[32] R. Goldman, R. Gabriel, and C. Sexton. Qlisp: An interim report. *Parallel Lisp: Languages and Systems, Proceedings of the US/JAPAN Workshop on Parallel Lisp, LNCS 441*, pages 161–181, 1989.

[33] D. Grit. Sisal on message passing architectures. *CONPAR 90-VAPP IV, Proceedings of the Joint International Conference on Vector and Parallel Processing*, LNCS 441:721–731, 1990.

[34] W.J. Hansen. Compact list representation: Definition, garbage collection, and system implementation. *Communications of the ACM*, 12:499–507, 1969.

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

[36] S. Hekmatpour. *Lisp: A Portable Implementation.* Prentice Hall, 1989.

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

[38] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. *ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, 1990.

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

[40] Sandy Irani. Competitive on-line algorithms for paging and graph coloring. Technical Report TR-92-013, University of California at Berkeley, January 1992. Available from www.icsi.berkeley.edu/techreports/1992.abstracts/TR-92-013.html.

[41] T. Ito and M. Matsui. A parallel Lisp language PaiLisp and its kernel specification. *Parallel Lisp: Languages and Systems, Proceedings of the US/JAPAN Workshop on Parallel Lisp, LNCS 441*, pages 58–100, 1989.

[42] C.B. Jay. Personal communication, 1998. Seminar Presentation of The MT Architecture: Towards a Fast Lisp, Computer Science Seminar, Department of Computer Science, University of Technology, Sydney (UTS).

[43] K. Johnson, M.F Kaashoek, and A. Wallach. Crl: High-performance all-software distributed shared memory. *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, 1995.

[44] Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, Hanover, Massachusetts, 1984.

[45] Guy Lewis Steele Jr. Data representations in pdp-10 maclisp. *Proceedings of the 1977 MACSYMA Users' Conference*, NASA Scientific and Technical Information Office, 1977.

[46] Guy Lewis Steele Jr. Fast arithmetic in maclisp. *Proceedings of the 1977 MACSYMA Users' Conference*, NASA Scientific and Technical Information Office, 1977.

[47] R.H. Halstead Jr. New ideas in parallel lisp: Language design, implementation, programming tools. *Parallel Lisp: Languages and Systems, Proceedings of the US/JAPAN Workshop on Parallel Lisp, LNCS 441*, pages 2–57, 1989.

[48] Samuel N. Kamin. *Programming Languages: An Interpreter-Based Approach.* Addison-Wesley, 1990.

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

[50] D. Koorey. Tlisp: A concurrent lisp for the transputer. *SIGSAM Bulletin*, 26:15–23, 1992.

[51] Kai Li and Paul Hudak. A new list compaction method. *Software-Practice and Experience*, 16:145–163, 1986.

[52] INMOS Limited. *The Transputer Data Book.* Prentice Hall, 1988.

[53] INMOS Limited. *Transputer Instruction Set: A Compiler's Writer Guide.* Consolidated Printers, 1989.

[54] Stuart E. Madnick and John J. Sloan. *Operating Systems.* McGraw-Hill Book Company, 1974.

[55] Santos Martins. Parallel implementations of functional languages. *Fourth International Workshop on the Parallel Implementation of Functional Languages*, 1992.

[56] 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.

[57] Marco T. Morazán. Distributed virtual memory for Lisp, April 1996. Written Portion of Second Level Doctoral Examination.

[58] Marco T. Morazán and Douglas R. Troeger. The mt architecture: A proposal for fast lisp. *Proceedings of ADMI'97*, pages 149–154, 1997.

[59] R. Morris. Scatter storage techniques. *Communications of the ACM*, 11:38–44, 1968.

[60] Simon Peyton-Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32, 1989.

[61] W. Screiner. Parallel functional programming: An annotated bibliography. Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, Available via http://info.risc.uni-linz.ac.at:70/0/archive/reports/1993/93-24.html, 1993.

[62] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. Department of Computer Science,CS-TR-454-94,Princeton University, 1994.

[63] Alan C. Shaw. *The Logical Design of Operating Systems.* Prentice Hall, 1974.

[64] Robert A. Shaw. *Empirical Analysis of a Lisp System.* PhD thesis, Ph.D. Dissertation, Stanford University, Department of Computer Science, Computer Systems Laboratory, Available as Technical Report CSL-TR-88-351, 1988.

[65] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts.* Addison Wesley Publishing Company, 1994.

[66] Harold S. Stone. *High-Performance Computer Architecture.* Addison-Wesley, 1990.

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

[68] R.F Tsao, L.W. Comeau, and B.H. Margolin. A multi-factor paging experiment: I. the experiment and the conclusions. *Statistical Computer Performance Evaluation*, pages 103–134, November 1972. Proc. of a Conference held at Brown University.

[69] V. Turchin. *REFAL-5: Programming Guide and Reference Manual.* New England Publishing Co., 1989.

[70] Jon L. White. Address/memory management for a gigantic Lisp environment or, gc considered harmful. *Conference Record of the 1980 LISP Conference*, pages 119–127, 1980.

[71] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. *Proceedings of the 1995 International Workshop on Memory Management*, Springer-Verlag LNCS, 1995.

[72] 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.