Fine-Grain Interrupts and Atomic Heap Transactions

by

James W. Clark

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering in Electrical Engineering and Computer Science

and

Bachelor of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

© James W. Clark, MM. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly
paper and electronic copies of this thesis document in whole or in part.
Fine-Grain Interrupts and Atomic Heap Transactions
by
James W. Clark

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2000, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Electrical Engineering and Computer Science

Abstract
Automatic storage management (i.e., garbage collection) is popular in advanced languages, but it is complicated by the presence of concurrency. Specifically, during an allocation some heap invariants are not maintained. As a result, we cannot interrupt an allocation and run other heap-using code. There are many ways to handle heap operations in the presence of concurrency. This document presents a taxonomy of those methods, and presents two improvements over current technology.

The taxonomy uses a transactional view of heap management to define three methods of dealing with an interrupt inside an allocation. The interrupt system can run the allocation forwards to completion before handling the interrupt, it can undo the allocation and handle the interrupt with the intention of starting the allocation over, or it can side-step the problem by providing the interrupting thread with a separate heap. A new technique, fp-locking, is presented which improves the performance of the aborting class of solutions. Another new technique, static-arena allocation, is then presented which has clear advantages over the current techniques.

These techniques were implemented on the SML/NJ language system, and a discussion of the implementations is included to support system designers. Measurements were made comparing the performance of the techniques. The static-arena allocation method is shown to be very competitive with modern technology.

Thesis Supervisor: Olin Shivers
Title: Research Scientist
Acknowledgments

Without the assistance and support of many people, this research would not have been possible.

First and foremost, I’d like to thank Olin Shivers, my research supervisor, for his guidance and support. He is the inventor of the fp-locking technique which I present in this paper, and he has worked with me to develop the taxonomy and the static-arena technique. His support and inspiration has greatly assisted my education.

Roland McGrath has been an invaluable source of code and advice. His work on the SML/NJ run-time system made the implementations in this document possible.

Lal George and Allen Leung have provided excellent technical assistance for the MLRISC back-end used by SML/NJ.

I’d like to thank James Tetazoo for letting me have so much fun while I worked on this project and Charlene for waiting patiently while I finished it. To Albert and Robbie, I would like to give mad props. I’d also like to thank Ray for his comments on this document, and for teaching me Go.
# Contents

1 Introduction 6  
1.1 Contributions ...................................................... 6  
1.2 A transactional view of storage management .................... 7  
1.3 Contents ............................................................ 7  

2 Storage Management and Interrupts 8  
2.1 Heap transactions .................................................. 8  
2.1.1 Operations as transactions .................................... 8  
2.2 An interrupt-free system ........................................... 9  
2.3 The perils of interrupts ............................................ 9  
2.3.1 Interrupting an allocation .................................... 10  
2.3.2 Interrupting a collection ..................................... 10  
2.3.3 Interrupting elsewhere ....................................... 10  
2.4 A taxonomy of interrupt-handling techniques ................... 11  
2.4.1 Run to completion .............................................. 11  
2.4.2 Aborting the transaction ...................................... 13  
2.4.3 Separate heaps ................................................. 15  

3 Static Arenas 17  
3.1 Atomic fission ..................................................... 17  
3.2 Static-arena style .................................................. 18  
3.3 The anatomy of a static arena .................................... 18  
3.4 Scanning interrupted threads ..................................... 19  
3.5 Conclusions for static-arena style code ........................... 21  

4 Mutations 22  
4.1 The simplest kind of mutation .................................... 22  
4.2 Mutation in generational heaps ................................. 22  
4.2.1 Card marking .................................................. 23  
4.2.2 Store lists ..................................................... 23  
4.3 The transactional perspective .................................... 24  
4.3.1 Run to completion .............................................. 24  
4.3.2 Aborting the transaction ..................................... 25  
4.3.3 The need for a fine-grained solution ......................... 26  
4.3.4 Atomic fission: static store subvectors ..................... 26  

5 Implementing Transactions 29  
5.1 General issues ...................................................... 29  
5.1.1 Detecting transactions and dangerous code ................ 29  
5.1.2 Registers ...................................................... 31  
5.1.3 Spills .......................................................... 31  
5.1.4 Heap availability .............................................. 32  
5.2 Atomicity for static-arena allocations ............................ 32
5.2.1 Complex solutions ........................................... 33
5.2.2 The simple solution ........................................ 33
5.3 Deferring interrupts ........................................... 33
5.4 Aborting transactions ......................................... 33
5.5 Static arena style ............................................ 34
  5.5.1 The arena size ........................................... 34
  5.5.2 The partially-initialized object address .................. 34
  5.5.3 The arena frontier ....................................... 35
6 Fine-grain Interrupts In SML/NJ ................................ 36
  6.1 Stock SML/NJ .................................................. 36
    6.1.1 Safe points and generational collecting ................ 36
    6.1.2 MLRISC .................................................. 36
  6.2 The abort technique on the Sparc architecture ................ 37
    6.2.1 Allocation ............................................... 37
    6.2.2 Annotations ............................................. 37
    6.2.3 Mutations ............................................... 38
    6.2.4 Results ................................................ 38
  6.3 Static arenas on the IA32 architecture ....................... 39
    6.3.1 Allocation ............................................... 39
    6.3.2 Registers ............................................... 40
    6.3.3 Annotations ............................................. 40
    6.3.4 Results ................................................ 42
  6.4 Static arenas on the Sparc architecture ....................... 42
    6.4.1 Annotations ............................................. 42
    6.4.2 Results ................................................ 43
7 Future Work ..................................................... 45
  7.1 Better annotations ........................................... 45
  7.2 Selective register partitioning .............................. 45
  7.3 Other kinds of collectors .................................... 45
  7.4 Operating system integration ................................ 46
  7.5 Finer measurements ......................................... 46
  7.6 Predicate registers ........................................ 46
8 Conclusions ...................................................... 47
Chapter 1

Introduction

Automatic storage management (i.e., garbage collection) is popular in advanced languages. Systems which use garbage collection have to be careful about how interrupts are handled, particularly if those interrupts can cause other heap-using code to be executed. This can happen when the system allows concurrency, or when the system allows interrupt handlers to be written by the programmer.

A language system designer usually develops his or her own method for implementing heap operations which tolerate concurrency. That method is tailored to match the system, depending on what costs or complexities are important to the designer. I will introduce a classification system for those solutions, based a transactional view of heap management. I will then describe a new method for handling interrupts in a garbage-collecting system and compare it to current practices. These techniques apply to most modern garbage collecting language systems including Smalltalk, Java, Scheme and SML/NJ.

I will focus on stop-and-copy garbage collectors. There are many alternative collection schemes, but stop-and-copy is a particularly popular and relatively high performance method [9]. While some of the ideas may apply generally, my examples and discussion draw only from stop-and-copy systems. Parallelism is also not discussed in detail in this document. Interrupts are often used to implement concurrency in modern systems, but this is not to be confused with true parallelism. I will focus on systems that support concurrency and execute on a uniprocessor.

1.1 Contributions

The contributions of this document are a taxonomy of methods for handling interrupts in a garbage-collecting system and the introduction of a new technique with better performance.

The taxonomy is meant to be useful not only for classification and analysis of existing systems, but also as a guide for implementing new systems. After reading this document, a system designer should be able to more easily decide which method will work best for his or her particular system, and should be able to implement that method without having to rediscover the associated issues. I will also call attention to specific weaknesses present in each design.

When an allocation is in progress, the heap is inconsistent. An interrupt cannot be serviced with an inconsistent heap. There are three ways to handle the situation: We can complete the operation and then service the interrupt, we can undo the changes made by the operation and plan to start it over again when the thread is resumed, or we can side-step the problem by giving the new thread a separate, consistent heap. These three methods are called run-to-completion, abort, and side-step.

The method introduced, which I call the static-arena method, enables fine-grained interrupts with low run-time and interrupt-time costs. This is important for systems with frequent interrupts. Additionally, the static-arena method avoids many of the weaknesses of current systems.

Additionally, I will point out another new technique, fp-locking, which improves the abort class of solutions by reducing the amount of interrupt-time work needed.
1.2 A transactional view of storage management

The heap is a resource used by many different agents in a language system. Those agents modify the heap through operations such as allocation or collection. The operations are distinct, and during the course of an operation the heap may appear invalid or inconsistent to the other agents. For that reason, we consider those operations to be transactions.

Since heap invariants are violated in the middle of a transaction, we must find a way to keep transactions atomic with respect to one another. How a compiler designer implements those transactions to maintain heap consistency is the basis of this document.

Simply put, interrupts are not allowed in the middle of a transaction. In some cases this means delaying the interrupt until the transaction is complete. In other cases, this means acting at interrupt time as if the transaction were never begun in the first place, and then restarting the transaction when the thread is resumed. The third, less common method attempts to avoid the problem by altering the heap itself so that transactions do not need to be atomic. This third technique does not completely solve the problem, as we will see, but it has a potential to improve performance when used in conjunction with other techniques.

I present in this document a technique which manages atomicity in a new way. It starts by dividing the allocation into two parts, only one of which must be atomic. Providing general atomicity for a sequence of instructions can be complex, but in this case the atomic sequence is so small that it can be treated using simple techniques.

The techniques in this paper (old and new) all point out an important issue: Garbage collection on a modern system requires the integration of the compiler with the run-time system (or operating system). The compiler and run-time system must collaborate at least on the implementation of allocation to make sure that interrupts can be safely handled. For the more advanced interrupt handling techniques, the compiler actually needs to communicate specific information about user code to the run-time system.

1.3 Contents

The second chapter discusses the transactional view of heap management and then describes the taxonomy of methods for allowing interrupts. The taxonomy is built by first examining the problem and then classifying solutions using simple characteristics. It does not go into very many implementation details, but rather discusses the techniques broadly. Chapter 3 introduces the new static-arena method, which is meant to improve upon the current techniques, particularly for systems which expect frequent interrupts. Those two chapters contain the main points and contributions of this document. The remainder serves to provide more details and to discuss a particular series of example implementations I designed.

Chapters two and three discuss methods of allocation—for simplicity discussion of mutation is postponed until chapter 4. The implementation details and machine-specific issues are introduced in chapter 5. Chapter 6 describes three experiments performed on the SML/NJ language system. Chapter 7 provides some ideas for future research, and chapter 8 summarizes the content of this document and presents the conclusions of my research.
Chapter 2

Storage Management and Interrupts

2.1 Heap transactions

In a language system, the heap is a resource shared by multiple threads. Those threads have two fundamental ways of manipulating that resource: allocate and collect. These operations allow the program to easily manage large amounts of data. Although the heap contains this data in a somewhat arbitrary manner, it does have specific invariants. Those invariants govern the way threads view the heap, and the way they perform allocate and collect operations.

Outside of the fundamental operations the heap is consistent. Inside, however, is another story. When an allocate or collect operation is in progress some of the heap invariants may be violated. The heap operations must therefore be performed atomically. Although the methods of achieving this atomicity may differ, the principle is generally the same: We cannot simply perform one operation when another is still in progress.

The issue arises primarily when a user thread can be interrupted by another heap client (such as another thread or the garbage collector). This occurs in concurrent systems which make use of thread-switching, and in a system which allows users to write interrupt handlers.

Let’s define some useful terminology. A user program may be composed of multiple threads which can execute concurrently. User code refers to code written in the language in question, unlike the run-time system which may be written in any language. When an interrupt occurs the processor eventually branches to the interrupt system, the part of the run-time system which deals with interrupts. The interrupt system may return to the interrupted thread, or may activate another thread. An interrupt handler is user code written to be called by the interrupt system when a particular interrupt occurs.

In this document I will be discussing a stop-and-copy garbage collection system, a popular high performance style of heap management. Although the ideas below may be applicable to other types of collectors such as mark-and-sweep or concurrent collection systems [9], this document’s scope is restricted to stop-and-copy.

As for observation and mutation of the heap, one can also consider them operations, but for now I will focus on allocation and collection, and discuss mutation on its own later. Observation is simpler than the other operations, and since it can usually be implemented in only one instruction, it is already atomic.

2.1.1 Operations as transactions

I view the two basic heap operations as transactions, much like database transactions or bank transactions. In allocation, the thread modifies the heap as a whole to create a new structure. It turns out that the new structure looks a lot like the old one, since the previously allocated sections
of the heap do not change. Regardless, it is important not to think of the allocation as an operation on a part of the heap. The reason for this distinction is that allocation modifies the heap frontier—the border between used and unused heap—which is a vital part of heap state. In a collection transaction, the whole heap is modified again, now resulting in an entirely new object. Transactions are not meant to be performed simultaneously, and in the middle of a transaction the heap might not be valid. This transactional view is the basis of this document.

2.2 An interrupt-free system

To help us understand the complications imposed by interrupts, we will first consider how an interrupt-free system would implement heap transactions. With this as a foundation, we will discover what problems arise with the introduction of interrupts and begin building solutions to those problems.

The heap is a large sequential segment of memory, with three important points: the beginning, the end (or limit), and the frontier. Allocations are performed sequentially from the beginning of the heap, filling up space until the end. The frontier pointer, or fp, indicates where the unused portion of the heap begins. The limit pointer, or lp, indicates the end of the heap.

The amount of free space left on the heap can be measured as the difference between the limit pointer and the frontier pointer. Before an allocation or group of allocations is performed, we check that there is enough room left on the heap by comparing the total size of the impending allocations to that difference. This is called a heap limit check. If a garbage collection is needed, then we call the garbage collector, indicating which registers contain live pointers.

| block: compare fp+8, lp | Check for 8 bytes of free heap |
| if > then goto gc | If necessary, prepare to call |
| gc: gcarg ← register mask | the garbage collector |
| call collector | Send live register mask to gc |
| goto block | Call the garbage collector |
| | Return to the thread |

Since allocation is ubiquitous, it is open-coded in the program. That is, rather than performing a procedure call to an 'allocate' subroutine, we insert the allocation code directly into the code stream. Allocation has the following basic parts:

1. Store data onto the heap (offset from the frontier pointer).
2. Assign the new object pointer (relative to the frontier pointer, usually equal to the frontier pointer).
3. Advance the frontier pointer.

| fp[0] ← a | Step 1: Store object elements onto heap |
| fp[4] ← b | |
| p ← fp | Step 2: Assign the new object pointer |
| fp ← fp+8 | Step 3: Advance the frontier pointer |

The last step can even be postponed until a later allocation in the block, at least until the next instruction which is a branch destination or until a collect transaction is performed. As long as we are assuming no interrupts, the order of these instructions is not important, and they need not be generated sequentially in the program.

2.3 The perils of interrupts

The previous section outlined a simple implementation of the two fundamental heap transactions based on the assumption that the program is not interrupted. In a modern run-time system, we
expect interrupts to occur for a variety of events such as a key press or a page fault. If no heap transactions occur during the course of the interrupt, then we might expect that they will have little effect on the user code. The heap will remain consistent across the interrupt and so the program's transactions are still atomic with respect to the heap. However, most modern languages wish to provide the user with the ability to write his or her own handlers which can do anything—including perform a heap transaction.

The interrupt system will need to save registers so that the interrupt context can be restored when the thread is resumed. But if a garbage collection is called during the interrupt, then the collector will need to know what those registers were, and which ones contained pointers to heap objects so that it can consider them as roots to be traced and replace them with the new copied values.

### 2.3.1 Interrupting an allocation

During the course of an allocation, the above scheme relies upon the consistency of the frontier pointer and of the segment of the heap which it is allocating. The fp must remain constant during the first two steps of allocation (initialization and object pointer assignment). Each instruction in that sequence uses an offset to the fp, and we assumed that the fp’s value doesn’t change. The portion of the heap which we are allocating must also remain consistent. During the allocation the code assumes exclusive ownership of that piece of storage in order to reliably generate the object. Furthermore, at the heap limit check—before the allocation occurred—the program picked out a specific segment of heap that it wants. Moving the fp before the allocation and not during it (as an allocation during an interrupt might do) would guarantee a consistent fp, but it doesn’t guarantee enough space on the heap to complete the allocation.

| 1 | compare fp+8, lp | Heap limit check |
| 2 | if > then goto gc |
| 3 | fp[0] ← a | Initialize heap object |
| 4 | fp[4] ← b |
| 5 | p ← fp | Assign object pointer |
| 6 | fp ← fp+8 | Advance fp |

In the above example, consider an interrupt at instruction 3 (i.e., before instruction 3 is performed). There is the possibility that the eight bytes are no longer available on the heap, and that a collection is necessary.

An interrupt at instruction 4 or 5 might overwrite the stored values, and move the fp, causing the pointer p to point to an invalid heap location. Likewise, an interrupt at instruction 6 might also overwrite the stored values, invalidating the object.

### 2.3.2 Interrupting a collection

As for collection transactions, we must avoid interrupting them at all costs. Between the start and the end of a collection, the heap is usually in a complete mess which user code cannot be expected to understand or operate on.

### 2.3.3 Interrupting elsewhere

So where can we allow interrupts? If the interrupt occurs somewhere outside of the two basic heap transactions (so that they are atomic), then it can be allowed. The heap will be consistent upon interrupt, and hopefully the heap will be returned to a consistent state when the thread is resumed.

Not all context needs to be unchanged during the interrupt. In fact, most dedicated registers (such as fp, lp, . . .) can and will change value. They are considered part of the heap state, and are not relied upon by the program except during a heap transaction. With interrupts outside of heap transactions we do not need to worry about heap consistency violations. However, such interrupts
are not trivial. Register liveness information must be available to the garbage collector, in case a collection is needed during the interrupt.

2.4 A taxonomy of interrupt-handling techniques

There are many ways to approach the challenge of building a system which supports interrupts. Using a transactional view, I will present in this section a means to classify the solutions found in many language systems today into three major classes.

When an interrupt occurs inside of an allocation the heap may be inconsistent, so we cannot allow another thread to begin. We cannot service the interrupt in the middle of the heap transaction. There are three ways to handle this situation:

- Complete the transaction, and then service the interrupt.
- Undo the work done by the transaction, and start the transaction over again when the thread is resumed.
- Give the interrupting thread an entirely different heap.

These techniques are called run-to-completion, abort, and side-step (or also forwards, backwards, and sideways).

2.4.1 Run to completion

In the commit, or run-to-completion method, we delay the interrupt until later, when we are out of the ‘atomic’ instruction sequence. Many interrupts do not rely on extremely short service latencies, so delaying them for a few cycles or even longer (though perhaps distasteful) is not fatal.

In this technique, we first define two different types of instructions. Some instructions are dangerous, i.e., interrupts are difficult to handle there. Specifically, an instruction is dangerous if the heap is inconsistent at that point in the thread, because important invariants are not met. During an allocation, when some heap objects may not be complete, the instructions are considered dangerous. Similarly, if the garbage collector cannot determine which registers are live at a particular instruction, then it would not be able to trace all live objects. If register-liveness information is not available for an instruction, it is dangerous. Instructions which are not dangerous will be called safe.

The definition of dangerous is a bit vague because it depends upon implementation. For example, if a particular language system allows derived pointers, but does not have a means for the garbage collector to properly trace them, then an interrupt while a derived pointer is active could cause trouble. In that system, we would define dangerous to include instructions where derived pointers are active.

The run-to-completion technique is simple. If an interrupt occurs during a dangerous code sequence, then we do not immediately service it. Instead we postpone the interrupt until the program enters safe code. If an interrupt occurs during a safe code sequence, then we can service it immediately. Because the interrupt system returns control to the thread when it defers an interrupt, we must rely only upon that thread to check for deferred interrupts in safe code. This means the compiler must insert polls in safe code sequences. Those polls check for deferred interrupts and then call the run-time system to service them.

In this technique, safe instructions need register-liveness information, while dangerous instructions do not. Furthermore, we need some way to distinguish between safe and dangerous code in the interrupt system (an example of compiler/run-time system interaction).

Dangerous points

The T programming environment [7, 6] used a technique called dangerous points. In this technique the allocations are close-coded (i.e., called from the program as a system procedure) and register-liveness information is made available at every user-code instruction. As a result, there is no dangerous code in the program itself. The dangerous code exists only in the run-time system.
T was able to find the live registers at any instruction because it partitioned the registers into two sets. One set contained only object pointers, while the other contained only integers or other values which the collector does not trace. If an interrupt occurred at any user-code instruction, the collector assumed that all object pointers in registers were live.

If an interrupt occurs during the allocation procedure, it is deferred until just before that procedure returns to the program. T detected such interrupts by comparing the interrupted PC to the address of the allocation procedure. The following example shows a simple cons operation in dangerous-points style.

<table>
<thead>
<tr>
<th>compare fp+8, lp</th>
<th>Heap limit check</th>
</tr>
</thead>
<tbody>
<tr>
<td>if &gt; then goto gc</td>
<td>Set up procedure arguments</td>
</tr>
<tr>
<td>r1 ← a</td>
<td>Allocate a cons object</td>
</tr>
<tr>
<td>r2 ← b</td>
<td>Copy object pointer</td>
</tr>
<tr>
<td>call cons</td>
<td></td>
</tr>
<tr>
<td>p ← r1</td>
<td></td>
</tr>
<tr>
<td>safe:</td>
<td></td>
</tr>
<tr>
<td>cons:</td>
<td>Initialize heap object</td>
</tr>
<tr>
<td>fp[0] ← r1</td>
<td></td>
</tr>
<tr>
<td>fp[4] ← r2</td>
<td>Assign object pointer</td>
</tr>
<tr>
<td>r1 ← fp</td>
<td>Advance fp</td>
</tr>
<tr>
<td>fp ← fp+8</td>
<td></td>
</tr>
<tr>
<td>if ≠ then goto intsys</td>
<td>Check for deferred interrupt</td>
</tr>
<tr>
<td>return</td>
<td>If so, activate the interrupt system</td>
</tr>
<tr>
<td></td>
<td>Return</td>
</tr>
</tbody>
</table>

In the close-coded cons routine, the interrupts are deferred until the deferred interrupt check (at safe). The interrupt system is able to determine if interrupts should be deferred by checking the interrupted PC value. If the PC was between cons and safe then the interrupt system defers the interrupt and sets the iflag, otherwise it services the interrupt.

**Dangerous Points in Modula-3**

Diwan et al. used dangerous points in the Modula-3 language system to support garbage collection [2]. Just as in T, all heap transactions were close-coded. Their implementation allowed interrupts at every instruction which was not part of a heap transaction. However, they only allowed garbage collections at specific locations, called ‘gc-points’. When a thread initiated a garbage collection, all interrupted threads which were not at a gc-point were resumed, and blocked on a gc-point.

In this variant of the dangerous-points technique, the garbage collector does not need supporting information (such as register liveness, see section 5.1.2) for every instruction in the program. This saves space in the tables containing that information.

**Safe points**

The dangerous-points technique requires that we close-code allocation. The SML/NJ programming environment [1] uses a different kind of run-to-completion technique called *safe points*. In the safe-points technique allocation is open-coded, and every instruction is considered dangerous. The user code must perform explicit periodic polls to check for deferred interrupts. If an interrupt is deferred, the program must prepare the heap so that all invariants are met.

SML/NJ performs those checks once per basic block. By doing so, it can make use of a clever optimization: The heap limit check and the deferred interrupt check can be combined. When an interrupt occurs, the interrupt system immediately prepares to defer it, and sets a flag somewhere (as in the dangerous-points method above) so that the program can detect the deferred interrupt. SML/NJ cleverly uses the *heap limit pointer* as the flag, which allows the compiler to combine the heap limit check and the deferred interrupt check.

The interrupt system sets the *heap limit pointer* to zero, guaranteeing that the next heap limit check will fail. When the heap limit check fails, the garbage collector is called. Instead of immediately
performing a collection, it checks first to see if an interrupt was deferred. If so, it resets the heap limit pointer and handles the interrupt.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>compare fp+8, lp</code></td>
</tr>
<tr>
<td>2</td>
<td><code>if &gt; then goto gc</code></td>
</tr>
<tr>
<td>3</td>
<td><code>fp[0] ← a</code></td>
</tr>
<tr>
<td>4</td>
<td><code>fp[4] ← b</code></td>
</tr>
<tr>
<td>5</td>
<td><code>p ← fp</code></td>
</tr>
<tr>
<td>6</td>
<td><code>fp ← fp+8</code></td>
</tr>
</tbody>
</table>

The example looks like code which isn’t designed to handle interrupts, because in the safe-points method the work is done entirely by the interrupt system. If an interrupt occurs at instruction 1 or instructions 3–6 then the interrupt system saves the interrupt information, sets lp to zero, and resumes the thread. When the next heap limit check fails garbage collector is called. The garbage collector notices that lp is zero, restores it, and then handles the interrupt (note that some time after instruction 6 another block begins and that block does a heap limit check).

If an interrupt occurs at instruction 2 then a heap limit after instruction 6 will be the one that detects and services it.

**Page faults**

The safe-points technique suffers from an important problem. Page faults are not interrupts which can be postponed. The very nature of a page fault is such that the interrupted instruction and those that follow it are impossible to execute, because the data needed is not in memory. The fault incurs a delay while the data is loaded from disk into memory. We would prefer to hide that delay by running some other thread until the interrupted thread can be restarted.

In the dangerous-points technique, allocations are close-coded and page faults are less of a problem. The allocation procedure is so frequently accessed that it is expected to remain in memory for a long time. If it happens to be swapped out, then no thread can run (all threads require allocation). We cannot hide the delay of loading the page, but since most code is safe we expect page faults in dangerous code to be rare.

However, in safe-points style page faults in dangerous code are much more frequent. Since the code is dangerous, no other threads can run while the page is being loaded. The system must simply shut down for milliseconds.

**2.4.2 Aborting the transaction**

Run-to-completion techniques deal with interrupts by first committing the transaction before servicing the interrupt. The abort technique deals with heap consistency in an entirely different way. First the work done by the transaction is undone, returning the heap to a consistent state. The interrupt is then serviced with the intention of restarting the transaction when the thread is resumed.

Consider this basic allocation sequence:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fp[0] ← a</code></td>
<td>Initialize heap object</td>
</tr>
<tr>
<td><code>fp[4] ← b</code></td>
<td></td>
</tr>
<tr>
<td><code>p ← fp</code></td>
<td>Assign object pointer</td>
</tr>
<tr>
<td><code>fp ← fp+8</code></td>
<td>Advance the frontier</td>
</tr>
</tbody>
</table>

The idea is very similar to aborting transactions in a database or file system. We back out of the transaction in progress, and then service the interrupt. The fascinating thing about the above example is that it requires no work to undo any of the instructions. The fp points in the same place throughout the transaction, and the instructions only store into locations after the frontier. Since we make no assumptions about the space after the frontier (it is uninitialized) these instructions need not be undone. We can service the interrupt immediately, and simply abandon the transaction.
Since the registers do not change during this sequence, we can also begin the transaction again with no effort.

The interrupt system needs a way to detect if the interrupt occurred within a transaction. Old versions of SML/NJ used the abort technique. SML/NJ’s method for detecting allocations was to scan backwards through the interrupted code and recognize the beginning and ending instructions of the transactions. If the interrupt system locates the first instruction of an allocation, then it knows to resume there. This method is somewhat time-consuming and difficult to implement, particularly because it is very machine dependent.

Flags and fp-locking

Placing a flag in memory and marking it during the transaction might be a simpler and less costly way to detect transactions, but it adds two memory operations per allocation to the user code:

| tflag ← 1 | Begin transaction |
| fp[0] ← a | Initialize heap object |
| fp[4] ← b | Assign object pointer |
| p ← fp | Advance the frontier |
| tflag ← 0 | End transaction |

Those operations would be quite costly. In addition, clearing the flag after advancing the frontier destroys the previous free-abort property. If we allow an interrupt after the $fp ← fp+8$ instruction then it has to be specifically undone. Previously we did not have to undo this work because the $fp ← fp+8$ instruction was the last instruction in the transaction, and interrupts after it were safe.

There is a more clever way to speed up transaction detection without going to memory and without requiring abort work. On all garbage collecting systems, the frontier is always word-aligned (at least), which leaves the lowest two bits constantly zero. We can use one of those bits as a flag and create a lightweight detection mechanism for the interrupt system:

| fp ← fp+1 | Set the flag |
| fp[-1] ← a | Initialize heap object |
| fp[3] ← b | Assign object pointer |
| p ← fp-1 | Advance the frontier and clear the flag |

The compiler statically adjusts the offsets within the allocation to account for the flag. The interesting thing about this sequence is that the last instruction advances the frontier and clears the abort flag atomically.

The interrupt system checks the low bit of fp. If it is set, the bit is cleared and we prepare to resume at the top of the transaction. If it is clear, we can resume at the interrupted instruction. The interrupt system still needs a way to find the top of the allocation. For this it can still use code scanning as mentioned above (now the top of the allocation is easy to recognize: $fp ← fp+1$), or it can search a table of allocation addresses.

The abort invariant

The above sequences required no work to abort them, but they were simple. Larger, more complicated allocations might not be so easy to undo. Fortunately, there is a rule that we can apply to all allocations to make sure that they are freely abortable: Registers which are live at the top of the transaction must remain constant throughout the transaction. Instructions which assign values to registers can only be performed if that register was dead at the top of the allocation.
Standing still

When using the abort technique, we have to be careful with long transaction sequences and speedy interrupts. If the interrupts occur fast enough, we may continue to abort the same transaction, and never make any progress! This case is rare because most allocations are small enough that we would never expect to receive interrupts quickly enough to cause problems. Extremely large allocations should be avoided, or split into smaller transactions.

2.4.3 Separate heaps

We have covered the forwards and backwards approaches to interrupting allocation, so all that remains is sideways. Side-step is a general term referring to avoiding the problem of transaction interruption by removing the resource dependencies between heap clients. The idea is to design the memory system so that allocations made by one thread do not affect the other threads.

In order to do this, we must split the resource among the threads. This means disjoint heaps, but it also means separating the pointers associated with the heap \((fp, lp)\). Each resource set can run independently of the others. An interrupt then saves and restores (or swaps out) all of these resources. This may sound a lot like true parallelism, and in fact it is a good way to get the most out of multi-processor machines, but the important point here is that the technique allows interrupts in the middle of an allocation.

Usually separate heaps means one heap per thread. If the programming style involves many short-lived threads, then the overhead of creating and destroying these heaps can slow things down significantly.

Once we separate the threads’ allocation spaces, it inevitably comes up that they want to interact. How do we handle pointers which are shared between threads? If thread A needs external data (from another thread), should the data move into A’s heap? Whether we copy the data into A’s heap or not, we need to record the interaction in a separate structure for the garbage collector. Otherwise the collector might discard the shared object prematurely.

We now need new transactions to record links between structures in different heaps. These recorded links will be considered by the collector when it is collecting a heap. But now an even deeper problem with thread interaction shows up. If a thread orders a collection on its heap, the objects in that heap will move around. During the collection a particular object might not even have a well-defined location. How can another thread which has a pointer to that object, and is running in parallel with the collection, refer to the object?
Another problem is that the links between objects in different heaps indicate to the collector that an object should not be thrown out. Cycles in those links can cause collections to leave dead data lying around. To prevent this type of memory leak an all-heaps garbage collection must be run once in a while.

Collecting a heap which has shared pointers requires synchronization between the threads involved. Likewise, performing an all-heaps collection requires synchronization between all threads. This means that interrupts can usually be serviced immediately; but occasionally a collection comes along that requires synchronization, and so each thread which is in the middle of an allocation must be dealt with to ensure that the collection does not disturb the atomicity of the transactions.

It may seem that we have lost the thread independence and interruptability we were looking for. Rather, we have found a means to avoid most cases where interrupts cause problems, but not all. The global synchronization issue must be solved with either run-to-completion or abort strategies. Although this method complicates the basic model, and has some disadvantages, it shows promise on multiprocessor machines, and points toward the following hybrid idea.

**Allocation arenas**

Using separate heaps requires that all pointers from one heap into another are recorded somewhere for the garbage collector to consider as roots for the destination heap. One clever idea allows more of the simplicity of the single heap model by hybridizing the shared heap and separate heap ideas.

In this technique, each thread has a private heap in which it performs allocations. Another heap serves as a public global heap. Although pointers to global heap objects may exist anywhere, there are no pointers into a private heap. That way, the private heaps can be collected in isolation. When a thread needs to share an object, the object must first be promoted to the global heap.

Collecting the global heap still requires synchronization with all threads, using either run-to-completion or abort as described above, or static arena style as described in the next chapter. However, the arenas can be small and thus cheaper to create and destroy, and there is no need for shared pointer lists.
Chapter 3
Static Arenas

The techniques mentioned in the last chapter approach the problem of maintaining atomicity in the presence of interrupts by completing or aborting any transactions in progress at interrupt time. We make sure that when the interrupt is processed there are no partially completed transactions, and thus we can guarantee atomicity.

Both of those techniques require work at interrupt time. In run-to-completion, the interrupt system checks for dangerous code, and possibly sets up the delayed interrupt, while the program itself must indicate dangerous/safe code boundaries and periodically check for delayed interrupts. Furthermore, interrupt latency grows depending upon the size of the dangerous code segments, as does the susceptibility to page faults. In the abort model, the interrupt system performs extra work at resume time to back up the PC to the beginning of the inlined transaction. Also, depending upon the sizes of the transactions, re-executing the aborted instructions may introduce significant interrupt overhead.

In this chapter I will describe a new technique which requires the least amount of work from the program and the interrupt system, placing most of the burden on the garbage collector.

3.1 Atomic fission

An allocation transaction can be separated into two parts: An allocation sequence and a heap initialization sequence. We start optimizing our transaction implementation by observing that only the allocation sequence needs to be atomic. That is the sequence which secures a section of the heap for the object. As long as the section is safe, the initialization can be performed later. Since the allocation sequence is small, it can be specifically coded to minimize run-time and interrupt-time work. In order to make the initialization work independent from the other threads, we need to remove the reliance upon the frontier pointer. We do this by assigning a different register to the allocated space:

\[
\begin{array}{ll}
1 & p \leftarrow fp \\
2 & fp \leftarrow fp+8 \\
3 & p[0] \leftarrow a \\
4 & p[4] \leftarrow b \\
5 & q \leftarrow p \\
\end{array}
\]

Allocate
Initialize
Assign object pointer

If the thread is interrupted outside of the atomic allocation sequence (instructions 1–2) then the garbage collector must be prepared to handle a partially-initialized object. For example, an interrupt at instruction 4 leaves the second field of the object uninitialized. The garbage collector must scan all of the initialized fields, and avoid the non-initialized ones.
3.2 Static-arena style

The initialization part of the allocation need not be completely contiguous, and the two parts of the allocation can even be separated by other code, including computation or another allocation. This freedom allows many kinds of optimizations. One such optimization is the ability to combine the allocation sequences from many allocation transactions, so that they allocate a space from the heap not for one but for many objects, which are then initialized separately.

I call this technique static-arena allocation. The first step in the allocation ensures a small dedicated space to the thread: A space that cannot be damaged or even accessed by other threads until the object pointers are shared. The size of this arena can be chosen at compile time to contain all of the objects allocated in the extended basic block.

The section of the heap which is allocated is called the static arena. The pointer (which is created during allocation) to the beginning of that section is the static-arena pointer, or sap.

The following example shows how a simple two-allocation block might look in static-arena style:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>sap ← fp</td>
</tr>
<tr>
<td>2</td>
<td>fp ← fp+16</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>sap[0] ← a</td>
</tr>
<tr>
<td>4</td>
<td>sap[4] ← b</td>
</tr>
<tr>
<td>5</td>
<td>p ← sap</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td>sap[8] ← c</td>
</tr>
<tr>
<td>7</td>
<td>sap[12] ← p</td>
</tr>
<tr>
<td>8</td>
<td>q ← sap+8</td>
</tr>
</tbody>
</table>

Allocate static arena
Initialize heap object
Assign object pointer
Initialize heap object
Assign object pointer

Assuming that the garbage collector can take care of partially-initialized objects (which we will come to in section 3.4), then instruction 2 is the only dangerous instruction. If an interrupt occurs immediately before that instruction is executed, then the space immediately after the fp might be used by the interrupting code, and so the program might not get a secure arena.

Since only one type of instruction can be dangerous, it is easy to modify the interrupt system to fix this case with minimal work. Specifically, between instructions 1 and 2 is the only point where sap = fp. The interrupt system only needs to check that equality to determine if the dangerous instruction was interrupted. If so, it fixes the problem at resume time, setting sap to the new value of fp (after other threads have allocated) by executing sap ← fp again. This is a type of abort technique, except that since only one instruction needs to be redone, the PC doesn’t need to be rolled back.

If interrupts occurs after instruction 2 there are no problems, because this block has already allocated its own section of the heap (which fp now points beyond).

The trickiest part of designing a system for static-arena style is performing garbage collection in the presence of partially-initialized objects. The static arenas must be scanned during the collection, and kept mostly intact so that the initialization can continue. The garbage collector must know, when tracing a suspended thread:

- If an arena exists
- Where it is
- How large it is
- Which words within it have been initialized

3.3 The anatomy of a static arena

Each static arena contains all of the space needed by a particular extended basic block. The arena has a fixed size which is the sum of the sizes of the allocations, as determined by the compiler. That
space is initialized sequentially and when the initialization is complete, the arena becomes a regular part of the heap.

A static arena therefore has a frontier much like the heap does, but it is managed statically by the compiler, and thus not kept in a register. A static arena may contain multiple objects, some of which are fully-initialized, some partially-initialized, and some completely uninitialized.

At garbage collection time, fully-initialized objects are treated exactly like normal heap objects. They are traced like normal objects, and since the interrupted program has finished initializing them, they are even copied outside of the arena. The remainder of the arena must be maintained as-is so that the program can continue initializing it.

### 3.4 Scanning interrupted threads

When the garbage collector scans an interrupted thread in static-arena style, and the interrupt in question occurred during the initialization of an object in an arena, then there is somewhat more work to be done than usual. The arena must be recreated in to-space, the sap must be set to point to the new arena, and all of the initialized words in partially-initialized objects must be scanned.

The new to-space arena should not contain the fully-initialized objects, or more specifically, the objects for which a pointer has already been assigned. Those objects will be scanned and copied independently by the collector. In the earlier example code (page 18) the \( p \leftarrow \text{sap} \) and \( q \leftarrow \text{sap+8} \) instructions each complete the creation of an object by building a pointer to that object in some register. If that pointer is live then it will be traced by the garbage collector. We ignore those completed objects in the scanning of the static arena, and since those objects are copied separately, the new to-space arena will not contain them. If the new arena is copied again (in another garbage collection), then those objects are again ignored—which is important because this time they don't
The heap

Overlap from previous objects

Partially-initialized object

Uninitialized object

Arena pointer

Start of partially-initialized object

Arena frontier

End of arena

Figure 3-2: A static arena which has been copied by the collector
even exist in the arena.

Because fully initialized objects are no longer part of the arena, the newly created to-space arena does not even need to contain space for them. The new arena can be created smaller than the first one, and the new sap offset by that amount so that the same offsets are used in accessing words of the new arena. In figure 3-2, we see that the sap points into some preceding object on the heap, not into the actual arena. As a result the offset from the sap to the words in the arena is the same as in the previous figure (figure 3-1. This small optimization does not necessarily save a significant amount of heap space, but it illustrates the idea that the garbage collector can do just about anything to the arena as long as it appears consistent to the interrupted program.

### 3.5 Conclusions for static-arena style code

The purpose of compiling in static-arena style is to reduce the run-time and interrupt-time costs normally associated with handling interrupts in a garbage-collecting system. We can do this by placing the burden of controlling the heap on the garbage collector, and organizing the allocation sequences (without increasing the number of instructions necessary) such that they do not interfere with one another.

Additionally, static-arena style code does not suffer from the primary disadvantages of abort or run-to-completion code. Interrupts are handled promptly. During page faults, the processor does not need to be completely halted (as in run-to-completion style) to load the new page. There is no re-execution of code (as in abort style), which means the compiler is less restricted.
Chapter 4

Mutations

There is actually another operation which can be performed on the heap besides allocate and collect. I have been ignoring the mutate operation for the sake of simplicity, and because the complexities of mutation are easily described using the foundation I have given above.

A mutation is the modification of the contents of a heap object which was previously allocated. It usually involves the replacement of one word value in memory with another. Some types of applications require no mutations at all (resulting in a purely functional program), while others might require many mutations.

If mutations are cheap, then they might provide a useful way to avoid extra memory allocations. They can also be convenient for the programmer in some programs where a purely functional algorithm would be more complex. On the other hand, mutation heap transactions sometimes need to be handled more carefully than allocation transactions, because of their interactions with the garbage collector.

4.1 The simplest kind of mutation

At its heart, a mutation is a very simple operation. Compared to garbage collection it operates on a very small number of values, and unlike allocation it has a fixed size (here I assume that all mutations are one-word mutations). In fact, the mutation transaction itself can often be performed in a single instruction.

\[
a[8] \leftarrow b\quad \text{Replace the third word of object } a \text{ with the value } b
\]

This one-instruction transaction is already atomic. No extra work is required to inform the interrupt system or the garbage collector, since they will already be taking care of register movement and the transaction itself cannot be interrupted.

4.2 Mutation in generational heaps

A generational garbage collection system makes collection cheaper by restricting the collection to the most recently allocated objects. In a generational garbage collection system, the object which we are modifying (a in the above example) might be in any heap, including the older generations. A generational system saves time by scanning only the youngest generation if it can. If an older-heap object is modified to contain a pointer to a younger-heap object, then there is the possibility that the younger object might be prematurely collected. To avoid this possibility, the collector must know about every pointer from an older to a younger heap. For the purposes of scanning the younger heap, this pointer is then considered a root.

The collector tracks every pointer from an older to a younger heap. The program is required to tell the garbage collector every time such a pointer is created. This is only possible when
a mutation occurs, and so each mutation updates the list of tracked pointers. If the younger object is eventually promoted to a generation which is at least as old as all of its tracked pointers, then those pointers can be forgotten by the collector.

In a mutation of an object in the youngest heap, there is no need to tell the garbage collector about the mutation. Unfortunately, at compile time the only objects which can be guaranteed to be in the youngest heap are the ones allocated after the last heap limit check.

All other mutations need to inform the garbage collector which object was modified. This means additional work for the mutation transaction. There are two basic techniques for sending this information to the garbage collector.

### 4.2.1 Card marking

In card marking, a map of the entire space in every heap (except the youngest) is created in some dedicated part of memory. Each slot on that map indicates to the garbage collector whether or not that particular part (called a card) of the heap has been modified. Modified cards are called *dirty*, and the unmodified ones are called *clean*. To be nominally effective, each mutated word must be contained in a card which is marked dirty. In particular, the map doesn’t have to be one-to-one. A single bit in the map can correspond to any number of words on the heap. At garbage collection time, the collector then considers every pointer in every dirty card to be a root.

The optimal card size is hard to determine. If the cards are too small the card map can become too large and require a significant time for the garbage collector to examine. However, since the garbage collector must scan every word in a dirty card, if the cards are too large this scanning becomes significant.

| a[0] ← b | Perform the mutation |
| t ← a / 256 | Compute the card number |
| t ← t − heapaddr | Mark the card |

In the above example, *heapaddr* is the address of the beginning of the heap. The card map entries are one-word long, and each corresponds to 256 bytes of heap memory. The example sequence is not interruptable.

### 4.2.2 Store lists

Another approach to communicating mutations to the garbage collector uses *store lists*, also called *remembered sets*. Here, a list is maintained of the addresses of every mutated word. Each mutation adds its address to the list, and the garbage collector only scans the words on the list.

**Store vectors**

The store list can be implemented as a vector in memory with a frontier and a limit much like the heap, in which case the program must perform additional checks to ensure that there is space on the store list for the mutated addresses. In that case the list is called a store vector. If the store vector becomes full, a garbage collection is needed.

The store-vector frontier pointer, or *sfp*, contains the address of the uninitialized portion of the store vector. The store-vector limit pointer, or *slp*, contains the address of the end of the vector. These pointers correspond in function to the heap’s frontier and limit pointers.

| compare sfp+4, slp | Check for space on the store vector |
| if > then goto gc | Collect if not enough |
| ... | |
| a[0] ← b | Perform the mutation |
| sfp[0] ← a | Store the vector entry |
| sfp ← sfp+4 | Advance the store frontier |
The above sequence shows a store-vector limit check and a mutation transaction. As in the card marking example, the transaction presented is not atomic. Ways to make it so will be discussed in section 4.3.4.

**Store lists on the heap**

Another way of implementing a store list is on the heap itself, as a linked list. In this implementation no new store-list frontier pointer or store-list limit checks are necessary, but the mutation transaction is a bit more complex. In fact, in this kind of store list adding a new entry involves performing an allocation.

In this case a store-list pointer (or stp) is needed to refer to the head of the store list.

<table>
<thead>
<tr>
<th>a[0] ← b</th>
<th>Perform the mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>fp[0] ← a</td>
<td>Initialize the store list entry</td>
</tr>
<tr>
<td>fp[4] ← stp</td>
<td>Update the stp</td>
</tr>
<tr>
<td>stp ← fp</td>
<td></td>
</tr>
<tr>
<td>fp ← fp+4</td>
<td>Advance the fp</td>
</tr>
</tbody>
</table>

In the above example new entries are placed on the head of the store list. As in the previous two examples, this one is not atomic.

### 4.3 The transactional perspective

No matter which technique is used, the basic idea is that the program communicates the set of mutated addresses via some data structure to the garbage collector. Each mutation transaction must perform work to add data to that structure. That work is in addition to the actual mutation, which means that we no longer have the simple one-instruction mutation discussed in section 4.1 but now the mutation transaction is several instructions, which must still be atomic with respect to the heap.

The fundamental difference between a mutation transaction and an allocation transaction is that the mutating instruction irrevocably modifies the heap. The run-to-completion and abort techniques discussed for allocation work to keep mutations atomic, because run-to-completion works for any kind of dangerous code and abort works because of specific characteristics of the mutation sequence. As we will see, however, applying the static-arena ideas is not as straightforward. The mutating instruction must be treated with care.

#### 4.3.1 Run to completion

Run-to-completion works without special consideration on mutation transactions. Delaying the interrupt service during a transaction means that a garbage collection will never occur when a partial mutation is in place.

<table>
<thead>
<tr>
<th>dflag ← 1</th>
<th>Mark this code as dangerous</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0] ← b</td>
<td>Perform the mutation</td>
</tr>
<tr>
<td>sfp[0] ← a</td>
<td>Store the vector entry</td>
</tr>
<tr>
<td>sfp ← sfp+4</td>
<td>Advance the store frontier</td>
</tr>
<tr>
<td>dflag ← 0</td>
<td>Entering safe-code</td>
</tr>
<tr>
<td>compare iflag, 0</td>
<td>Check for deferred interrupt</td>
</tr>
<tr>
<td>if ≠ then trap</td>
<td>If so, activate the interrupt system</td>
</tr>
</tbody>
</table>

The above example shows a generic run-to-completion system, where the next safe code segment performs a check. The dfflag provides a mechanism for the interrupt system to detect dangerous code (there are many ways to do this, as described in section 5.1.1). As described in section 2.4.1, the iflag is set by the interrupt system when interrupts are deferred.
The safe-points method—where interrupts are always deferred until heap limit checks—also provides atomicity for mutation transactions. We can also use the dangerous-points method, close-coding the mutation transaction so that user code can always allow interrupts.

In all cases, a run-to-completion mutation looks very similar to a run-to-completion allocation. The technique applies in a general sense to both transaction types.

### 4.3.2 Aborting the transaction

The abort technique does not work for any given code sequence. On the contrary, when dealing with allocations (in section 2.4.2) we saw that the instructions had to be specifically tailored to allow an abort. In order to see if the abort technique works on mutations, we must imagine the various abort scenarios.

#### Aborting with store lists on the heap

Consider the following mutation sequence implemented with store lists on the heap, in which the store list entry is created before the mutation:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fp[0] ← a</td>
</tr>
<tr>
<td>2</td>
<td>fp[4] ← stp</td>
</tr>
<tr>
<td>3</td>
<td>stp ← fp</td>
</tr>
<tr>
<td>4</td>
<td>fp ← fp+4</td>
</tr>
<tr>
<td>5</td>
<td>a[0] ← b</td>
</tr>
</tbody>
</table>

The abort technique presents first the problem of re-assigning the stp. As I stated in the discussion of the abort technique (see section 2.4.2), any live pointers at the beginning of an abortable transaction must remain constant and live for the duration of the transaction. Instruction 3 violates that rule. Even though we do not normally consider the fp a 'live' or 'dead' register, instruction 4 similarly violates the rule.

Strictly speaking, instruction 4 can usually be re-executed without undoing its effects. The frontier is advanced unnecessarily, and some heap space is perhaps wasted, but these problems are not fatal.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a[0] ← b</td>
</tr>
<tr>
<td>2</td>
<td>fp[0] ← a</td>
</tr>
<tr>
<td>3</td>
<td>fp[4] ← stp</td>
</tr>
<tr>
<td>4</td>
<td>stp ← fp</td>
</tr>
<tr>
<td>5</td>
<td>fp ← fp+4</td>
</tr>
</tbody>
</table>

As in the above example, creating the store list entry after performing the mutation fixes the fp ← fp+4 instruction, but the stp ← fp instruction must still be undone.

The problem with this transaction is that unlike the allocation transaction, we can’t just walk away from it. At least one instruction needs to be specifically undone. The problem can be fixed by designing the interrupt system to undo that instruction if necessary. The interrupt system can perform a special-case check for the times when stp = fp and specifically undo that instruction, by performing a stp ← fp[4].

Again, it is important that the registers which are live at the top of the transaction remain live throughout it. This includes b in the above examples. For example, if b were considered dead by the collector after the mutation, the object at b might be collected even though it is referenced by the object at a. That is true until the store list entry is properly created, at which point b may become dead. This holds for the following two cases as well.

#### Aborting with store vectors

Aborting a store-vector mutation is similar to aborting a store-list mutation, but simpler. Both transactions perform a ‘frontier advance’ (in the store-list example above this was fp ← fp+4),
which cannot be easily aborted. The store-vector transaction, however, does not need to modify live
registers.

```
| a[0] ← b | Perform the mutation |
| sfp[0] ← a | Store the vector entry |
| sfp ← sfp+4 | Advance the store frontier |
```

This sequence is abortable.

**Aborting with card marking**

Card marking is the easiest of all to abort.

```
| a[0] ← b | Perform the mutation |
| t ← a / 256 | Compute the card number |
| t ← t - heapaddr | Mark the card |
| t[0] ← 1 |
```

The mutation can be performed before or after the card marking. Regardless, none of the
instructions require undoing.

### 4.3.3 The need for a fine-grained solution

A fine-grained interrupt system allows interrupts at every instruction. Some language systems
require fine-grained interrupts with low run- and interrupt-time overheads. Run-to-completion in-
troduces interrupt delays and suffers during page faults, and the abort technique adds work to the
interrupt system and requires re-running code. If we wish to allow low-cost interrupts during the
mutation transaction, then the means by which we communicate with the garbage collector must be
modified.

Plain mutation registration (card marking or store lists) will not suffice as before. In the abort
technique we could interrupt after them, but we were assured that the registration would be re-
executed. If we do not wish to re-execute code, then a garbage collection might occur between the
registration and the mutation, no matter how they are ordered. There is a chance that the collector
will miss the mark\(^1\).

If mutation is done before registration, then an interrupt and collection after the mutation will
have this effect. If registration is done first, then a collection might reset the mark or purge the store
list entry before the mutation. This is possible because the location which was marked as mutated
might not appear to the compiler to be a pointer into a younger-heap object, and therefore is not
worth considering as a root. The collector cannot simply leave all dirty marks in place or else there
will eventually be too many.

We need a way to ensure that the collector does not purge card map or store list entries before
the mutation is completed. More specifically, the collector needs to set up the stack frame so that
the dirty mark or store list entry is restored when the thread is resumed. Doing this on a per-stack-
frame basis means that the compiler does not need to scan all stack frames first to determine which
entries can be removed—it can instead handle them as part of the collection thread.

### 4.3.4 Atomic fission: static store subvectors

Atomic fission, as I discussed in section 3.1, allows fine-grained interrupts during allocation trans-
actions by splitting the transaction into two parts: An atomic sequence which is small enough to
allow cheap per-instruction interrupts, and a separate non-atomic interruptable sequence. Store-list
updating is somewhat similar to allocation, and so one might expect that the techniques also ap-
ply. Once the transaction is separated into atomic and non-atomic parts the atomic parts can be

---

\(^1\) or the store list entry

26
coalesced just as in static-arena style. Here I will discuss specifically what is necessary to make the static-arena idea work on store vectors. The result is called static store subvectors.

The static-store-subvectors technique uses a store vector, from which subvectors are allocated for each block just as arenas are allocated in static-arena style. The pointer to the allocated subvector for the block is the static-store-subvector pointer (or the sssp).

The mutation transaction cannot be so easily separated as the allocation was into atomic and non-atomic parts. Particularly, the initialization of the store vector entry must occur before the mutation, but initialization and mutation must be atomic with respect to the garbage collector, so that it does not prematurely purge the entry. The first step is to make the non-allocating part of the mutation transaction non-atomic.

**A three-step mutation**

It is necessary to introduce a third step to the mutation, which signals to the garbage collector that the mutation is over and the store vector entry can be purged if possible. Thus when the third step of the mutation is not seen, then the collector assumes the mutation is not complete, and that the store vector entry must be restored.

For example, a mutation could require two identical entries on the store vector. Adding the second entry on the store vector would constitute the third step of the mutation. Having two entries on the store vector for some address indicates to the garbage collector that the mutation is complete and it may purge the store vector entries. The two separate segments of code which create the entries can then be split into atomic and non-atomic parts, and the whole sequence can then be converted to static-store-subvector style.

| sssp ← sfp | Set the sssp |
| sfp ← sfp+8 | Advance the store frontier |
| ... | ... |
| sssp[0] ← a | Store the first vector entry |
| a[0] ← b | Perform the mutation |
| sssp[4] ← a | Store the second vector entry |

If a mutation is incomplete but its associated stack frame is no longer live (the thread will never be resumed), then the entry may be purged. In partial mutations with live stack frames, as mentioned above, the store vector entry must be removed from the vector and stored with the frame at collection time, so that when it is resumed the entry can be returned to the vector.

Somehow the garbage collector must be able to identify, when it scans a suspended thread, which partial mutations belong to it. Static store subvectors provide this functionality—the entries within the thread’s subvector are the ones which need saving.

**Dangerous code in mutations**

In a system where mutations are used often we might want to make sure that the mutation sequence is as small as possible. In that case, rather than adding instructions to perform the third mutation step, we can use the dangerous code marking techniques from run-to-completion to make the transaction small yet maintain non-atomicity. This puts some more work on the garbage collector.

All that is needed is a way for the garbage collector to recognize when the vector entry has been initialized but the mutation has not occurred. In a sense the sequence between those events is dangerous code, and can be marked just as transactions were marked in run-to-completion. This is not the same as using run-to-completion to handle interrupts, because the interrupt is not delayed when dangerous code is reached.

When the garbage collector detects this kind of dangerous code, it knows to set up the store vector entry so that it can be restored. In the three-step mutation the uncompleted entry was the one which was not duplicated on the store vector. It could also have been detected by looking at the last entry in the subvector. In the dangerous code case, we use the latter method. It requires
that the garbage collector know the location of the frontier in the subvector—a value which is not stored in a register.

**More on static store subvectors**

Using one of the above techniques, the non-allocating part of a mutation can be made non-atomic. Now it is possible to treat only the allocation of the store vector entries atomically, and allow interrupts everywhere else. Furthermore, since the allocation sequence is quite small, interrupts within it can be tolerated using the same simple methods we use for static-arena allocations (see section 5.2).
Chapter 5

Implementing Transactions

The previous chapters introduced the general concepts and algorithms necessary to allow interrupts in a language system. In this chapter, I will discuss many of the details which I abstracted or glossed over before, and discuss how these implementation decisions affect the greater design.

5.1 General issues

5.1.1 Detecting transactions and dangerous code

The abort technique requires that the interrupt system be able to detect if it has interrupted a heap transaction. Likewise, the run-to-completion technique (except for safe points) requires the interrupt system to detect dangerous code. There are many ways to implement this.

Disabling interrupts

In run-to-completion, the interrupt system needs to detect the dangerous code only so that it can return immediately to that code, without servicing the interrupt at all. Many machines have instructions which do just that: temporarily disable interrupts. In some systems it might be advantageous to use those instructions.

Unfortunately, they require that the program be in kernel or supervisor mode, since the processor does not trust user-level programs to disable interrupts. This means that using such instructions is usually impossible. However, in an ideal programming system where the run-time system is the operating system, and compiled code is trusted, it might be completely reasonable to use kernel mode instructions like these.

Disabling interrupts is dangerous. As previously mentioned, deferring interrupts over page faults can cause significant delays. Disabling interrupts over page faults is fatal. If we wish to disable interrupts over a piece of code we must make sure that the entire code sequence and all of the necessary data are already loaded into memory. Interrupt disabling essentially requires close-coded transactions, and even then they must be carefully designed.

Flags and fp-locking

Perhaps the simplest technique of all is to have the program set a flag at the beginning of a dangerous sequence and clear that flag at the end. Then the interrupt system can easily detect the dangerous sequences by checking the flag.

As mentioned above, introducing flag-setting and clearing instructions to the code stream can be costly, depending upon how short and how frequent the dangerous code is. If the flag is located in memory (rather than in a register) then the costs can be unbearable. Of course, dedicating a register for such a flag could also slow down the program, because fewer available registers mean more data spills.
In section 2.4.2 I introduced the fp-locking method for marking transactions with the low bit of the frontier pointer. The interrupt system can easily check the low bit and determine if dangerous code was interrupted or not. Here is an example of fp-locking around an entire allocation in run-to-completion or abort style:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fp ← fp+1</code></td>
<td>Lock the low bit of the <code>fp</code></td>
</tr>
<tr>
<td><code>fp[-1] ← a</code></td>
<td>Initialize heap object</td>
</tr>
<tr>
<td><code>fp[3] ← b</code></td>
<td></td>
</tr>
<tr>
<td><code>p ← fp-1</code></td>
<td>Assign object pointer</td>
</tr>
<tr>
<td><code>fp ← fp+7</code></td>
<td>Atomically clear the low bit and advance <code>fp</code></td>
</tr>
</tbody>
</table>

Note how the compiler accounts for the setting of the lock bit by adjusting the offsets.

In the abort technique, the bit must be cleared before resuming the thread. The most interesting part of this technique can be found when dangerous code is restricted to allocation transactions as in the above examples. If the last instruction of the allocation transaction advances the `fp` past the allocated object, then the clearing of the lock bit can be done simultaneously. With this optimization, the run-time cost of the flag is one register instruction per allocation.

When using generic flags for detection in the abort technique, the committing of the transaction (the pushing forward of the frontier pointer) might be performed before the flag is cleared, allowing an abort to occur in between. This could leave useless dead heap space lying around. Fp-locking avoids this by clearing the flag and committing the transaction in one instruction.

**PC tables**

When implementing the detection of dangerous code by the interrupt system, it is possible to avoid any run-time costs whatsoever. However this generally comes with substantial interrupt-time overhead. On a system with infrequent interrupts, such methods might be desirable. Implementing PC tables is one example, and code scanning is another.

If the compiler can communicate to the interrupt system the set of all PCs of instructions which are within a dangerous code sequence, then the interrupt system can simply check whether the interrupted PC is a member of that set. For example, the compiler could augment the code stream with a sorted table of pairs of PCs (one beginning a dangerous sequence, the other ending it), on which the interrupt system could perform a binary search.

**Code scanning**

If the dangerous (and safe) code sequences are carefully designed by the compiler to be recognizable, then the interrupt system could attempt to scan the instruction stream to determine which type of code was interrupted. This technique depends mostly on the representation of machine instructions, and what indications in the instructions are used to represent safe and dangerous code.

On the IA32 architecture instruction representations are variable-length, which makes forward scanning very difficult, and backward scanning nearly impossible.

**Close-coding: looking at the PC**

When using the dangerous-points technique (close-coding transactions), the only dangerous code is within the well-known transaction procedures. Checking if we have interrupted one of those functions can be as simple as looking at the interrupted PC [7].

The same idea holds for garbage collection. Since we never want to interrupt the collector (we always want run-to-completion-style collection transactions), and collection is close-coded, if we interrupt collector code (as determined by looking at the PC) we defer the interrupt until the collection is complete. On a similar vein, since the collection transaction is so large, we could also replace the interrupt system temporarily with one which only deferred interrupts.
Static store subvector mutations: some of the above

For detecting the dangerous code in static store subvector mutations, interrupt disabling is overkill since the goal of static store subvectors is allowing interrupts within the mutation transaction anyway. PC tables and code scanning can be used as prescribed above, except that it is the garbage collector which does the work of searching or scanning, and therefore the costs are not as significant. Flags are possible, but the intended recipient of the flag is now the garbage collector, and not the interrupt system.

5.1.2 Registers

If the garbage collector is called during an interrupt and the interrupted thread is waiting to be resumed then the collector will eventually scan the stack frame. This frame will contain, among other things, the values of the registers at the time of the interrupt. If any of those values are live pointers to objects then they must be scanned. But how does the collector determine which ones are live pointers and which are not?

Register masks

Since the compiler knows at compile time which registers are live, it can relatively easily add this information to the code stream. The garbage collector simply looks up the register information for the particular interrupted PC, and then scans whichever registers are indicated. Although there are ways to trim down these register-mask tables, their size can be troublesome since there would be one such mask per interruptable instruction.

These tables are very similar in purpose to the tables used by Diwan et. al. to support garbage collection in Modula-3 [2]. Those tables indicated not only register liveness but also the existence of derived pointers. They were able to compress those tables to 16% of the code size.

Register partitioning

We can avoid this extra data by partitioning the registers into two sets: One for live pointers, and one for everything else. Then the garbage collector always knows which registers are live pointers, because it assumes that the compiler has generated code specifically to avoid breaking that invariant. Although easy on the garbage collector, register partitioning restricts the compiler’s output, and can therefore produce slower code.

How many registers should be in each set? The code may deal with many object pointers and few integers. This essentially wastes the entire non-live-pointer register set, and requires more live pointer spills than would be needed in an unpartitioning compiler. Another program might use mainly integers, producing the opposite effect. Some programs might contain instances of both extremes.

Dynamic partitioning

With proper analysis, the compiler might be able to determine at compile time the optimal sizes of the two register sets for a given program, or even on a per-basic-block basis. This dynamic partitioning provides a sort of compromise, because while the compiler is less restricted in its output, the garbage collector still needs to know the particular partitioning at each instruction. Dynamic partitioning also produces significantly smaller register-mask tables.

5.1.3 Spills

Stack swapping

During an interrupt, the spill area must be saved and restored just like the registers. This can be a fairly large interrupt-time cost, but it is easy to avoid. Since the spill area is usually on the stack or in some other space pointed to by a register, replacing the spill area with a new one can be as
simple as replacing the spill area pointer. When that pointer is the stack pointer, this is (literally) called stack swapping.

While it saves a lot of time for each interrupt, the run-time system and the compiled code must be carefully designed to make stack swapping easy. If the stack pointer moves at all during the course of running or interrupting a program, then it must be saved and restored in such a way that the program sees a consistent pointer upon resuming the interrupt. Since the garbage collector needs to examine the spill area, the location of that area relative to the stack pointer must be clear at all times.

**Live spilled values**

Live spilled values must also be known by the garbage collector when it traces suspended thread records. Normally, when the compiler generates the garbage-collector call, it generates code to indicate live spilled values to the compiler by placing them all in a special record or perhaps building a bitmask of the spill area. For interrupts at every (or nearly every) instruction, we cannot take such special care.

Solutions to this problem are analogous to solutions for indicating register liveness. The spill area can be mapped out with liveness tables in the code stream, but this is especially costly so the best solution is almost always partitioning. Partitioning the spill area is usually cheap because the size of the spill area is not completely restricted by the architecture.

### 5.1.4 Heap availability

Since an interrupt might cause an allocation or a collection, whenever we resume from an interrupt we must make sure that the amount of free heap space remaining is sufficient to allow that thread to continue at least to the next heap limit check.

This can be done somewhat easily in the safe-points method. There, the interrupts are postponed until a heap limit check, which then calls the run-time system to perform a garbage collection (of course, the collection doesn’t actually occur, instead the interrupt is handled). It is easy and somewhat common to set up the return code from the garbage collector call to jump back to the heap limit check. That way the limit check is performed again in case the call actually handled an interrupt, and re-executing that code is fairly cheap relative to garbage collection.

Furthermore, language systems which use guard pages to detect heap overflow need not perform the extra limit check at all, and no matter what interrupt-handling scheme is used the guard page will always properly indicate heap overflow.

The run-time system in static-arena style does not need to know how much heap is needed by the program, but instead needs to know how large the allocated static arena was. When it copies the arena it must leave enough room between the arena and the succeeding object.

However, for methods other than static-arena style and safe points and on systems which do not use guard pages, some way of ensuring sufficient heap availability is needed. One way is to use code annotations to mark certain regions of code as requiring a certain amount of heap to run. Then the interrupt system performs a limit check just before returning to the code. We could also store the heap availability requirement in a word on the stack or (if registers are plentiful) in a register. Then the interrupt system can simply check that value.

### 5.2 Atomicity for static-arena allocations

The static-arena style, as described above, separates the allocation and initialization sections of the allocation transaction, and then requires only the allocation to be atomic. Since the allocation sequence is small, achieving this atomicity is quite easy. Here is the basic allocation sequence in static-arena style (remember sap is the static-arena pointer, and fp is the frontier pointer).

| sap ← fp | Set the sap |
| fp ← fp+8 | Push the frontier forward |
In the above sequence, the second instruction is the only dangerous one. Since there is only one such dangerous instruction, with some care interrupts can be allowed there. If an interrupt occurs on the dangerous instruction, the interrupt system can re-execute the instruction $\text{sap} \leftarrow \text{fp}$ (setting the sap), which will correct any problems caused during the interrupt.

5.2.1 Complex solutions

A method is still needed for detecting when the allocation sequence has been interrupted. Many of the methods discussed in section 5.1.1 will work. First I will briefly provide some examples of applying those techniques to the static-arena allocation sequence, and then in the next section I will reveal the simplest and cheapest method.

Fp-locking, for instance, works well around the static-arena allocation because the last instruction in the sequence advances the frontier. Code scanning is another viable technique here, since this is the only point in the user code where we will find the $\text{sap} \leftarrow \text{fp}$ instruction. The interrupt system could check for that instruction just before the interrupted point, and if found re-execute it.

We could even use an atomic exchange-and-add instruction—if the compiler supports it—to provide an interruptable allocation sequence. The IA32 architecture has such an instruction, which can be used as follows:

<table>
<thead>
<tr>
<th>sap $\leftarrow 8$</th>
<th>Set the sap to the size needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>exchange-and-add fp, sap</td>
<td>Allocate the static arena</td>
</tr>
</tbody>
</table>

5.2.2 The simple solution

However, the simplest solution is the best. As I mentioned in section 3.2, the only time sap is equal to fp is between those two instructions. The interrupt system can easily compare the two values to detect the dangerous scenario. If they are equal at interrupt time, then the interrupt system must ensure that they are equal at resume time by performing $\text{sap} \leftarrow \text{fp}$. This method incurs the least run-time overhead, and is very cheap at interrupt time.

5.3 Deferring interrupts

In the run-to-completion scheme, interrupts which occur in dangerous code need to be deferred and handled later. Storing and reloading the interrupt information is generally easily implemented, but choosing a method for actually detecting and servicing postponed interrupts can be tricky.

As discussed above, disabling interrupts solves both the dangerous code detection and interrupt deferral problems, but it is not always applicable.

The safe-points technique (allowing interrupts only at heap limit checks) is a simple solution. The interrupt system stores the interrupt information, and sets the heap limit pointer to zero, or some other low value. The program is then resumed until it hits a limit check, and calls the garbage collector. The garbage collector checks the limit pointer to see if it has been tampered with, and if so the collector resets it, and then handles the interrupt (performing no actual collection, yet).

The dangerous-points technique (close-coding atomic transactions and postponing interrupts within them) is similarly simple. As discussed above, the dangerous code can be detected by the interrupt system by looking at the interrupted PC. In order to service the delayed interrupt, we can simply add a check at the end of the close-coded transaction function, just before it returns.

5.4 Aborting transactions

Aborting a transaction requires knowledge of the address of the first instruction in the transaction—the address to which we return when the thread is resumed. Finding that address is very similar to detecting that we've interrupted the transaction in the first place. In fact, the method used will usually depend on how the detection is done, and vice versa.
We can use code scanning to search through the program instructions to find the beginning of the transaction. This works well with code scanning for detection, since if we have detected that we are inside of a transaction we probably had to scan to the first instruction anyway.

Using PC tables to detect the transaction also provides an easy solution, since the PC tables in fact tell us where each transaction starts and ends.

5.5 Static arena style

When a static-arena style block is interrupted, the garbage collector needs to know the dimensions of the arena and the partially-initialized objects inside. For simplicity, I will assume that the blocks are designed to initialize words in the arena sequentially and to complete each object (assign a pointer to it) immediately after it is fully-initialized. This means there can be at most one partially-initialized object per arena. The collector must know:

- The size of the arena.
- The address of the partially-initialized object, if any.
- The location of the arena frontier (the first uninitialized word).

As usual, these can all be indicated via annotations. And again as usual, there are ways to avoid using annotations. Which method to choose depends on the specific costs with respect to the situation.

5.5.1 The arena size

The size of the arena is constant throughout the block. Even if a collection is run and the arena is scanned and copied to become smaller (see section 3.4), the offsets used by the program to access the arena—all relative to the sap—remain the same. It is the amount of heap originally allocated for the arena which I will call the size. Since the size changes rarely (it is in fact similar to the heap availability requirement discussed in section 5.1.4), it can be annotated efficiently. It can also be stored on a per-block basis in a thread-specific field.

Surprisingly, it can even be determined dynamically by the interrupt handler. The first time a given basic block is interrupted the size of the arena is fp – sap. Upon resuming the interrupt the sap may have changed, but it is known by the run-time resume code, so if that block is interrupted again, the handler can recognize it by comparing the sap to the previous one. The only way that the sap compare might trigger improperly is if the thread initiates a collection between the interrupts, but it will not do so until it has finished initializing the static arena. Although this method for finding the static-arena size for the collector is lightweight and does not take up extra space as an annotation would, it is a bit complicated and asks for some per-interrupt work, so it might not be the best method.

5.5.2 The partially-initialized object address

The partially-initialized object address changes frequently per block. In fact, in between object completion and the initialization of the first word of the next object there is no partially-initialized object. The garbage collector needs to know if such and object exists, and if so where (relative to the sap). Per-instruction annotations can easily convey this information.

In practice, knowing whether a partially-initialized object exists is not so important. If it does not exist we can annotate or otherwise communicate that the partially-initialized object starts at the arena frontier, and is therefore empty. For the garbage collector, this is the same thing.

Backwards instruction scanning can also be used to find it, if the architecture's instruction representation lends itself to such scanning. For this we must assume that there are no branches during the entire initialization of the arena. This restriction is harsh but necessary. Beginning just before the interrupted instruction, we scan back looking for an instruction which assigns a pointer as offset to the sap (e.g., p ← sap+8). Since the only such instructions will be those that complete
an object, we can assume the partially-initialized object starts immediately after it. We must also recognize the top of a block or perhaps some instruction from the allocation sequence in order to know if the partially-initialized object starts at the beginning of the arena.

5.5.3 The arena frontier

The arena frontier changes at almost every instruction in the block. Much like the partially-initialized object address, the frontier location can be easily represented with per-instruction annotations.

Code scanning for the last initialized arena word will also work. This time we would scan for an instruction which stores a value in an address offset from the sap (e.g., `sap[8] ← c`).
Chapter 6
Fine-grain Interrupts In SML/NJ

To test the soundness of providing fine-grain interrupts on a production compiler, and to provide examples and experience for this document, I implemented many of the ideas presented in this paper within version 110.20 of Bell Labs’ SML/NJ, a compiler and programming environment for Standard ML.

SML/NJ currently uses the safe-points technique for atomicity. For this paper, I modified the compiler to generate code suitable for fine-grain interrupts. Roland McGrath helped by modifying the run-time system to support the various techniques.

A total of three different versions were prepared. The first version was for the Sparc architecture, and used the abort technique with fp-locking. The second version implemented static-arena style on the IA32 architecture. The third implemented static-arena style on the Sparc.

6.1 Stock SML/NJ

6.1.1 Safe points and generational collecting

SML/NJ uses the safe-points technique. The safe points in SML/NJ are the garbage-collector calls, and interrupts are deferred by setting the limit pointer to zero to trigger the next limit check. The heap is generational, and for mutations SML/NJ uses a store list implemented as a linked list on the heap.

6.1.2 MLRISC

MLRISC is a compiler backend which accepts an intermediate language and outputs machine code for a particular machine [3]. It it used by SML/NJ and most of the compiler work needed to achieve fine-grain interrupt handling was done on MLRISC.

MLRISC instructions generally correspond to machine instructions, except in cases where the instructions contain subexpressions which are too complex for the target instruction set. The stage of the SML/NJ compiler which generates the MLRISC code is where many of the interrupt-handling decisions are made. Code annotations are generated at this level, and the transaction style is determined by this level as well as the extended block layout.

In order to avoid problems with possible optimizations which might move instructions into or out of the static-arena allocation sequence, the allocation sequences were treated as MLRISC instructions. I modified the MLRISC instruction tree definition and the target compiler definitions so that the allocation sequence would be treated atomically until the final code generation stage. This also avoided introducing spills into the static store subvector allocation, which was possible because the subvector pointer (sssp) was not a dedicated register.

The version of MLRISC in SML/NJ 110.8 did not provide a mechanism for code annotations, so I had to create one. The version in SML/NJ 110.20 did allow annotations, but they were not
propagated to the final machine code generation stage, and I had to make modifications to allow that.

Although I did not thoroughly test the results of partitioning the registers I did try it briefly, modifying the register allocator in MLRISC to segregate values among two different register partitions was easy, due to the modularity of the MLRISC register allocator.

The result of register allocation in MLRISC is not a modified code stream with new registers, but a function which maps pseudo-registers onto machine registers. This was useful, since I associate at a high level each pseudo-register with a value type corresponding to whether it is a live object pointer or not, and thus I did not have to modify the allocator in the non-partitioned case to propagate such values to the machine registers.

6.2 The abort technique on the Sparc architecture

The abort technique allows for quicker interrupt response time than run-to-completion, and doesn’t suffer from the page-fault problem (see section 2.4.1). Modifying SML/NJ to support it required work to many parts of the compiler, all in the backend, and parts of the run-time system interface.

This implementation was done on SML/NJ 110.8. Although parts of the backend changed a great deal from 110.8 to 110.20, none of those changes affected the compiler’s interrupt-handling techniques and none seriously affected the implementations of abort or static-arena style, as far as I could determine.

I modified only the parts of the run-time system and backend which were required to get the abort technique running on the Sparc architecture. Modifying the parts of the compiler which support other architectures would have been no more difficult, but it would have required more time.

6.2.1 Allocation

I used the abort technique on allocations. The allocations were marked as dangerous using the fp-locking scheme. For SML/NJ this meant that not many changes were necessary to the code generator. The MLRISC code was modified to emit a lock instructions (fp <- fp+1) at the top of each allocation sequence, and to then unlock it at the next frontier advance. In cases where allocations occurred sequentially, the unlock and lock sequences canceled each other out. That is, when the clear-and-advance-frontier instruction—fp <- fp+7 for example—was followed by a lock instruction, fp <- fp+1, it became fp <- fp+8.

The transactions were still separate from the abort point of view however. An interrupt during the second allocation caused an abort back to the top of that allocation only. The abort locations were communicated to the interrupt system via annotations.

6.2.2 Annotations

In this implementation three types of information were conveyed via per-instruction annotations: Abort Offset (AO), Heap Availability (HA), and Register Liveness (RL). Each datum took one word, for a total of three words of annotation per instruction. In the following simplified code examples, I will represent general purpose (non-dedicated) registers as a-z.

Figure 6-1 presents an example block in abort style on the Sparc. The three types of annotation are shown for each instruction. Note how the register-liveness mask remains constant through the transaction, and that p isn’t considered live until after the allocation is complete. The abort offset (AO) represents the number of instructions which need to be ‘undone’ if the thread is resumed.

It is possible that the dangerous code could be marked with annotations, and in fact the abort offset does this nearly exactly. I could have, for instance used the abort offset -1 to indicate that an abort was not necessary. However, I chose to use fp-locking because it speeds up the interrupt code when an abort isn’t necessary. In those cases there is no need to check the annotation stream.
6.2.3 Mutations

Mutations were left implemented as store lists on the heap, just as SML/NJ does by default. As discussed in section 4.3.2, making store-list mutations atomic in abort style requires specific work to undo the store list update. The instruction sequence is as follows.

<table>
<thead>
<tr>
<th>Code</th>
<th>AO</th>
<th>HA</th>
<th>RL</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp fp, lp</td>
<td>0</td>
<td>0</td>
<td>a,b</td>
</tr>
<tr>
<td>bgu gc</td>
<td>0</td>
<td>16</td>
<td>a,b</td>
</tr>
<tr>
<td>add fp, 1, fp</td>
<td>0</td>
<td>16</td>
<td>a,b</td>
</tr>
<tr>
<td>st a, [fp-1]</td>
<td>1</td>
<td>16</td>
<td>a,b</td>
</tr>
<tr>
<td>st b, [fp+3]</td>
<td>2</td>
<td>16</td>
<td>a,b</td>
</tr>
<tr>
<td>sub fp, 1, p</td>
<td>3</td>
<td>16</td>
<td>a,b</td>
</tr>
<tr>
<td>add fp, 7, fp</td>
<td>4</td>
<td>16</td>
<td>a,b</td>
</tr>
<tr>
<td>ld [a+4], c</td>
<td>0</td>
<td>8</td>
<td>a,p</td>
</tr>
<tr>
<td>add fp, 1, fp</td>
<td>0</td>
<td>8</td>
<td>c,p</td>
</tr>
<tr>
<td>st c, [fp-1]</td>
<td>1</td>
<td>8</td>
<td>c,p</td>
</tr>
<tr>
<td>st p, [fp+3]</td>
<td>2</td>
<td>8</td>
<td>c,p</td>
</tr>
<tr>
<td>sub fp, 1, q</td>
<td>3</td>
<td>8</td>
<td>c,p</td>
</tr>
<tr>
<td>add fp, 7, fp</td>
<td>4</td>
<td>8</td>
<td>c,p</td>
</tr>
</tbody>
</table>

Heap limit check
Lock fp
Initialize object
Advance frontier
Some computation
Lock fp
Initialize object
Assign object pointer
Advance frontier

The only instruction which might need undoing is sub fp, 1, stp. The interrupt system detects that case by comparing fp - 1 to stp. If it is equal, then it resets stp by executing ld [fp+3], stp before resuming the thread.

6.2.4 Results

The set of benchmarks used to compare the implementations presented in this paper with the base SML/NJ implementation are the de facto standard for SML/NJ projects. Although the benchmarks do not measure every aspect of performance and do not test every feature of the language, they at least provide a standard set of reasonable programs.

The numbers I obtained were not very precise. The benchmarks were measured using SML/NJ timer functions, and the numbers varied more than I had expected. Getting accurate estimates of the overheads on a specific program were therefore difficult, but averaging the values over several runs provided an approximation.

Some run-time overhead resulted from fp-locking and restricting the allocation sequences to allow abort. Table 6.1 shows how each benchmark was affected by the different code style. The commit column shows the results of the stock SML/NJ (run-to-completion) style, while the abort column shows the modified SML/NJ compiler results. All of the times were measured in seconds. The average change was a nearly 8% increase in run times.
<table>
<thead>
<tr>
<th>name</th>
<th>commit</th>
<th>abort</th>
<th>diff(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>boyer</td>
<td>31.393</td>
<td>34.060</td>
<td>+8.50</td>
</tr>
<tr>
<td>life</td>
<td>22.943</td>
<td>25.220</td>
<td>+9.92</td>
</tr>
<tr>
<td>knuth-bendix</td>
<td>20.547</td>
<td>22.603</td>
<td>+10.00</td>
</tr>
<tr>
<td>lexgen</td>
<td>22.220</td>
<td>23.543</td>
<td>+5.95</td>
</tr>
<tr>
<td>mlyacc</td>
<td>19.683</td>
<td>21.667</td>
<td>+10.08</td>
</tr>
<tr>
<td>vliw</td>
<td>22.590</td>
<td>24.143</td>
<td>+6.87</td>
</tr>
<tr>
<td>fft</td>
<td>20.427</td>
<td>23.667</td>
<td>+15.86</td>
</tr>
<tr>
<td>logic</td>
<td>21.983</td>
<td>25.200</td>
<td>+14.63</td>
</tr>
<tr>
<td>simple</td>
<td>24.263</td>
<td>26.290</td>
<td>+8.35</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>20.087</td>
<td>21.800</td>
<td>+8.53</td>
</tr>
<tr>
<td>ray</td>
<td>23.700</td>
<td>24.820</td>
<td>+4.72</td>
</tr>
<tr>
<td>barnes-hut</td>
<td>24.127</td>
<td>26.337</td>
<td>+9.16</td>
</tr>
<tr>
<td>average</td>
<td>22.830</td>
<td>24.946</td>
<td>+9.38</td>
</tr>
</tbody>
</table>

Table 6.1: Comparison of run times on the Sparc architecture between the standard SML/NJ implementation and an abort style implementation

6.3 Static arenas on the IA32 architecture

The static-arena technique improves upon the abort technique mentioned above by placing less work on the user code and interrupt system and more work on the garbage collector. We first implemented static-arena style on the IA32 architecture in SML/NJ 110.20.

6.3.1 Allocation

The SML/NJ compiler already initializes the heap in sequential order, and so modifying it to perform static-arena style allocations was straightforward. The allocation transactions were split into allocation and initialization sequences, and the allocation sequences were combined. The code generator first performed analysis on the code to determine the necessary size of the static arena, then inserted the appropriate allocation sequence, and then generated the rest of the extended basic block.

The allocation sequence was implemented two ways. First I tried a sequence which uses the IA32's atomic exchange-and-add instruction (xaddl).

- movl 12, sap
- xaddl sap, fp

| Store the integer 12 in the sap |
| Allocate a 12-byte arena |

Then, for comparison, I tried an fp-locking sequence.

- incl fp
- movl fp, sap
- addl 11, fp

| Set the low bit of the fp |
| Assign the sap |
| Clear the low bit and advance the fp |

The first required no work on the part of the interrupt handler, while the second required the re-execution of the movl fp, sap instruction if the low bit of fp was set at interrupt time. Additionally, the xaddl sequence left the sap properly aligned, pointing to the first word in the arena, but the fp-locking sequence left the sap offset by one byte. Although this was not a problem in implementation, replacing the movl instruction with leal fp[-1], sap would have left the sap aligned. For the rest of this section I will act as if the sap is aligned with the beginning of the arena, as if I had used leal. The difference is not significant, since the arena is accessed via offsets which can be adjusted statically to account for the misalignment.
6.3.2 Registers

Register partitioning would have been impossible in this version of the implementation, because the IA32 architecture only presents eight general-purpose registers. Partitioning those registers would significantly restrict the compiler. SML/NJ already dedicates two of those registers (di and sp, for fp and stack respectively).

Since the register pool was small to begin with, it was determined that dedicating more registers would result in poorer performance. Therefore fp and sap could not both be assigned to registers, and so fp was reassigned to a location on the stack, while di was rededicated to sap. Putting fp in memory may have been a fatal mistake for this implementation, since performing memory operations in the static-arena allocation turned out slower than I had expected.

6.3.3 Annotations

The IA32 architecture has a variable-length instruction representation which not only makes code scanning difficult, but also makes annotation formatting difficult. It is useful to be able to quickly determine the correspondence between an instruction address and an annotation address, so that annotation decoding can begin immediately. With variable-length instructions an annotation ‘table’ cannot be made to correspond with instruction addresses. We decided upon an annotation format with variable-length annotation representations, so that addresses correspond easily.

It is important to note that the annotation data does not always change on a per-instruction basis. In fact, some types of annotation changes are mutually exclusive, leading to many of the representation optimizations below.

Register liveness, however, changes nearly once per instruction. On the IA32 architecture SML/NJ uses all eight of the general-purpose registers. However, two of those registers are dedicated to the stack pointer and the frontier pointer (changed in static arena style to be the static-arena pointer). These dedicated registers need not be considered in the register-liveness information, and so only six bits are necessary.

When the garbage collector is ready to look up the information it needs, it finds the appropriate address in the annotation stream and begins to scan backwards, reading the bytes as if they were instructions describing the state of the interrupted thread. When it has collected all of the information necessary, it stops scanning and proceeds to promote the stack frame. We view each annotation as describing an event to the collector. If enough of the most recent events are known, the collector can deduce the information. These events correspond to things such as a change in register liveness, the initialization of a word in the static arena, a register spill, a mutation, or even a branch.

The following events were designed so that they never occur simultaneously in the code stream, and thus at most one needs to be encoded in an annotation. The first three are only needed when the spill area is not partitioned.

**Tagged Spill** This indicates that a live object pointer (or a live tagged value) has been spilled. Register reloads are not annotated, but simply result in a change of the liveness mask. In that case, the collector might consider a spilled pointer live when it is not, but that is acceptable. This annotation requires an argument to indicate the offset in the spill area.

**Untagged Spill** This indicates that a word in the spill area has been replaced by an untagged value, so that location should not be scanned by the garbage collector. In a sense, this overrides a prior Tagged Spill annotation.

**Float Spill** On the IA32, the spilling of a float value has to be considered a separate event because it creates an untagged double-word in the spill area.

**Sssp Init** Since the sssp is not dedicated to a register it might move from register to register, or be spilled and reloaded. This annotation indicates that it is now in a register. An argument to the annotation indicates which register.

**Sssp Spill** In case the sssp is spilled, this annotation indicates (with argument) which spill location now contains it. This annotation and the Sssp Init annotation cancel each other out. This
annotation is needed even when the spill area is partitioned (in that case the sssp would be spilled in the untagged or non-gc-traced partition).

**Mutation**  This annotation acts as the third step in the three-step mutation. It coincides with the mutating instruction and indicates to the collector that the store list entry in question can be purged.

**Arena Object**  This annotation indicates that an object has been completed on the arena (a pointer to the object has been assigned). Since arenas are initialized sequentially in this implementation, this implies that all previous objects have been completed.

**Arena Word**  This annotation indicates that a word within an arena object has been initialized. Since arenas are initialized sequentially in this implementation, this implies that all previous words in the arena have been initialized.

**Arena Size**  When an arena is allocated, this indicates the size.

**Subvector Size**  When a store subvector is allocated, this indicates the size.

**Begin Block**  The collector scans backward to find events, and it will eventually hit the annotation corresponding to the top of the basic block, at which point it should stop scanning. This event marks that point.

None of these events can coincide. Additionally, these events represent all of the information needed by the garbage collector except for the register liveness. An annotation is then composed of a register-liveness mask and optionally one of the above events. Although some of the events above never coincide with a register liveness change (i.e., the liveness information can be retrieved from a preceding instruction), it happens to be easiest to include the register information with them.

If we could encode each annotation in a single byte, then we would not have to worry about how to align the annotations with the instructions to provide address correspondence. We would only have to pad the annotations with a type of no-operation for instructions which take multiple bytes. However, many of the above events require an integer argument which may need multiple bytes to represent. Therefore, we needed to make use of the extra space ‘provided’ by multiple-byte instructions.

Note that some of the events described above occur only with a certain class of machine instructions. For example, Mutation events only occur with an instruction which stores a value in memory, such as a[8] ← b. Other kinds of instructions only occur within some known sequence, like Arena Size, which occurs in the arena allocation sequence.

**Annotation scanning**

The garbage collector, when it comes upon an interrupted thread, scans the annotation stream backwards beginning at the byte corresponding to the first byte of the interrupted instruction. I will not describe exactly how the annotations were encoded (which is beyond the scope of this document), but I will describe below how the garbage collector reacts to what it finds on the annotation stream.

Figure 6-2 shows an example block on the IA32 with the associated event descriptions. Each instruction contains at least one annotation indicating the state of the registers. Instructions which perform one of the key events contain an additional annotation describing the event.

Since this short block performs both an allocation and a mutation, then both a heap limit check and store-vector limit check are needed. The rest is a natural sequence, but note that the subvector allocation is done between the arena allocation and initialization.

If an interrupt and collection occur at instruction 10, then the collector scans the annotations starting at the Arena Word(4) event. Since the Arena Word annotation covers all Arena Word events before it, the collector can ignore the Arena Word(0) annotation.

The subvector and arena sizes are noted, and then the collector stops scanning (at Begin Block) and then traces the pointers which it has learned are live. This includes the live registers, obtained
from the annotation on instruction 10, and two words in the arena. All of the words in the subvector are checked, but none are removed since no Mutation events were seen.

In this implementation, the runtime always initialized the store vector to zeroes, so that the collector can recognize which entries have been initialized.

If an interrupt and collection occur at instruction 12 the same work is done, with the exception that since this time an Arena Object(4) annotation was found, none of the words at offset 4 or lower are traced. In this case the arena words are not traced until p is traced.

The store vector entry is saved in the stack frame so that it can be restored when the thread is resumed.

6.3.4 Results

The central problem with this implementation was that moving the fp into a memory location was extremely costly. The only instructions which actually used fp in the user code were in the static-arena allocation which occurred once per basic block, but that was enough to slow the code down significantly.

In the end, the xaddl instruction was not worth using. The static-arena allocation using it created a 20% run-time overhead. The xaddl instruction was simply too slow on a memory operand.

Allocation using Fp-locking was an improvement, and resulted in a 10% overhead, but again this was not what I had hoped for. In order to keep a competitive run-time speed, I should have kept the fp in a register. This would have resulted in 5 allocatable registers.

Static arena style also increased heap usage in some benchmarks, as I will discuss in the results part of the next section.

6.4 Static arenas on the Sparc architecture

The next implementation was done on the Sparc, which has enough registers to allow dedicating one each to fp and sap. In fact, we also dedicated a register to the store-vector frontier pointer sfp.

6.4.1 Annotations

The implementation was almost identical to the IA32 static-arena implementation, except that since the Sparc does not have variable-size instructions, the annotation formatting could be simplified. On the other hand, the larger number of registers meant that more space was needed for the register mask. After dediating registers, there were 19 allocatable registers. In order to include the event
descriptions and event arguments (e.g., Arena Size(64)) together with the register mask in one word per instruction required some tricks.

Instead of encoding the register mask itself into every instruction, I encoded only a 'delta' or a description of the change in the register mask since the last instruction. Since each instruction changes at most one register (with few exceptions) then most instructions did not need a full register mask. The instructions which did need the full mask did not have events associated with them.

To illustrate the differences here is the same block as before, this time in Sparc code:

```
1 cmp fp, 1p  Begin Block
2 bgu gc  Heap limit check
3 cmp sf p, s1p  Store-vector limit check
4 bgu gc
5 add fp,1,fp
6 sub fp,1,sap
7 add fp,7,fp  Arena Size(8)
8 add sfp,1,sfp
9 sub sfp,1,sssp
10 add sfp,3,sfp  Subvector Size(4)
11 st a, [sap+0] Arena Word(0)
12 st b, [sap+4] Arena Word(4), b dead
13 mov sap, p Arena Object(4), p live
14 st a, [sssp+0] Assign object pointer
15 st p, [a+0] Create subvector entry
```

All of the instructions with no event listed are annotated with a full register mask.

The scanning is much the same as before, except that the collector starts scanning on the word corresponding to the instruction before the interrupted instruction. If the interrupt was at instruction 14, the full register mask isn't known until the collector scans to the mask at instruction 9. It must then take into account the changes at instructions 12 and 13.

### 6.4.2 Results

The Sparc implementation provided the expected improvements over the IA32 implementation. Because there were more available registers, both sap and fp could be in a register. On average, the static-arena version performed 0.63% better than the standard SML/NJ version. It seems clear that compiling in static-arena style does not significantly affect the run time of the program.

The safe-points style of the original SML/NJ version required heap limit checks in each extended basic block (to handle interrupts even in non-allocating loops). The static-arena style did not require those checks because interrupts were allowed everywhere. For some of the benchmarks, the removal of those checks resulted in an improvement of the run time.

It was interesting to see the heap usage of the static-arena version as compared to run-to-completion (standard SML/NJ). The allocations were combined at the top of the extended basic block at the static-arena allocation, so that multiple objects were allocated in each arena. Since the extended basic blocks included conditional branches, this means that occasionally not all of the arena was used. The static-arena allocation ensured that the maximum amount of space needed was available, and thus resulted in some wasted space in some blocks. The average of the percent heap usage differences was 154%. The mandelbrot benchmark experienced a drastic change in heap usage, even though the run time was improved. The average of the differences not including the mandelbrot result was 41.36%.

The use of a store vector instead of a heap store list improved the heap usage of some benchmarks.
<table>
<thead>
<tr>
<th>name</th>
<th>run1</th>
<th>run2</th>
<th>diff(%)</th>
<th>gc1</th>
<th>gc2</th>
<th>diff(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>boyer</td>
<td>69.54</td>
<td>67.87</td>
<td>-2.40</td>
<td>7860</td>
<td>9909</td>
<td>+26.07</td>
</tr>
<tr>
<td>life</td>
<td>81.12</td>
<td>76.16</td>
<td>-6.11</td>
<td>7868</td>
<td>7991</td>
<td>+1.56</td>
</tr>
<tr>
<td>knuth-bendix</td>
<td>55.69</td>
<td>55.79</td>
<td>+0.18</td>
<td>19885</td>
<td>21397</td>
<td>+7.60</td>
</tr>
<tr>
<td>lexgen</td>
<td>64.14</td>
<td>64.67</td>
<td>+0.53</td>
<td>1830809</td>
<td>1830705</td>
<td>-0.01</td>
</tr>
<tr>
<td>mlyacc</td>
<td>46.97</td>
<td>46.69</td>
<td>-0.60</td>
<td>358167</td>
<td>358388</td>
<td>+0.06</td>
</tr>
<tr>
<td>vliw</td>
<td>55.12</td>
<td>51.92</td>
<td>-3.20</td>
<td>661589</td>
<td>661184</td>
<td>-0.06</td>
</tr>
<tr>
<td>fft</td>
<td>56.56</td>
<td>55.50</td>
<td>-1.87</td>
<td>5533</td>
<td>19038</td>
<td>+244.08</td>
</tr>
<tr>
<td>logic</td>
<td>69.25</td>
<td>70.03</td>
<td>+1.13</td>
<td>6923</td>
<td>7104</td>
<td>+2.61</td>
</tr>
<tr>
<td>simple</td>
<td>40.52</td>
<td>41.38</td>
<td>+1.88</td>
<td>9895</td>
<td>11893</td>
<td>+20.19</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>95.55</td>
<td>90.48</td>
<td>-5.31</td>
<td>1979</td>
<td>29590</td>
<td>+1395.20</td>
</tr>
<tr>
<td>ray</td>
<td>42.31</td>
<td>45.72</td>
<td>+8.06</td>
<td>9081</td>
<td>16639</td>
<td>+83.23</td>
</tr>
<tr>
<td>barnes-hut</td>
<td>48.50</td>
<td>49.59</td>
<td>+2.25</td>
<td>7954</td>
<td>13491</td>
<td>+69.61</td>
</tr>
<tr>
<td>average</td>
<td>60.44</td>
<td>59.65</td>
<td>-0.83</td>
<td>243961.92</td>
<td>248944.08</td>
<td>+154.18</td>
</tr>
</tbody>
</table>

Table 6.2: Uninterrupted run-time performance of static-arena style versus standard SML/NJ run-to-completion. In this table, run1 and run2 are the run times of the benchmark on standard SML/NJ and on static-arena SML/NJ, respectively. The gc1 and gc2 columns are counts of minor collections done during those runs.
Chapter 7

Future Work

7.1 Better annotations

Using annotations to save work at run time and at interrupt time is an interesting part of this research. The disadvantages of using annotations are an increase in code size and the overhead of reading and parsing the annotation data.

The encoding of the annotations is a tricky issue, and it is not entirely clear how much is saved by reducing the size of the annotation data. For example, my implementations of abort and static-arena methods in SML/NJ used a one-to-one correspondence between instruction address and annotation address. It is possible to make the annotations smaller by observing that since the information is not dense (some instructions have no interesting annotation data), it can be compressed. The side effect of that would be an increase in the amount of time required by the garbage collector to read the annotations. For a given system, the designer has to decide where to balance those costs.

For example, I mentioned earlier that a PC table could indicate which instructions in the program were dangerous in run-to-completion style. On a Sparc, assuming each allocation transaction contains at least 3 instructions, the PC table would contain at most 2/3 the size of the code itself. I expect that on average the actual result would be smaller, 1/4 of the code size or smaller.

7.2 Selective register partitioning

Register partitioning is very attractive in an interruptable system because the register masks are no longer needed. However, restricting the use of the registers may result in extra spills, depending on the size of the partitions and the needs of the program.

Developing a compiler which statically determines the best partitioning for a given segment of the program would help to reduce the costs. The compiler could even decide that for some section of the code, the optimal partitioning is no partitioning at all. As in the heap availability requirement discussed in section 5.1.4, the partition information could be stored at run time in a special location in memory, or it could be included as an annotation with the code.

7.3 Other kinds of collectors

Although stop-and-copy collectors are prolific, there are other kinds of garbage collectors [9] which use different strategies and have different heap invariants. Although the run-to-completion and abort styles probably apply to any collector which does not run in parallel with the user threads, the static-arena technique is less likely to be so generally applicable. It would be interesting to see how the methods in this document apply to other kinds of collectors.
7.4 Operating system integration

The techniques discussed in this document all require integration and communication between the compiler and the run-time system. This is a necessity in a modern garbage-collecting system. This kind of integration can lead to performance improvements if it is extended to the operating system. With proof-carrying code, we could run user code without the layers of protection which are currently required. The ability to run user code in what would today be called kernel mode has interesting implications.

For example, we could use guard pages to remove the need for heap limit checks. If we place the heap such that the page immediately after the heap limit is not in memory, then we would detect heap overflow with a page fault. The garbage collector would then treat the last (incomplete) object just as it treated the partially-initialized objects in a static arena, except that the object would have to be placed at the frontier of to-space. That removes the need for inserting heap limit checks in the code, but requires the heap initialization to be sequential.

7.5 Finer measurements

The results given in chapter 6 were not very precise. It would be interesting not only to get good data comparing the run times of various implementation styles, but to also obtain other more detailed numbers.

Interrupt latency and overhead are two very important measurements, and ones which should make apparent the differences between the techniques. Interrupt overhead comes from work done by the interrupt system, and from work done by the garbage collector in the presence of interrupted threads.

7.6 Predicate registers

The new IA64 architecture boasts a set of one-bit ‘predicate’ registers which can be used for conditional execution without branching. In addition to their usefulness as flags (one would no longer need to set aside a register or memory location for some flags) the predicate registers might also provide new ways of implementing the techniques discussed in this document.

Predicate registers could obviously work to communicate information in a program in the same way as regular registers. That is, executed code can store information in those registers for ‘downstream’ code to read. We could also use the registers to send information to the interrupt system or the garbage collector which might be activated during an interrupt. Additionally, the interrupt system and garbage collector can use dedicated predicate registers to send information to the user code itself.

For example, we can predicate the allocation instructions on a particular predicate register—that is, we encode those instructions so that the processor executes them only if the register is set, otherwise it instead executes a no-op. If the allocation operations are so encoded, then the interrupt system can effect an abort on the user code by simply turning off the register. Later allocation instructions would be skipped, and a conditional branch in user code could return to the start of the allocation.
Chapter 8

Conclusions

It is important to view the heap operations allocate, collect, and mutate as transactions which must be atomic with respect to one another. These operations modify the heap in ways which make it inconsistent. If they are performed simultaneously, or one within another, then they may interfere with each other and damage the state of running threads.

When an interrupt occurs during the course of one of these transactions, there are three basic ways to ensure atomicity. These are the commit, abort and side-step methods:

- Complete and commit the current transaction, and handle the interrupt afterward
- Undo the changes made by the current transaction, and plan to restart it when the thread is resumed
- Give the interrupting thread a separate heap, so that the transactions do not interfere

The commit or run-to-completion method is generally the simplest method to implement, particularly using safe points. However, it suffers from a significant flaw. The run-to-completion technique is not capable of efficiently handling page faults. Because page faults must always be handled immediately, the interrupted code sequence cannot be completed. This means that no user code may be run during the interrupt. Furthermore, the remedy for a page fault is an expensive operation: The missing data must be loaded from disk. The entire system must stop for the duration of the interrupt.

The abort technique allows fine-grained interrupts, and so does not have this problem. It is trickier to implement, because it cannot use the safe points optimization. The interrupt system needs to detect if a transaction was interrupted, which can be costly. The new fp-locking technique avoids this cost and simplifies the implementation.

Even more promising is the new static-arena technique for heap allocation and its counterpart for mutations, static store subvectors. These also allow fine-grained interrupts, but since they allow the thread to be resumed at the same point it was interrupted, no work needs to be undone at interrupt time. In static-arena style, the allocation is separated into atomic and non-atomic parts. The atomic part is very small:

\[
\begin{array}{l}
\text{sap} \leftarrow \text{fp} \\
\text{fp} \leftarrow \text{fp+8}
\end{array}
\]

Set the sap
Push the frontier forward

Because of its size, this sequence can be made atomic with very simple methods. The cheapest method is to simply compare the sap and fp values, since they are only equal between those two instructions, and perform a special-case abort inside the interrupt system. This abort is done by simply re-executing the sap \leftarrow fp instruction.

Static-arena style requires additional communication between the compiler and the garbage collector. Specifically, the collector needs to know the dimensions and the status of the static arena at each instruction. In fact, interaction between the compiler and the run-time system is an important
way to increase performance. These two parts of a language system already rely upon one another when the system uses a garbage collector: The compiler must design the code to maintain the heap invariants, and the run-time system must manage the heap between threads and perform collections properly.

Finally, three implementations of the fp-locking and static-arena techniques were performed on the SML/NJ language system. On the IA32 architecture, static-arena style may not be suitable because of the small number of registers, and so it is suggested that the fp-locking version of the abort technique be used. For the Sparc and the new IA64 architectures, static-arena style seems very promising.
Bibliography


[3] Lal George. Little has been published on Lal George's MLRisc code-generator. However, on-line documentation can be found at http://www.cs.nyu.edu/leunga/www/MLRISC/Doc/html/.


