Atomic Heap Transactions and Fine-grain Interrupts

Olin Shivers James W. Clark Roland McGrath
{shivers,sbj,roland}@ai.mit.edu

MIT Artificial Intelligence Laboratory

Abstract

Languages such as Java, ML, Scheme, and Haskell provide automatic storage management, that is, garbage collection. The two fundamental operations performed on a garbage-collected heap are "allocate" and "collect." Because the heap is in an inconsistent state during these operations, they must be performed atomically. Otherwise, a heap client might access the heap during a time when its fundamental invariants do not hold, corrupting the heap.

Standard techniques for providing this atomicity guarantee have large latencies and other performance problems that impede their application in high-performance, interrupt-laden, thread-based systems applications. In particular, the standard techniques prevent thread schedulers from switching threads on VM page faults.

We cast the space of possible implementations into a general taxonomy, and describe a new technique that provides a simple, low-overhead, low-latency interlock. We have implemented this technique in a version of SML/NJ, and, because of its applicability to thread-based systems, are currently implementing it in the scheduler of our raw-hardware SML-based kernel, ML/OS. Our technique can be extended to provide other atomic sequences besides storage allocation.

1 Introduction

Storage allocation and garbage collection can be difficult in the presence of interrupts and multi-threaded run-time environments. In this paper, we will discuss how this is done, and present a general way of looking at the requirements of automatic storage management and program concurrency. We'll illustrate with examples from existing language implementations. While our examples will be drawn primarily from Scheme and ML systems, the lessons are broadly applicable. Fast, low-overhead multi-threaded heap allocation is a fundamental technology for advanced programming languages—these results are just as relevant to Java, Modula-3, Dylan, Smalltalk, *et al.* as they are to ML, Haskell and Scheme. We'll examine ways that the operating system and the compiler can collude to make atomic heap allocation and interrupt handling cheap. Finally, we'll show a new technique for cheap heap interlock and discuss its implementation.

Program threads must allocate storage atomically. This is because the allocation heap is a shared resource, and during allocation the heap is in an inconsistent state. If we were to interrupt or suspend a thread in the middle of an allocation, then other allocating threads or the garbage collector could mangle the consistency of the heap.

For example, suppose we have a run-time heap that is managed by a simple stop-and-copy garbage collector [Wilson]. The run-time system maintains a *frontier pointer* into the heap giving the boundary between allocated and unused storage in the heap. Storage is allocated simply by incrementing this pointer and initialising the allocated block. With this arrangement, a thread does an allocation, or "cons" operation, with the following simple pseudocode:

```
1   If heap exhausted, trap to GC
2   r1 := heap pointer
3   increment heap pointer by 2 words
4   store CAR field into r1(0)
5   store CDR field into r1(4)
/*  Now r1 contains the allocated and
    initialised cons cell. */
```

- If the thread is suspended after instruction 1, some other thread will be able to allocate data, possibly exhausting the heap. So when resuming at instruction 2, this thread might actually allocate off the end of the heap, overwriting other data or causing an address violation.
- If the thread is suspended after instruction 2, some other thread may allocate from the heap. When our thread resumes, `r1` will point to already-allocated data, which will then be overwritten by the cons operation.
- If the thread is suspended after instruction 3, some other thread may exhaust the heap and trigger a garbage collection. But the cons cell in `r1` hasn't been initialised yet—the car and cdr fields contain random bits. The garbage collector will trace these pointers, possibly relocating and altering whatever storage to which they might be pointing.

In short, the heap is a shared resource that needs to be locked during consing. While our thread is performing the cons operation, no other heap clients—either other allocator threads, or the GC thread—can be allowed to operate on the heap. Heap transactions must be atomic.

Techniques for locking the heap all reflect the same basic fact of life: heap allocation is ubiquitous. If a thread is suspended while holding the heap lock, every other thread in the program will block at the very next allocation. This usually doesn't take long. For example, the SML/NJ [SML/NJ] implementation of Standard ML allocates procedure frames and boxes floating-point numbers in the heap. SML/NJ programs allocate a word of storage about every four instructions. So it's not very likely that any thread in the system is going to be able to complete more than a handful of instructions before blocking on the heap. SML/NJ is an extreme example, but the fact remains that programs written in most advanced languages cons very frequently, and it is thus unacceptable to allow a heap lock to become a system bottleneck.

The ubiquity of storage allocation has some implications:

- Acquiring and releasing the heap must be very cheap. A two-word cons operation can be implemented in four instructions, so a locking protocol that requires four instructions doubles the cost of consing. Heap allocation should be roughly competitive with stack allocation in implementations that do not use a stack. System calls are typically at least two orders of magnitude more expensive than this, so bracketing the allocation code with system calls to lock and unlock the heap will swamp the cost of the actual allocation with the cost of the locking.
- Because access to the heap is all but required for threads to proceed, threads cannot be suspended while holding the heap lock. There are several ways to arrange for this, which we can call "abort," "commit," and "side-step:"[1]
  - Abort: If the thread is suspended while holding the lock, the heap transaction can be aborted. The lock is released, and when the thread resumes, it must retry the allocation.
  - Commit: If the thread is suspended while holding the lock, the heap transaction can be run to completion before actually suspending the thread, thus releasing the lock.
  - Side-step: The heap can be restructured for parallel access, eliminating the bottleneck. As we'll see, however, this can only partially be done.

This taxonomy essentially covers all implementations of programming languages that provide garbage collection and allow interrupt handlers or other forms of program concurrency. Let's consider each of these approaches and their tradeoffs.

2 Aborting the transaction

An early implementation of SML/NJ used the following clever technique [LockFree]. Let us suppose we keep the current heap-allocation pointer in a register, `fp`. The heap is bounded above by a trap-on-write guard page, so no test is required for heap exhaustion; assume that cons-cell pointers are tagged by setting the two low bits to 11. Allocating a two-element record on a byte-addressable machine looks like this:

```
Register      r1: result cons
assignments   r2: car
              r3: cdr
              fp: heap frontier pointer

fp[4] := r3    ; store cdr (& trap if GC)
fp[0] := r2    ; store car
r1 := fp+3     ; ans is fp + type tag
fp := fp+8     ; bump frontier pointer
```

Now, this instruction sequence has an interesting property: *it can be aborted any time before the final* `fp` *increment by simply starting the whole operation over again.* That is, there is no need to "undo" or "roll back" any computations—we can abort the transaction at any point by simply walking away from it. This keeps the allocation sequence lean; there is no need to keep a log or other side-information to assist the roll-back operation. A key feature of this sequence is that the final operation is the one that commits the transaction to external visibility—the frontier pointer in the `fp` register is the state that is shared between different allocation operations in different threads. It is atomically updated by the last instruction in the sequence.

Aborting the transaction requires only that we reset the thread's resumption PC to the beginning of the allocation sequence. When a thread is interrupted, the interrupt handler examines the interrupted code. If the instruction sequence is a storage-allocation sequence, the handler scans backwards looking for the start of the cons operation, and the resumption PC for the suspended thread is reset to this point, thus releasing the implicit lock. Notice that cons operations have a very stylised form—they are the only operations that do a store to an offset from the `fp` register, for example. This makes it feasible for the interrupt handler to examine the code stream to check for an allocation sequence.

This "locking" protocol is absolutely minimal in some sense: it doesn't require any instructions at all on the part of the thread! An n-word block of memory can be allocated and initialised in $n+2$ instructions, which is competitive with stack allocation. It's a clever instance of having the compiler and the operating system work together to achieve very low-overhead storage allocation.[2] However this scheme has some severe problems:

- Interrupt overhead is increased a lot. Examining the interrupted instruction stream, parsing the instructions, and scanning backwards is a lot of processing. It also requires going to memory—and almost certainly missing cache on a Harvard architecture, since the instructions are not likely to be in the D-cache.

  Operating-systems research has shown the utility of lightweight, fine-grained interrupts occurring at sub-millisecond intervals [Qua]. The overhead of the abort technique makes interrupts very heavyweight. It is impossible to have schedulers that rely upon information gathered by low-overhead, fine-grained interrupts, or to have threads that service high-bandwidth real-time I/O sources, such as CD-rate audio streams, video, or other time-domain media. But these are precisely the kind of applications for which we'd like to provide robust implementations based on advanced languages.

[1] Or, backwards, forwards, and sideways.

[2] Compare this four-instruction cons to the cost of malloc'ing and initialising a two-word record in C. The C allocation will probably cost two or three orders of magnitude more than the ML one.

Note that interrupt *overhead* is what's increased, not interrupt *latency*. The thread doesn't have to be checked and possibly reset until it is about to be rescheduled—the interrupt handler is free to run immediately.

- It's tricky, very machine and operating-system dependent, and difficult to port. This, in fact, is what led to later SML/NJ implementations abandoning this approach in favor of simpler, less aggressive techniques.

3 Commit

An alternative to aborting an interrupted atomic transaction is to proceed and commit the transaction—run it to completion—before taking the interrupt. This is a very popular implementation technique, due to its simplicity. Among others, the T 3.0 implementation of Scheme [T, Orbit] uses a commit strategy, and it has also been adopted by later releases of SML/NJ, as well as the Berkeley Spur parallel microprocessor. The T and SML/NJ variants of this technique have interesting differences.

In T, consing is performed by calling a bit of hand-written assembler "millicode" using a custom subroutine linkage known to the compiler. The cons routine sets a lock bit in memory, and performs the cons. The operating system's asynchronous signals are all mapped to the same handler, which is another bit of hand-written assembler in the T run-time kernel. When an interrupt (a Unix signal) arrives, the handler code checks the lock bit. If the lock bit is clear, it is safe to interrupt, and the handler vectors out to the actual Scheme procedure registered for that signal. If the lock bit is set, the thread is in the middle of a cons, so the handler just sets a pending bit, stores the signal information, and immediately returns, resuming the thread. After the cons operation is completed, when the heap is returned to a consistent state, the epilogue of the cons routine clears the lock bit and then checks the pending bit. If an interrupt is pending, the cons routine then belatedly transfers to the handler code and the thread takes the interrupt. (This technique was later extended to a slightly more general setting by Bershad and others, for general atomic sequences [Unimutex], and in the operating-system community is sometimes referred to as "software interrupts.")

Another variant of run-to-completion is the technique of "safe-points." Interrupts are only allowed at particular compiler-selected safe points in the code, where the thread polls to check for pending interrupts. Current releases of SML/NJ use this technique. Storage allocation is open-coded. Interrupts are only allowed at the entrance to extended basic blocks (that is, a tree of code with the only entrance at the root). At the beginning of every block, there is code that checks to see if the heap has enough free storage to execute the block, jumping to the garbage collector if not. This check can typically be done in one or two instructions. Once execution gets past this instruction, the code tree is executed atomically, so all storage allocation within the code tree proceeds with no further limit checks.

When a signal arrives, the interrupt handler resets the heap-limit register to make it appear to the thread that the heap is exhausted. The signal is recorded, and the thread resumed. The next heap check fails, and the garbage collector is responsible for dispatching to the deferred interrupt handler instead of initiating a garbage collection.

SML's safe points are an interesting contrast with T's locking. T code can be interrupted anywhere except at specific points (inside the allocation millicode); SML can *not* be interrupted anywhere except at specific points (basic block entries). The Berkeley Spur system used hardware assist to provide a system more or less equivalent to T's.

The safe-points approach has some advantages relative to interrupt-anywhere implementations, such as abort&retry:

- **Reduced GC-information requirements**
  When a copying garbage collector starts a collection, it needs precise information about which values in the register set are live pointers that need to be traced, copied, and relocated, and which values are non-pointers which should be left unaltered.

  Any time a thread is suspended, some other thread or interrupt handler could trigger a garbage collection. This means the garbage collector must be able to recover register-usage information about every code point at which a thread can be suspended. A fine-grained interrupt model allows any instruction to be a suspendable code point. This means the garbage collector must be able to determine for every instruction what the register usage is. This is a lot of extra information to keep around, unless we choose to statically partition the registers into traceable and non-traceable sets. (If we also allow for the possibility of "derived pointers," *i.e.*, values that point into the middle of data structures, the register-annotation needs increase even further.)

  A safe-points approach, like SML/NJ's, allows thread suspension only at a small number of points. The compiler records a register-use mask only for these safe points. By increasing the granularity of thread suspension, the space overhead for GC information is reduced a great deal.

- **Tolerant of imprecise interrupts**
  A general feature of a safe-points interlock is that register state is not changed while the thread is suspended by a real hardware/OS interrupt (*e.g.*, which may have subsequently triggered a garbage collection). Processors that make it hard for user code to capture and manipulate the processor state of an interrupted computation can make it impossible to correctly forward all pointers at GC-time. By polling for interrupts at safe-points, we can guarantee the processor state is in a well-known, easily-understood configuration.

On the other hand, all run-to-completion techniques have some major disadvantages, which are particularly serious for the safe-points approach.

- **Increased interrupt latency**
  Deferring interrupts, obviously, increases the time to respond to an incoming interrupt. This is most serious in the safe-points approach. In T's approach, interrupts are only deferred if they occur during an allocation, but in SML/NJ's safe-points approach, interrupts are always deferred to the end of the basic block. Furthermore, if we try to drive down the cost of the polling by polling less frequently, we drive up the latency (Feeley has treated the optimisation of this tradeoff in the context of his Gambit Scheme system in some detail [Feeley].)

- **Page fault**
  A major flaw with run-to-completion is that a thread cannot always immediately proceed forwards. If a thread page-faults while holding the heap lock, other threads cannot run while the missing page is being brought in from disk. The only way to release the heap lock is to run forwards—and that requires the missing page. So, even though there may be dozens of threads ready to run, the system remains completely locked for the milliseconds it takes to do disk I/O.

  This problem could be finessed in the millicode approach by requiring the OS to lock the allocation millicode into main memory. This cannot be done in the safe-points approach since the inlined allocation sequences occur throughout the code.

  This is the major advantage that abort&retry has over safe-points. The abort&retry allocation sequences are carefully crafted so that they can be trivially aborted, even in situations where the process cannot proceed forwards. So we can use abort&retry in systems where the thread-scheduler handles page faults by switching threads. As our research group works with an experimental SML-based operating-system kernel, this is critical for our needs. This is the principle reason why it is worth dealing with the imprecise-interrupt difficulties raised by allowing interrupts to occur anywhere.

4 Abort revisited: a new technique

The problems with abort&retry are that interrupt overhead is too high, and the implementation is too complex. The problems with run-to-completion are that interrupt latency is too high, and the bad interaction with page faults. All of these problems can be fixed with an alternative lightweight heap interlock, at a cost of one instruction per allocation. The interlock depends upon the fact that the heap frontier pointer is kept in a register, and can thus be updated atomically with no overhead. The idea is to keep the lock bit in the low bit of the register that holds the frontier pointer. Heap allocation is typically double-word aligned, so the bottom three bits are always 000, and thus available to be used by the compiler. We will use one of these bits: if the low bit is set, then the heap is locked. A transaction is aborted by clearing the lock bit and resetting the thread's PC to the beginning of the transaction.

Assume the machine is byte-addressable, the heap is word-aligned, cons cells are tagged with the low two bits 11, and that heap overflow is detected with a trap-on-write guard page. Here is what a two-word cons operation looks like in this scheme:

```
Register       r1: result cons
assignments    r2: car
               r3: cdr
               fp: heap frontier pointer

fp := fp+1      ; engage interlock          (*)
fp[3]  := r3    ; store cdr (& trap if GC)  (*)
fp[-1] := r2    ; store car                 (*)
r1 := fp+2      ; result = fp + tag - lock  (*)
fp := fp+7      ; commit op & clear interlock
```

The heap is locked anytime the frontier pointer's low bit is 1. When a thread is interrupted, the handler checks `fp`. If the low bit is set, the thread's resume PC must be reset to the beginning of the sequence. So if we interrupt after any of the `(*)` instructions, we resume at the beginning.

The fundamental trick is that we *simultaneously* clear the interlock and commit the transaction with one, atomic instruction (`fp := fp+7`). It can never be the case that we've released the lock but haven't committed the transaction, or *vice versa*. Since a heap allocation has to bump the frontier pointer anyway, that instruction comes for free, so the whole locking overhead is one instruction—the initial increment that engages the interlock. Even better, it is a simple register operation that doesn't require leaving the processor and going to memory.

Now our handler overhead is very low: just check one bit of a register. The handler doesn't have to go to memory for the check. It doesn't have to do any parsing or backwards scanning of the interrupted instruction stream—none of the high-overhead, hard-to-port complexity that the first interlocking scheme required.

There are several ways the interrupt handler can determine the restart address. One technique is to attach a sorted vector of restart addresses to every unit of compiled code. When the interrupt handler has to break a lock and reset a thread, it simply does a fast binary search on this table. The binary-search reset code doesn't actually examine the instruction stream, so it doesn't depend on the particulars of the instruction set. This makes it much more portable.

We can shift the expense from the interrupt handler to the allocator by alternately requiring the cons sequence to initially store the retry address in some standard register or memory location where it can be retrieved quickly if the interrupt handler decides to abort the transaction. This lowers the cost of the abort, at the price of adding one cycle to each cons operation. Note that the retry store is not part of the atomic sequence—it can be hoisted, scheduled, removed from loops or otherwise shifted to some convenient location that dominates the transaction. The retry store doesn't have any data dependency on any other instruction—the retry PC is only ever fetched by the interrupt handler.

5 Side-step and PCLSR

The final possibility is to side-step the problem entirely: restructure the heap data structures to eliminate the need to serialise access to the heap. Then threads do not need to lock the heap at all.

We can accomplish this by giving each thread its own chunk of storage for allocation. When a thread exhausts its private allocation area, it gets another from the global heap. Now, threads only need to serialise when they are getting private allocation areas from the global heap, which should be two or three orders of magnitude less frequently. The compiler can arrange to allocate only small structures from the private area; large structures can be allocated from the global heap. This decreases internal fragmentation, and still amortises the cost of heap synchronisation over large blocks.

This sort of approach is commonly used in garbage-collected thread packages implemented for multiprocessors. For example, the experimental concurrent collector implemented at DEC SRC on the Firefly [RTC-GC] and the general ML-threads package written for SML/NJ [ML-threads] both use this idea.

Giving each thread its own private allocation area does not completely remove the need for synchronisation. Application threads can now allocate concurrently amongst themselves, but they must still serialise with the collector thread. When the garbage collector runs, every live thread in the address space must have its heap data in a consistent state. So we can't completely parallelise the heap. Threads must still take a lock when allocating from their private areas, using either the abort&retry or the run-to-completion techniques discussed earlier. What's new is that each thread effectively has its own distinct lock, which it shares with the garbage collector.

Because application threads are independent, they only need to have their locks broken when they are suspended in mid-cons *and* a GC happens, not each time they are suspended. If a thread is suspended while allocating data in its private area, and it is resumed before a GC occurs, the allocation can pick up right where it left off. Presumably, garbage collections happen much less frequently than interrupts, so the lock-breaking overhead associated with many, many thread suspensions vanishes.

With this technique, we can take the view that the garbage collector is the agent responsible for forcing the other threads to relinquish their heap locks. The run-time system can do this by maintaining an internal list of all live threads which the garbage collector can scan before starting its collection. However, this would increase the GC startup time, which is a problem for real-time incremental collectors. Maintaining the list also increases the amount of work that must be done when forking new threads.

However, if threads lock using abort&retry, then the collector can break locks on the fly, as it traces through the heap and discovers suspended thread activation records. This notion of forcing a thread into a consistent state on demand, providing an atomic view of critical operations, is called PCLSRing the thread, a term taken from the general "PCLSR" mechanism in the ITS operating system, and described in an unpublished but influential note by Bawden [PCLSR].

This approach spreads the lock-breaking work out incrementally over the entire collection process, and also eliminates the need to break the locks of threads that have become garbage, since the collector never sees them. It also interacts well with generational collectors: a generational collector can assume that only threads in the newest generation can hold locks. Threads copied to older generations have their locks broken. Whenever a thread is suspended, a new activation record is allocated in the newest generation.

PCLSRing cons operations depends critically upon using abort&retry. If heap locks were broken using run-to-completion, the garbage collector would be in the position of running threads forwards a little bit in the middle of garbage collection, while the collector has the heap in an inconsistent state. This is a lock conflict: the thread wants to run holding its heap lock while the collector simultaneously holds the lock.

Giving each thread its own private allocation area does have a drawback: VM thrashing. Multi-threaded applications commonly have many threads. Researchers at DEC SRC have reported that typical applications can have tens or hundreds of threads [Modula-3]. If Java succeeds as a serious commercial implementation language, this will become increasingly common. If each thread has a several-kilobyte private allocation area, the virtual memory system is in danger of being overloaded as the actual processors (or underlying OS threads) switch amongst the application threads, each allocating from a different region of virtual memory.

6 Implementation

We have built a prototype implementation of our lock-bit technique for SML/NJ running on a SPARC with George's MLRisc code generator [MLRisc]. We added a code-annotation facility to build the associated tables we needed containing PC restart addresses and register data.

While functional, our implementation is fairly crude. For example, we have made no effort to be clever about how we store our tables of retry addresses and other data. Our tables are large: For each instruction we store a retry address (zero for non-consing code), an integer specifying how many slots in the thread's block of temporary storage are being used for register spills, and the basic block's heap requirement, a value that is checked before resuming the code after an interrupt. This is a tremendously inefficient, redundant representation, quadrupling the size of the compiled binaries. The basic technique allows for much tighter, compressed information. (For example, Diwan, *et al.* have reported on various techniques to efficiently store fine-grain register information for garbage-collection purposes [Diwan$^+$]. A recent effort has shown the ability to reduce roughly equivalent annotation tables to 20% of code size for the x86 [JavaGC].) We did simplify our annotation requirements by statically partitioning the register set into traced, "descriptor" registers, and untraced, "raw" registers.

Our implementation inserts code at the beginning of every basic block to check for heap overflow; in addition, when the interrupt system resumes an interrupted thread, it also re-performs the heap-overflow check, driven by a PC annotation. This somewhat clumsy arrangement was simple to implement, given that the stock SML/NJ system came with the basic-block heap checks already in place. Our original intention was later to eliminate the space and time overhead of polling by using a trap-on-write guard page to detect heap overflow. (This is not, however, certain to be an improvement: standard operating systems typically make it fairly expensive to service memory faults from user code. Depending on the frequency of interrupts, the frequency of checks, and other parameters, synchronous checks can actually be cheaper.) However, as we'll discuss later, we have subsequently reworked the entire design to such a degree that we are abandoning this entire branch of code development for a fresh start on an aggressive, new implementation. So we left our original overflow-detection strategy in place and are moving on.

A final detail is that SML/NJ uses a fairly sophisticated generational garbage collector. Our technique for implementing the atomic write barrier required by the collector is discussed in a following section.

7 Performance

We ran several tests to compare the performance of our fine-grain interrupt, abort-based system with the stock SML/NJ commit system, which uses run-to-completion and polls for interrupts at every basic block boundary. Our tests were run on a 200MHz dual-SPARC system with 1024 Mb of memory running Solaris 2.5.1.

We compiled a suite of standard SML programs twice: once with the stock SML/NJ safe-points mechanism, and once with our abort mechanism. Table 1 shows the number of basic blocks in each program, the average basic block size (statically), the size of the largest basic block, the largest number of registers spilled by a basic block, the average number of bytes heap-allocated per block (statically), and the largest heap allocation in a single basic block.

We ran these programs several times with no signals, and then again, with very high rates of signal delivery, in order to compute the amount of time spent servicing interrupts with no-op signal handlers. However, our measurements had such variance as to be almost completely useless (table 2). In some cases, the interrupted process even ran faster than the uninterrupted process! About all we learn from these measurements is that the fine-grained abort machinery seems to be roughly comparable to the stock commit machinery. Without significantly more precise timing facilities than those provided by Solaris on a SPARC, it will be difficult to obtain higher-quality measurements.

Finally, we ran the programs with no interrupts at all simply to measure the locking overhead. Our fine-grained abort implementation necessarily runs slower than the stock SML/NJ commit strategy: we did not remove the per-basic-block heap-limit check, but we added one instruction of locking overhead per allocation, and increased register pressure by statically partitioning the register set. However, the numbers in table 3 show that the extra cost is fairly cheap. We caution against drawing much from these early numbers beyond the observation that the allocation mechanism is currently roughly comparable to SML/NJ's stock system. We have hopes that our next implementation, described in section 9, will actually be able to provide fine-grain interrupt service more cheaply than the current stock SML/NJ commit system. Of course, we'll repeat that allocation schemes that use an abort model have one qualitative performance benefit relative to commit models: thread schedulers can be page-fault aware, switching threads to hide disk latency on page faults. The milliseconds spent waiting for one page to come in from disk can pay for a lot of locking overhead.

8 Variations

8.1 Generational collectors and store lists

There is a third heap operation we must consider, besides allocation and collection: side effects such as stores to mutable cells. This has extra implications in systems that use generational collectors. In these systems, updates to data structures living in old generations must be noted for consideration as extra roots by the collector. When the GC needs to collect a young generation, it scans these extra roots looking for old-generation data structures that have been altered to contain pointers to new-generation data. The two actions of logging the write for the GC and actually performing it must be atomic.

There is a variety of techniques for logging the write operation. In "card-marking" systems, side-effecting operations also set a bit in a table indicating a dirty page (or "card") that must be scanned by the GC. Other systems maintain a "store list" or "store vector" of addresses that have been written since the last GC; logging a write in a store-list system involves allocating a new entry recording the write and linking it onto the head of the store list. Still other systems use page-protection tricks and arrange for fault handlers to mark an entire VM page as dirty.

Store vectors and card-marking techniques can be made atomic with their accompanying side-effects using the same techniques we've used to make allocation atomic. However, store lists are not amenable to these techniques. The problem is that performing a store operation in a store-list system involves three actions: (1) doing the actual store, (2) allocating the new log entry, and (3) linking the new log entry onto the head of the GC's global store list. We are unable to express such a sequence without performing side effects that would need to be rolled back in the event of an abort. We've lost the magic property found in our allocation sequences of being able to abort the operation simply by abandoning the computation.

For our SML/NJ implementation on a SPARC, we managed a hack. The only instruction whose effect would need to be rolled back in the event of an abort is the instruction that moves the newly allocated store-list entry into the store-list register `sl`. We placed this instruction at the end of the sequence, just before the final commit/lock-clearing instruction. It's easy for the interrupt handler to detect the rare occasion that a thread is interrupted in the one position that would require rollback work, between the final two instructions. This is the case when `sl` = `fp` − 1 and the heap is locked. When this happens, the interrupt handler aborts the transaction by both undoing the update to `sl` and resetting the `PC`. (We could probably have done better by arranging for the interrupt handler to go ahead and commit the transaction in this case.) Here is the code for updating the cell whose address is in `r1` with the value in `r2`, using a store-list write barrier:

```
fp += 1          /* Lock heap.               */
*r1 := r2        /* Do the store.            */
fp[-1] := r1     /* Alloc & init new         */
fp[3] := sl      /*   store-list entry.      */
sl := fp-1       /* Update store-list reg.   */
fp += 7          /* Unlock heap.             */
```

In general, however, we do not recommend implementing write barriers for generational GC's with store lists; we intend to abandon this in our next implementation.

8.2 Parallelisation

A limitation of these schemes is that they only work for uniprocessors. Uniprocessors, however, are an extremely important case. You probably have one on your desk, and may carry around several on your person. We can adapt these techniques to multiprocessors by giving each processor its own allocation area, forcing parallel allocation to happen in different areas of memory. Doligez and Leroy, for example, have reported on other related issues surrounding storage management in a parallel implementation of Caml [ParaCaml].

8.3 Run-to-completion and lightweight locking

Note that we can apply our bit-locking trick to implement T's interrupt-anywhere, run-to-completion allocation, which would even allow us to open-code the allocation sequences. This is an improvement, but unless we can guarantee that the entire allocation sequence is present in main memory, still leaves open the issue of bad interactions with page faults and thread scheduling.

8.4 Hardware support

Some processor architectures have "predicate registers," a bank of one-bit registers that can be used to tag instructions.

If the indicated predicate register is set, the instruction is taken; if it is clear, the instruction is "squashed," or skipped. Predicate registers are useful to avoid branch stalls, allowing one to schedule small if-then-else-join control flow graphs (called "hammocks") by simply co-scheduling both branches of the conditional interleaved in the same stream, with one branch's code dependent on a predicate register, and the other branch dependent on its complement.

We can exploit predicate registers when they appear in general purpose processors to implement the opportunistic strategy of our abort&retry locking protocol by using them in a non-standard way. Note the important property of the allocation sequence: if the sequence is interrupted, we must not execute any of the following instructions in the sequence, but must instead retry from the beginning of the sequence.

We can achieve this with predicate registers in the following way. We tag all of the atomic instructions in the sequence with a predicate register that is reserved by the interrupt system. We lock the heap by setting this predicate register at the beginning of the allocation sequence, and then proceed to perform the allocation. The interrupt handler clears the predicate register, thus shutting down any remaining portion of the sequence. Finally, the compiler places a branch instruction at the end of the sequence that is predicated on the register's complement; if taken, the branch jumps back to the beginning of the sequence for a retry.

The common case is that the predicate register is set, the allocation occurs, the branch instruction is squashed, and the register is cleared. However, if an interrupt occurs during this sequence, when the thread resumes all following parts of the allocation sequence will be squashed, while the branch instruction will be active, causing a retry.

Using predicate registers in this way has one big advantage: instead of requiring the block of instructions that comprise the allocation sequence to be an indivisible block of instructions, they can now be split up, reordered, hoisted or otherwise manipulated by the scheduler, allowing them to be packed into available slots in the instruction schedule, exploiting static, instruction-level parallelism.

Once again, we are getting performance improvements by tightly integrating the compiler and the operating-system's interfaces.

Note, also, that VLIW architectures have alluring properties for the generation of tight atomic sequences: they allow the code generator to explicitly perform multiple operations in synchrony, by packing them all into the same wide instruction in the code schedule. A VLIW might allow us to simply compose an atomic "fetch&add" operation on the `fp` register in one wide instruction. This may be something to consider for systems that run on media processors, other DSP engines, and the forthcoming Intel IA64 architecture.

8.5 Hybrid interlocking

A safe-points interlocking scheme pays once per basic block—which is good if a basic block performs multiple allocations, and bad if it performs none.[3] In contrast, an abort scheme pays once per allocation—which is good if a basic block performs no allocations, and bad if it performs several. So safe-points is going to work out rather nicely for an allocation-intensive program, such as a theorem prover, while an abort scheme might do well for numeric codes.

On the other hand, why must it be either/or? It's conceivable to have a hybrid scheme, and allow the compiler to switch a thread between abort and commit strategies depending upon the allocations/block ratio of different sections of code. This could lead to an overall solution more efficient than either technique. It's easy for a thread to indicate which recovery regime to use when the heap is locked: the `fp` register has the low three bits available, and the locking protocol is only using one of these bits. The compiler can choose to use the three bits to encode abort-locked, commit-locked, and unlocked, and switch back and forth with a single instruction.

[3]Note that the check can't be removed even if the block does no consing at all, since that might lock out interrupts indefinitely.

9 Future plans: atomic fission

We are now in the process of implementing a brand-new system for use in ML/OS, an experimental SML-based OS kernel that runs on an x86 [ML/OS]. While we cannot report on experience with a working system, the details of our design are sufficiently interesting as to merit brief description.

Our x86 implementation has two major, related changes: we are pushing code annotation as far as we can, and we are splitting apart the allocation and initialisation of memory blocks.

Full code annotation

We are experimenting with shifting as much of the bookkeeping as possible onto detailed PC annotations. For example, the x86 has very few registers, so rather than statically partition the register set into fixed traced and untraced sets, we store this partition information for each instruction. This eases register pressure and also allows pointers in registers to go dead in a timely way.

Splitting the transaction

We have split the allocation and initialisation of memory blocks. Only the heap allocation is performed atomically. The sequence of instructions that stores the initial values into the freshly allocated block is not atomic; it can be split up and the instructions shifted around in the code schedule by the compiler. Again, this is accomplished simply by pushing code annotation further: the code annotations must contain enough information to tell the garbage collector how many words in the current partially-initialised block have been assigned properly and should therefore be traced. To keep matters simple, we only allow one partially-initialised block at a time.

Removing initialisation operations from the atomic transaction reduces the transaction to an extremely short sequence of code:

```
fp := fp+1        ; Lock
r1 := fp-1        ; Allocate
fp := fp+7        ; Commit
```

This is so short, in fact, that there's no longer any need to roll the PC back on interrupt. If we know that an allocation operation always targets register `r1`, we can adopt a simpler, faster alternative. If the interrupt handler resumes a thread that was suspended with the heap locked, it first sets r1 to the *current* value of fp minus one. If the atomic sequence resumes between the first two instructions, this is a redundant operation, but does no harm. On the other hand, if the atomic sequence resumes between the second two instructions, this resets r1's stale contents to its correct value. The third instruction terminates the atomic sequence.

Shortening the atomic sequence so drastically also reduces the odds that the thread will be interrupted while holding the lock.

Note, as a possible variation, that we could shift a small amount of complexity from the interrupt handler to the allocating thread by transforming the atomic allocation sequence into a Herlihy-style "lock-free" or "optimistic concurrency" operation [Herlihy], using a compare&swap instruction, and emitting an explicit branch-to-retry instruction at the end of the sequence:

```
L1:
  r2 := fp            ; Original fp value
  r1 := fp            ; Allocate
  r3 := fp+8          ; New fp value
  cswp(r2,fp,r3)      ; Attempt fp update
  jnz L1              ; Retry if conflict
```

The cswp instruction atomically compares r2 to fp; if they match (that is, if fp hasn't been altered while we were allocating), fp is updated to be the value of r3, and the machine's zero condition code is set to indicate success.[4] This sequence is two instructions longer than the low-bit-locking technique. However, it has a nice feature: the interrupt handler doesn't need to do anything at all to abort the transaction. Instead, the thread itself recognises when it must retry a transaction and does so with a synchronous branch, a much less exotic method of aborting a transaction. As we'll see below, we also have an opportunity to amortise the cost of the extra two instructions over several allocations.

The static allocation arena

The final optimisation we intend to implement is to coalesce all the allocation transactions in a basic block. Let us assume that a given basic block of code allocates three chunks of memory: a five-word record, a three-word record, and a four-word record, for a total of 48 bytes of memory. There will be a single code sequence at the beginning of the basic block to atomically allocate all 48 bytes of heap storage together; this storage is the "static allocation arena," and will be pointed to by a dedicated sap register. *After allocating the basic block's private arena, the rest of the basic block will not touch the* fp *register again.* Further allocation and initialisation operations are simply done by computing offsets from sap and storing initial values into these offsets. These operations are not visible to other threads; they are only visible to the current thread and the garbage collector. Hence there is no synchronisation overhead for the thread, although we must add extra code annotations to map a suspended PC value to information about the static arena, which enables the collector to trace any partially-allocated block in the static arena.

[4] This is typically considered a memory operation; however, the x86 architecture also permits the compare&swap instruction to be applied to registers.

With this optimisation, we only pay for a single synchronisation operation in each basic block that allocates, regardless of how many allocations it performs. Basic blocks that don't allocate at all pay nothing. This gives us the best of the safe-points and abort worlds: per-basic-block synchronisation overhead, and fine-grained interrupt handling.

10 Storage allocation as a kernel operation

We close with some final thoughts on the relationship between storage allocation, compilers, and operating systems. Fundamentally, storage allocation is a kernel service, as is servicing interrupts. The problem we have been facing is that the actual kernel services for locking and storage allocation are too heavyweight for the requirements of advanced programming languages. Consider that in Unix, we could simply bracket our allocation sequences with a pair of sigblock system calls to block and then re-enable the servicing of signals—but this would drive up our overhead by several orders of magnitude.

What we do instead is seek a way to achieve a tighter integration of the compiler and the primitive run-time services. For example, one way to view our heap-locking trick is to note that the frontier pointer is a thread-global bit of state that permanently resides in a register. We have exported OS state out into user-visible register state, where it can be cheaply referenced by user threads and examined by the interrupt system (not the user's interrupt handlers, but the actual interrupt *system*).

With this point of view, inlining cons sequences is essentially just inlining kernel code—again, tightly integrating the OS service and the compiler for enormous efficiency gains. So it comes as no surprise that the issues we've been handling have their roots in general OS service guarantees, such as ITS' PCLSR mechanism. *Vice versa,* it's straightforward to apply the locking tricks we've discussed to abortable sequences, providing general lock-free synchronisation. Massalin has reported on the utility of these operations in OS interfaces [Qua, Herlihy].

Finally, as long as we are tightly integrating these two components, why not just take one more step and provide atomic operations by simply turning off interrupts? We are, after all, just inlining kernel code, and OS kernels commonly turn off interrupts when they need to lock themselves into the processor. Doing so means that we never have to retry, busy-wait or take locks. Interrupt handlers never have to check for anything. The heap is *always* consistent. And we always proceed forwards.

This may seem extreme, but the modern armamentorium of compiler technology provides several ways to export these privileges to user code in a manageable fashion, such as proof-carrying code and safe languages with inlining [PCC]. The case is even stronger when we are dealing with embedded processors in lightweight consumer devices that run a bounded set of trusted code.

As advanced programming languages with automatic storage management and threads become more and more widespread in the world—*e.g.*, in embedded processors that occur in consumer devices, or high-performance media processors and network servers—this tighter integration will become increasingly critical.

11 Acknowledgements

The core ideas in this paper were developed during Shivers' extremely pleasant stay at the University of Hong Kong in 1992. They have benefitted from discussion with Doug Kwan, Francis Lau, John Ellis, Scott Nettles, Greg Morrisett, Alan Bawden, Andrew Appel and John Reppy. John Reppy and Lal George provided support for our implementation in SML/NJ. Clement T. Y. Shin provided detailed Spur references. Anonymous reviewers made several helpful suggestions.

name	blocks	i/blk	imax	smax	h/blk	hmax
boyer	1235	21.819	445	0	32.981	864
life	578	7.962	589	26	6.651	1056
knuth-b	1473	7.761	848	0	5.553	1676
lexgen	2881	7.807	204	13	5.240	340
mlyacc	11351	8.874	3321	88	7.341	4968
vliw	8352	7.704	855	33	5.479	1468
fft	553	8.937	112	0	5.722	204
logic	1044	13.028	735	8	16.897	1440
simple	3652	9.545	218	3	8.639	344
mandelb	315	6.517	112	0	3.746	204
ray	1072	7.631	266	4	5.679	496
barnes-	2272	9.041	214	5	7.678	392
ratio-r	1353	8.395	167	0	5.280	312
count-g	1023	7.699	125	0	5.720	204

Table 1: Static statistics about the benchmarks

	Commit				Abort			
name	noint	int	ints	t/int	noint	int	ints	t/int
boyer	31.393	31.470	1891	0.122	34.060	34.160	2051	0.146
life	22.943	23.057	1382	0.246	25.220	25.500	1529	0.549
knuth-b	20.547	20.193	1220	-0.869	22.603	22.990	1384	0.838
lexgen	22.220	22.517	1374	0.648	23.543	23.330	1421	-0.450
mlyacc	19.683	19.863	1226	0.440	21.667	23.443	1443	3.694
vliw	22.590	23.367	1408	1.655	24.143	25.043	1514	1.783
fft	20.427	20.283	1219	-0.353	23.667	21.673	1299	-4.604
logic	21.983	20.993	1266	-2.346	25.200	24.297	1457	-1.860
simple	24.263	23.993	1446	-0.560	26.290	26.313	1583	0.044
mandelb	20.087	20.157	1208	0.174	21.800	21.937	1314	0.312
ray	23.700	23.483	1425	-0.456	24.820	25.350	1537	1.034
barnes-	24.127	24.290	1464	0.335	26.337	26.547	1593	0.395
ratio-r	219.433	215.997	12145	-0.849	225.930	221.163	12708	-1.125
count-g	48.970	48.637	2917	-0.343	55.633	56.073	3371	0.392
	(sec)	(sec)		(msec)	(sec)	(sec)		(msec)

Table 2: Overhead for servicing null interrupts

name	commit	abort	abort/commit
boyer	31.650	33.793	1.068
life	23.080	25.387	1.100
knuth-b	20.517	22.743	1.109
lexgen	22.327	23.427	1.049
mlyacc	19.900	21.607	1.086
vliw	22.610	24.570	1.087
fft	20.387	23.593	1.157
logic	21.683	25.143	1.160
simple	24.213	26.233	1.083
mandelb	20.043	21.953	1.095
ray	27.927	24.673	0.884
barnes-	24.227	26.240	1.083
ratio-r	216.717	225.003	1.038
count-g	49.710	55.303	1.113

Table 3: Interrupt-free run time

References

[Diwan+] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language. Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI), pages 273–282.

[Feeley] Marc Feeley Polling efficiently on stock hardware. In *Proceedings of the 1993 ACM SIGPLAN Conference on Functional Programming and Computer Architecture*, Copenhagen, Denmark, June 1993, pp. 179–187.

[Herlihy] P. M. Herlihy. Wait-free synchronization. *ACM transactions on programming languages and systems,* 13(1), January 1991.

[JavaGC] James M. Stichnoth, Guei-Yuan Lueh, and Michał Cierniak. Support for garbage collection at every instruction in a Java compiler. Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI), pages 118–127, May, 1999.

[LockFree] Andrew W. Appel. To the best of our knowledge, this very clever technique is due to Andrew Appel and remains unpublished. One of us (Shivers) saw Andrew sketch this approach out on a blackboard in the late eighties, point out the critical trivial-abort property, and state that it had been implemented and subsequently abandoned. But no description of the technique can be found in any of Appel's published works, and when pressed, he becomes evasive, claims poor memory, and denies any involvement with the covert manipulation of autonomous processor-states. We note without further comment his long-standing support by powerful Department of Defense agencies.

[ML/OS] The Flux OSKit: A substrate for kernel and language research. Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin and Olin Shivers. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP-16)*, October 1997, Saint-Malo, France.

[ML-threads] J. Gregory Morrisett and Andrew Tolmach. Procs and Locks: A portable multiprocessing platform for Standard ML of New Jersey. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* San Diego, May 1993.

[Modula-3] Greg Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, 1991, Englewood Cliffs, New Jersey.

[MLRisc] Little has been published on Lal George's MLRisc code-generator compiler back-end. However, on-line documentation can be found at `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/MLRISC/index.html`.

[Orbit] David Kranz. *Orbit: An Optimizing Compiler for Scheme.* Ph.D. dissertation, Yale University, February 1988. Research Report 632, Department of Computer Science. A conference-length version of this dissertation appears in *SIGPLAN 86.*

[ParaCaml] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. Proceedings of POPL '93.

[PCC] George C. Necula, Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, Seattle, Washington, October 1996.

[PCLSR] Alan Bawden. "PCLSRing: Keeping process state modular." Unpublished memo, available via anonymous FTP as `ftp://ftp.ai.mit.edu/pub/alan/pclsr.memo`.

[Qua] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* Ph.D. dissertation, Columbia University, 1992.

[RTC-GC] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* June 1988.

[SML/NJ] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Third International symposium on Programming Language Implementation and Logic Programming,* LNCS 528, pages 1–13, Martin Wirsing, editor, August 1991. Springer-Verlag, New York.

[Spur] Mark Hill, *et al.* Design decisions in Spur. *COMPUTER*, 19(11):8–22, November 1986.

[T] Jonathan A. Rees and Norman I. Adams IV. T: A dialect of Lisp or, Lambda: The ultimate software tool. In *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming,* pages 114–122, August 1982.

[Unimutex] Brian N. Bershad, David D. Redell and John R. Ellis. Fast mutual-exclusion for uniprocessors. In *Proceedings of the Fifth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, October, 1992. ACM Press, SIGARCH Computer Architecture News 20 (Special Issue October 1992), SIGOPS Operating System Review 26 (Special Issue October 1992), and SIGPLAN Notices 27(9).

[Wilson] Paul R. Wilson. Uniprocessor garbage collection techniques. In International Workshop on Memory Management, St. Malo, France, September 1992. (Proceedings published as Springer-Verlag *Lecture Notes in Computer Science*, no. 637). Also available at url `ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps`