AtomCaml: First-Class Atomicity via Rollback

Michael F. Ringenburg * Dan Grossman

Department of Computer Science & Engineering,
University of Washington, Seattle, WA 98195
{miker,djg}@cs.washington.edu

Abstract

We have designed, implemented, and evaluated AtomCaml, an extension to Objective Caml that provides a synchronization primitive for atomic (transactional) execution of code. A first-class primitive function of type `(unit->'a)->'a` evaluates its argument (which may call other functions, even external C functions) as though no other thread has interleaved execution. Our design ensures fair scheduling and obstruction-freedom. Our implementation extends the Objective Caml bytecode compiler and run-time system to support atomicity. A logging-and-rollback approach lets us undo uncompleted atomic blocks upon thread pre-emption, and retry them when the thread is rescheduled. The mostly functional nature of the Caml language and the Objective Caml implementation's commitment to a uniprocessor execution model (i.e., threads are interleaved, not executed simultaneously) allow particularly efficient logging. We have evaluated the efficiency and ease-of-use of AtomCaml by writing libraries and microbenchmarks, writing a small application, and replacing all locking with atomicity in an existing, large multithreaded application. Our experience indicates the performance overhead is negligible, atomic helps avoid synchronization mistakes, and idioms such as condition variables adapt reasonably to the atomic approach.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

General Terms Languages

Keywords Atomicity, Transactions, Concurrent Programming, Objective Caml

1. Introduction

Concurrency has been an important and widely-used programming idiom for decades, even on uniprocessors. Programmers can mask latency by spawning new threads to handle I/O or other inefficient tasks while other threads continue to compute. They can also improve the code structure and response time of interactive applications by spawning new threads to handle user requests while the original thread waits for new requests. Unfortunately, concurrency remains a common source of software errors, resulting in incorrect behavior, crashes, and security vulnerabilities. For example, recent searches for "race condition" on the SecurityFocus [32] and US CERT [37] websites yielded 994 and 537 hits respectively. These difficulties are well-known and inspired much of the related work described in Section 3.

Writing code in which threads communicate via mutable shared memory will never be easy, but there is an increasing belief that locks and condition variables (the most common concurrency primitives in today's high-level languages) make matters worse: As low-level primitives they are difficult to reason about and provide only weak guarantees. For example, locks provide correct synchronization only if every thread accessing shared resources acquires and releases locks at the correct times.

Recently, several researchers have proposed replacing locks with atomicity (see, e.g., [14, 18, 16]). If a block of code is marked as atomic, the language implementation guarantees that the code appears to execute without interleaving of other threads. Instead of "disabling interrupts," the implementation also ensures fair scheduling. Unlike locks, this guarantee is provided regardless of the behavior of the other threads.[1] Figure 1 contains an example of an atomic block and Section 2 reviews some of the technical reasons to prefer atomicity.

For atomicity to become the next-generation concurrency primitive, now is the time to investigate the relevant questions in language design, language implementation, and programming idioms. To do so, we have built AtomCaml, a prototype system that extends the Objective Caml bytecode system with atomicity. At the language level, the atomic construct is a first-class function of type `(unit->'a)->'a` that takes a function and executes it atomically. The block can contain arbitrary code, including buffered output, exceptions, and calls to Caml functions or native C code. AtomCaml is available for download from our website [1].

Our implementation uses logging and rollback to undo an uncompleted atomic's effects if the thread executing it is pre-empted. Like the Objective Caml implementation, we assume a uniprocessor execution model—i.e., that (shared-memory) threads are interleaved, and not run in parallel. Though support for multiprocessing is also important, we believe uniprocessors are an important special-case that allows particularly efficient logging. Just as garbage collectors and operating systems use "specialized" techniques for uniprocessors, atomicity implementations should too.

Our evaluation includes microbenchmarks showing the low overhead of atomic, libraries demonstrating key programming idioms, a small application demonstrating atomic's ease-of-use, and a "port" of the PLANet active-network implementation [22, 21] that removes all uses of locks. In particular, atomic is typically easier to

* Supported in part by an ARCS fellowship sponsored by the Washington Research Foundation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'05 September 26–28, 2005, Tallinn, Estonia.

Copyright © 2005 ACM 1-59593-064-7/05/0009. . . $5.00.

[1] Some systems provide a slightly weaker guarantee, requiring that all shared memory accesses in all threads occur inside atomic sections for the atomicity property to hold.

```
let withdraw amt =
  Mutex.lock acctLock;
  let oldBalance = readBalance () in
  let newBalance = oldBalance - amt in
  setBalance newBalance;
  Mutex.unlock acctLock
```

```
let withdraw amt =
  Thread.atomic (fun () ->
  let oldBalance = readBalance () in
  let newBalance = oldBalance - amt in
  setBalance newBalance
  )
```

Figure 1. A withdraw function written with Objective Caml's Mutex library, and the same function written with `atomic`. The atomic code is just as easy to write, and provides a stronger synchronization guarantee.

use than locks, but idioms using condition variables require some care. In porting PLANet, we fixed concurrency bugs and noticed no significant change in performance.[2]

The rest of this paper is organized as follows: Section 2 discusses some benefits of using `atomic` rather than locks. Section 3 discusses related work including other approaches to providing atomic execution of code. Section 4 describes the design of AtomCaml from the programmer's perspective. We present the functionality we added to Caml, the guarantees it provides, and the interesting language design questions that arose. Section 5 describes the implementation of AtomCaml, including our basic rollback approach, the implementation details, and some interesting implementation choices. Section 6 describes our experience using AtomCaml. Section 7 concludes and presents directions for future work, including support for true shared-memory parallelism.

2. The Case for Atomic

Concurrency errors are still prevalent in modern software in part because locks are difficult to use correctly in complex systems. Accesses to shared resources may be spread across many procedures and files. If any access is not protected by the correct lock or locks, a race condition may occur. The race may result in incorrect behavior by the unprotected code, or it may cause incorrect behavior by protected pieces of code that access the same resource. In the latter case, the source of the error can be difficult to locate. Further compounding the difficulty, a thread may deadlock because another thread (often executing a different procedure) fails to release a lock. These complex interactions force programs to obey subtle program-wide invariants.

On the other hand, atomicity provides a key error-localization advantage: An atomic block will execute as though there was no interleaving, even if code executing in other threads is poorly written. Conversely, a thread executing an atomic block that fails to complete in a timely manner may make little progress, but fair scheduling ensures other threads will not be starved. With locks, a thread holding a lock too long can starve other threads.

Atomicity also has several other advantages over locks:

- As code evolves, it is possible to update any collection of data objects atomically without risking deadlock or changing existing code to obey a new locking discipline.
- Wrapping an abstract datatype's operations in atomic blocks can make the abstraction thread-safe without unneeded locking.
- As Flanagan and Qadeer showed [14], atomicity is often conceptually what programmers achieve with existing synchronization primitives; providing atomicity directly makes programming easier.
- Atomicity and locks peacefully coexist. It is trivial for programmers to implement locks with the `atomic` primitive, but the implementation also need not remove conventional lock implementations.

[2] The performance point is admittedly a tad weak since a real active network would not run as bytecodes in user space.

Atomicity does not eliminate the *granularity* problem: Programmers must still decide whether concurrent operations are fine-grained (potentially increasing performance and nondeterminism) or coarse-grained (which in the limit becomes sequential code). However, atomicity does make it easier to mix fine-grained and coarse-grained operations; locking idioms require complex and error-prone hierarchical locking schemes.

3. Related Work

Atomicity is not a new idea: Operating systems disable interrupts for short sequences. Databases group operations into atomic transactions. Distributed systems employ protocols to commit consistent updates. But as a general-purpose concurrency primitive, atomicity has enjoyed a recent surge of interest in language design and implementation.

Harris et al. have pioneered most of the recent work on language support for atomicity. Their design for Java [18, 17] adds a statement form for atomic execution and uses software transactional memories (STMs) [20, 33] to ensure threads commit consistent views to shared memory. Their atomic blocks have guards that must hold before the atomic block executes. More recently, they developed a system for Haskell [19] in which a transactions monad provides atomicity and the composition (both sequential and alternative) of smaller transactions. Both systems are extremely promising and motivated our work considerably.

Our work complements or extends the work by Harris in several ways. First, the Java system provides a weaker guarantee (atomic blocks appear atomic only to other atomic blocks) and the Haskell system relies on Haskell's purity for the stronger guarantee. Second, in AtomCaml `atomic` is a first-class function of type `(unit->'a)->'a` (STM Haskell's atomic is also first-class, but has return type `IO a` rather than `'a`). Third, we eschew the sophisticated data structures and commit protocols of STMs for simple logging and uniprocessor-based simplifications. Fourth, we develop an API for allowing calls to native code from atomic (instead of raising a run-time exception). Fifth, we report new experience with applications and concurrency idioms.

Other language work has made related contributions. Welc et al.'s transactional monitors [40] provide a weaker guarantee (code guarded by a given monitor appears atomic to threads executing code guarded by the same monitor) in hopes of allowing more parallelism. They also investigate the overhead of techniques (such as write barriers) that we expect will be part of most atomicity implementations. Manson et al. [25] use a logging-and-rollback mechanism similar to ours to add preemptible atomic regions to Real-time Java. Less recently, the Venari project [41] used locks to implement serialized transactions in SML. The ARGUS language [24] provided atomicity for actions that share only objects with special atomic types that had to be accessed via handler calls.

Flanagan, Qadeer, and Freund [14, 13, 11] have taken the complementary approach of checking or inferring that lock-based code is actually atomic. In this sense, atomic is not a term-level primitive but rather a checked type annotation. Programmers must still

```
let add_to_bbuf bbuf item =
 Thread.atomic (fun () ->
  if (is_full_bbuf bbuf) then
    Thread.yield_r bbuf.out_ptr
  else ();
  bbuf.buffer.(!(bbuf.in_ptr))<-item;
  advance bbuf bbuf.in_ptr)
```

```
let remove_from_bbuf bbuf =
 Thread.atomic (fun () ->
  if (is_empty_bbuf bbuf) then
    Thread.yield_r bbuf.in_ptr
  else ();
  let ans = bbuf.buffer.(!(bbuf.out_ptr)) in
  advance bbuf bbuf.out_ptr;
  ans)
```

Figure 2. Bounded buffer insertion and removal functions that use `yield_r` to implement conditional critical regions in AtomCaml. The advance function updates the associated reference (`bbuf.in_ptr` or `bbuf.out_ptr`). This update wakes any threads that were suspended due to a `yield_r` on that reference.

use locks, but a type system can help detect unintended atomicity violations. Their work builds on an underlying data-race detector.

Data-race detectors and deadlock detectors also help find bugs in locking code, but race-freedom is neither necessary nor sufficient for atomicity [14]. The detectors attempt to identify race conditions statically (e.g., [12, 10, 9, 4, 5, 2]) or dynamically (e.g., [31, 8, 7, 38]), so that the programmer can fix them.

We use rollback to implement atomicity whereas some researchers have investigated a variant of exceptions that reverts state in addition to transferring control [26, 34]. Despite similar implementations, such a language construct is fundamentally different in use. Our support for external calls is also slightly more powerful. Logging and rollback has also been used to prevent priority inversion [39], allow safe thread termination [30], and automate software checkpointing [6, 36]. Rollback without needing logging can support specific atomic code sequences on uniprocessors, including heap allocation [35] and an implementation of locking [3].

In the context of functional languages, Concurrent ML [28] and the more recent kill-safe abstractions of Flatt et al [15] are elegant concurrency systems based on asynchronous message passing rather than implicit communication via shared memory. More generally, it is well-understood that the lack of mutation inherent to functional programming reduces the need for synchronization. Our logging approach exploits this fact: the expense of logging is proportional to the number of mutations executed in an atomic block.

Finally, hardware support for software transactions remains ongoing research. Most recently the TCC project [16] uses hardware buffers, consistency protocols, and (in the extreme case) locking the bus to implement atomicity. Transactions have also been used as an optimistic implementation of locking [27].

4. The Design of AtomCaml

In this section, we discuss the design of the AtomCaml language. Section 4.1 describes the `atomic` primitive itself. Section 4.2 discusses the `yield_r` primitive and how it can be used to implement conditional critical regions. Section 4.3 presents the interface that lets programmers specify how external C functions interact with atomic blocks. Finally, Section 4.4 describes design choices regarding the interaction between atomic blocks, exceptions, and input.

As we see below, the only user-visible changes to Objective Caml are `atomic`, `yield_r`, and some additional facilities for interacting with external C code.[3] In addition, our system is fully backwards compatible with Objective Caml—all Objective Caml programs are equivalent AtomCaml programs.

4.1 The `atomic` Primitive

The `atomic` primitive in AtomCaml is a first class function of type `(unit->'a)->'a`. The function takes a thunked block of code as its argument and executes it atomically. The implementation ensures that there will appear to be no interleaving during the block's execution. For example, to execute the following code atomically:

```
let totalWidgets = !blackWidgets + !blueWidgets in
if totalWidgets > 0
then print_string (pickWidget () ^ " available")
else raise NoWidgets
```

we can simply write:

```
atomic (fun () ->
 let totalWidgets= !blackWidgets + !blueWidgets in
 if totalWidgets > 0
 then print_string (pickWidget () ^ " available")
 else raise NoWidgets)
```

As we describe in more detail in Section 5, AtomCaml provides this atomicity guarantee with a rollback and retry-based approach. When the currently executing thread reaches an atomic block, it simply begins executing the block. If the thread is not pre-empted during the execution of the block, then it has executed atomically, since truly parallel execution is not possible in Objective Caml or AtomCaml. On the other hand, if the thread is pre-empted in the middle of the atomic block, we roll the block back (undoing its side effects), and retry the block the next time the thread executes.

The atomic block can contain arbitrary Caml code, including function calls to Caml code or to external C code (see Section 4.3). The atomic block can also raise exceptions that are caught inside or outside the block; an exception that leaves the atomic block causes the atomic block to complete. In addition, Objective Caml's built-in buffered output (`print_int`, `print_string`, etc.) can appear inside atomic blocks, but our current implementation raises an exception if a built-in input function is called inside an atomic block. It would also be possible to allow input inside an atomic block, but as we discuss in Section 4.4, it is unclear if it is a good idea. So far, our applications have not needed it.

Under our semantics, a nested atomic block (for example, an `atomic` inside a function called from an atomic block) is redundant. Our atomic sections appear to execute without interleaving. Because the enclosing atomic block will (appear to) execute without interleaving, all nested blocks will automatically (appear to) execute without interleaving—regardless of whether or not they are labeled as atomic blocks.

4.2 Conditional Critical Regions and `yield_r`

Conditional critical regions (CCR's) are a useful concurrent programming idiom. For example, a function that adds an item to a bounded buffer must first check if the buffer is full, and if so, wait for space to be available. In a lock-based implementation, we would like to avoid acquiring the buffer's lock if the buffer is full. However, we must also ensure that another thread does not fill the buffer between when we check and when we add the new item. CCR's

[3] In the actual implementation, we put the primitives in the `Thread` module.

make this possible by allowing us to delay entry to a critical section until a specified condition (a guard expression) holds.

Similar issues arise when programming with `atomic`. We must check whether the buffer is full inside the same atomic block that adds the item. Otherwise another thread could fill the buffer between when we check and when we insert our new item. However, if it turns out that the buffer is in fact full, we would like to terminate the atomic block immediately and try it again later, when the buffer is no longer full.

Semantically, an explicit call to `yield` (already provided in Caml) suffices: `yield` causes the thread to be suspended. Because the atomic block has not completed, we will undo any effects and restart the block when the thread next runs. After programming this idiom with `yield`, we noticed that the yielding code is often waiting for a mutable reference to have its contents changed and it is useless to rerun the thread until such change occurs.

The `yield_r` primitive (of type `'a ref -> unit`) lets a thread indicate exactly that: `yield_r x` suspends the thread and allows (but does not require) the scheduler to skip the suspended thread whenever the contents of the reference bound to `x` are the same as when `yield_r` was called. Thus our bounded buffer insert function can check if the buffer is full, and if so, simply execute a `yield_r` on the buffer's out pointer. Figure 2 shows a thread-safe bounded-buffer implementation using `atomic` and `yield_r`. These are the same functions we use in the simplified web cache application described in Section 6.3. Appendix A provides another example, using the primitives to implement a variant of condition variables. Note that it is always correct to use `yield` instead of `yield_r`.

4.3 Calling External C Functions

Unlike the prior work described in Section 3, AtomCaml allows programmers to call external C functions from inside atomic blocks. Programmers can specify three types of behavior when they declare an external function:

- If their code can be run without modification in an atomic context (e.g., if it only modifies local state), they can use a declaration of the form:

  ```
  external foo : type_of_foo = "c_foo"
  ```

  The C function `c_foo` will be invoked whenever `foo` is called.
- If the programmer has created a separate version of the external function that is safe to call from inside an atomic section, they can use a declaration of the form:

  ```
  external foo : type_of_foo = "c_foo1 c_foo2"
  ```

  When we call `foo` from a non-atomic context, the first C function (`c_foo1` in this case) will be invoked, and when we call it from an atomic section, the second (`c_foo2`) will be invoked. AtomCaml provides facilities, described below, that make it simple to create atomic-safe versions of many C functions.
- If a function should never be called in an atomic context (e.g., if the programmer has not created an atomic-safe version because they never expect the function to be called in an atomic context), they should use a declaration of the form:

  ```
  external foo : type = "c_foo1 raise_on_atomic"
  ```

  If `foo` is ever called inside an atomic block, a `Sys_error` exception will be raised with an argument indicating the name of the function that was improperly called.

Fundamentally, there is only one mechanism here:

```
external foo : type_of_foo = "c_foo"
```

is the same as

```
external foo : type_of_foo = "c_foo c_foo"
```

and is in fact implemented that way. The third option is the same as the second if `raise_on_atomic` is just a C function provided by the run-time system. It could be if we did not (for convenience) provide the function name in the value that `Sys_error` carries.

When a block rolls back, its side effects must be reversed. So to create atomic versions of C functions, we let programmers specify actions that must occur when the atomic block rolls back. We also let them specify actions that must occur when the atomic block successfully completes. The latter facility lets unreversible actions be delayed until it is known that reversal is unnecessary. AtomCaml provides two functions to specify these actions:

- The `caml_register_rollback_action(void(*reg_func)(void*), void* reg_env)` function causes the function `reg_func` to be called with argument `reg_env` when the currently executing atomic block rolls back.
- The `caml_register_commit_action(void (*reg_func)(void*), void* reg_env)` function causes the function `reg_func` to be called with argument `reg_env` when the currently executing atomic block completes.

The virtual machine thread scheduler will not pre-empt threads in the middle of an external C function, so we do not need to worry about being interrupted between executing an action and registering the corresponding rollback action. Figure 3 shows how to use these functions to create atomic-safe versions of C functions that increment and delete heap-allocated counters. The rollback action undoes any increments and the commit action completes any delete (which was delayed during the atomic block).

This interface suffices for implementing atomic-safe buffered output. The first output can register an action that reverts the buffer to its original state if the atomic block rolls back. In addition, the atomic-safe version of the flush function can register the flush as an action to be taken when the atomic block successfully completes.

4.4 Language Design Choices

Several interesting language-design questions arose during our work. We now consider the two most interesting: Should we allow input inside atomic blocks? What do exceptions thrown from atomic blocks mean?

Given our implementation's rollback-and-retry mechanism (see Sections 4.1 and 5), it is not difficult to allow input within atomic blocks, but for semantic and efficiency reasons (and the lack of a compelling need) we have made the policy decision not to. To allow input, each input statement in an atomic block would simply read the next item from the buffer. If we rolled back, the items read would be put back into the buffer. Implementing this approach, however, would force all input functions (even the non-atomic ones) to see if input is already available in a buffer because of another thread's rollback. Furthermore, we cannot statically bound the size of the buffer, because we cannot predict how much data may need to be put back into it due to a rollback. (For output, the atomic functions' buffers cannot be bounded, because we cannot flush them until we complete the block. However, the non-atomic functions' buffers *can* be bounded, because we can flush them whenever. This asymmetry between input and output surprised us.)

There are also semantic problems with allowing input after output in an atomic block: The input cannot depend on the output because we delay the output until the atomic block completes. For example, the output could prompt a user for information and the input could be their response. Our atomic primitive is for shared-memory concurrency; it is unwise (and impossible, given our current choice to raise an exception if an input function is called inside an atomic block) to perform external communication atomically.

Turning to exceptions, the interesting case is when an exception thrown from an atomic block is not caught in the atomic block.

```
struct Counter {
  int val;         // current value
  int did_registration;  // boolean
  int old_val;  // pre-atomic value
  int pending_delete;    // boolean
};

typedef struct Counter * counter;

void commit_ctr(void *v) {
  counter c = (counter) v;
  if(c->pending_delete)
    delete_ctr(Val_int((int)c));
  else
    c->did_registration = 0;
}

value inc_ctr_atomic(value v) {
  counter c = (counter) Int_val(v);
  do_registration(c);
  return inc_ctr(v);
}
```

```
void do_registration(counter c) {
  if(c->did_registration) return;
  c->did_registration = 1;
  c->old_val = c->val;
  caml_register_commit_action(commit_ctr,c);
  caml_register_rollback_action(rollback_ctr,c);
}

void rollback_ctr(void *v) {
  counter c = (counter) v;
  c->val = c->old_val;
  c->did_registration = 0;
  c->pending_delete = 0;
}

value delete_ctr_atomic(value v) {
  counter c = (counter) Int_val(v);
  do_registration(c);
  c->pending_delete = 1;
  return Val_unit;
}
```

Figure 3. An atomic-safe version of C code that increments and deletes heap-allocated counters. The functions `caml_register_rollback_action` and `caml_register_commit_action` register functions that are called if the currently executing atomic block rolls back or completes, respectively. The code undoes increments on rollbacks and delays deletes until an atomic block completes. `new_ctr`, `inc_ctr`, and `delete_ctr` are straightforward and not shown.

In our view, an exception is just a nonlocal control transfer and when control transfers outside the atomic block, the atomic block commits successfully. This policy is semantically simple and aligns with our view that atomic is nothing more and nothing less than a concurrency primitive.

Another possibility is to roll back an atomic block when an exception occurs, which is tempting because exceptions often indicate unexpected conditions. This is a fundamentally different exception semantics: Rather than just transfer control, an exception raise would also revert to a previous state. While such an operator has its advocates [26, 34], we consider it an orthogonal language feature that happens to enjoy a very similar implementation. From the perspective of atomicity, `yield` and `yield_r` allow a thread to abort an atomic block and retry it later. Quite differently, this variant of exception would abort an atomic block and not retry it (continuing as though the atomic block were empty). This feature has its own pitfalls, however. For example in AtomCaml this code will never fail, because the write to `x` will not be undone—even if `f` raises an exception:

```
let x = ref 0
atomic (fun () ->
   x := 1;
   f() (* f may raise an exception *))
if !x = 0 then failwith "huh" else ...
```

If exceptions had a rollback-and-do-not-retry semantics, an exception thrown in `f` would cancel the write to `x`. The code would then fail because `x` would be 0 when we reached the conditional.

5. The Implementation of AtomCaml

We now discuss our strategy for implementing the design described in the previous section. As described earlier, we implemented AtomCaml as an extension to bytecode-compiled Objective Caml. Section 5.1 describes our basic approach to guaranteeing atomicity. Section 5.2 discusses why Objective Caml and other mostly functional languages are a good target for this approach. Section 5.3 presents some details of our implementation. Finally, Section 5.4 discusses key implementation choices.

5.1 The Approach—Atomicity via Logging and Rollback

The ease and correctness of `atomic` is well-known; it is why operating system code disables interrupts for short code sequences. The problem with making atomic a language construct in a safe language is that we do not trust code to re-enable interrupts, nor do we want draconian restrictions like disallowing function calls in atomic blocks. We show how to provide particularly efficient support for atomicity on systems without true parallelism (e.g., uniprocessor systems and systems like Objective Caml where the run-time system requires that at most one thread executes at any given time), without disabling interrupts or restricting code.

We designed our approach based on the belief that most atomic blocks will be short, and that most code will execute outside of atomic blocks. The applications described in Section 6 confirm these observations. Thus we have designed a system where short atomic blocks execute with low overhead and where non-atomic code is not slowed down compared to systems without atomicity.

The key idea is this: We know exactly one thread is executing at any time. When the currently executing thread reaches an atomic block, it optimistically begins executing the block. If the thread completes the block without being pre-empted by the scheduler, no further action is required—the block executed atomically. Because atomic blocks are typically short, this case is the common one. If the scheduler does pre-empt a thread during an atomic block, the thread rolls back to the beginning of the block and removes any evidence that the block was partially executed. The next time the thread executes, it will restart at the beginning of the block.

Given this general approach, several questions remain:

- In a language with side effects, how can we ensure that the program behaves as if a rolled back block never executed? In particular, how do we handle writes to mutable data and output?
- How do we ensure that an atomic block eventually completes successfully, without starving other threads?

- How do we handle functions that may be used inside both atomic and non-atomic contexts? Their behavior must be different in these two cases, but we do not want to slow down non-atomic code.
- How do we ensure that our approach has low overhead?

The rest of this section answers these questions in turn.

To enable rollbacks, side effects (memory writes, output, message sends, etc.) in atomic blocks must be reversible. The essential uniprocessor "optimization" is that our system has no need to log reads to memory. In atomic blocks, the system tracks updates to mutable locations with a log that records the location and the previous value. If rollback becomes necessary, it simply reverts every location in the log to its original value. The only exceptions to this rule are variables and memory that are local to the atomic block—they do not need to be logged or reversed because the thread will recreate them when it retries the block. In a garbage collected language, the memory occupied by these local variables will be reclaimed because there will be no references to them after the rollback. We handle output and message sends by buffering, and by not allowing the buffers to flush in the middle of an atomic block. Rollbacks can then remove the writes and sends from the appropriate buffers.[4] Input during atomic blocks is an interesting policy question, discussed in Section 4.4.

We also must ensure that an atomic block will eventually be able to complete (provided, of course, that it does not enter an infinite loop). To achieve this, the scheduler allocates extra time to a thread's next execution if it fails to complete an atomic block after two attempts. (Note that the block will have been given a full time slice on the second attempt.) If the block still fails to complete, we can allocate even more time on subsequent executions. This extra time may not be necessary—the atomic block may complete faster because of work done by other threads. However, by allocating more time, the scheduler ensures that inherently long atomic blocks will eventually be able to complete. To ensure fair scheduling (i.e., that we do not starve other threads), the scheduler skips the thread for a number of rounds proportional to the extra time allocated. The scheduler can also enforce an upper bound on the allocated time per thread execution, in order to prevent unresponsiveness. These policy issues may depend on an application and should be tunable just like garbage-collection parameters.

To allow function calls inside atomic blocks, the system must also log writes in functions called from inside an atomic block. We could test at function entry whether the current thread is in an atomic context, but this test would slow down non-atomic code. Instead, the compiler generates two versions of each function: an atomic version that logs side effects and buffers output, and a non-atomic version that executes normally. We can determine statically which version to call: when we call the argument to an `atomic`, or when we make a call inside the atomic version of a function, we call the atomic version. Otherwise, we call the non-atomic version. In a functional language, this requires adding a second function pointer to each closure,[5] and creating new versions of the `apply` bytecodes that follow the new code pointers. We can also reduce the code size somewhat by statically identifying functions that are called only in atomic contexts, or only in non-atomic contexts, and generating only the appropriate versions. We can also avoid duplicating purely functional code, because there will be no writes to log.

Because non-atomic code is essentially unchanged and rollbacks are rare, the primary source of overhead is logging writes. Logging occurs only inside atomic blocks (which are typically only a small fraction of the code), and writes must already be tracked by the (generational) garbage collector. Thus this overhead has little effect on the overall running time of the applications discussed in Section 6. Even if an atomic block has trouble completing (thus suffering multiple rollbacks), only the thread executing it slows down. This is in contrast to locking systems, where other threads wait for locks held by a thread executing a problematic critical section.

Should it prove necessary, we could investigate dynamic and static approaches to reduce rollback frequency. For example, a thread can yield rather than start an atomic block near the end of its time slice (when it is less likely the block will complete). The compiler can also use static analysis to shrink atomic blocks with right- and left-movers at, respectively, the beginning and end of the block (see [13, 14, 23] for information about movers).

5.2 Why Objective Caml?

We chose Objective Caml as a starting point for our prototype for several reasons. First, as a mostly functional language, writes to mutable data structures in Caml are relatively rare. Since these writes are our primary source of overhead,[6] we expect that logs will remain quite small and our experience confirms it. Second, Objective Caml already lacks support for true parallelism, so we exploit the same simplifications that the existing run-time system already does. Third, we can make atomicity a fully first-class construct, i.e., a function of type `(unit->'a)->'a`. Without higher-order functions this is clearly impossible. Moreover, in a purely functional language like Haskell, atomic is useful only as a monad [19].

Making `atomic` a function helps enormously in localizing the changes necessary to the implementation. We simply add the declaration

```
external atomic: (unit->'a)->'a = "atomic"
```

to the `Thread` module, and the front end correctly parses and typechecks uses of `atomic`. In short, we changed *nothing* in the front end.

Making `atomic` a function also simplifies nested atomic blocks. As described in Section 4.1, nested atomics are redundant; such a call should just apply its argument because there is already a log set up. However, a single `atomic` construct may be used in both nested and non-nested contexts. Treating `atomic` as a function solves this problem. We already have two versions of every function, and we already determine which one to call based on the context. The outermost `atomic` will always be called from a non-atomic context (otherwise it would not be the outermost), and nested `atomics` will always be called from atomic contexts (otherwise they would not be nested). Thus the non-atomic version of `atomic` does the setup for logging and rollback, and calls the atomic version of the passed function, and the atomic version is simply:[7]

```
let atomic th = th ()
```

Since we are already in an atomic context, we automatically call the atomic version of `th`.

We chose to change the bytecode compiler rather than the native compiler for three reasons: (1) simplicity (it is a smaller system), (2) portability, and (3) a user-level thread scheduler (so we did not have to deal with the kernel). The disadvantage is that bytecode typically runs slower than native code, making our performance results less reliable. Our approach should work fine with native code and a system thread scheduler provided the latter provides

[4] Removing writes and sends from buffers is really just a special case of reversing memory writes.

[5] We can save this extra per-closure space, but at the expense of increasing the overhead of atomic function calls. See Section 5.4.

[6] Recall that initialization writes do not need to be logged, because we will lose all references to the data after a rollback.

[7] As we discuss in Section 5.3.3, `atomic` is actually a C function built in to the virtual machine. However, the behavior of the atomic version of `atomic` is identical to the Caml code given here.

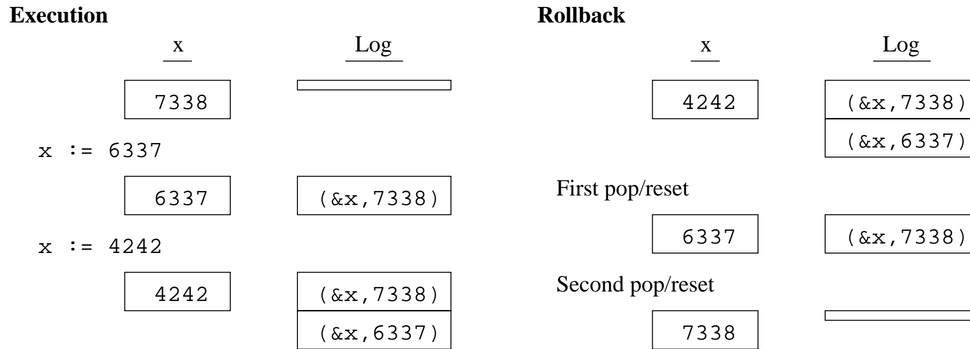


Figure 4. This figure illustrates the effect of rolling back an atomic block that modifies a mutable variable `x` twice. The reference `x` initially holds the value `7338`. The first update pushes an entry onto the stack containing the address of `x` and it's former value (`7338`). The second update pushes an entry containing `x`'s address and its value after the first update (`6337`). On rollback, we first pop the stack and reset `x` to `6337`. We then pop the next element and reset `x` back to its original value: `7338`.

a hook for executing code when a (lightweight, i.e., intraprocess) thread is pre-empted.

5.3 Implementation Details

We implemented AtomCaml by modifying the Objective Caml bytecode compiler, the run-time system (particularly the thread scheduler and the garbage collector), and the `Thread` library. As mentioned earlier, we did not touch the front end of the compiler.

This section describes (at a high level) the modifications we made to implement AtomCaml. None of these modifications rely on any unusual properties of Caml or the Objective Caml virtual machine. Thus similar extensions should be possible in other mostly functional languages.

5.3.1 Logging and Rollback

As Section 5.1 describes, AtomCaml logs all non-initialization writes occurring inside atomic blocks and reverses them if the block is rolled back. To do so, the AtomCaml compiler uses alternate versions of bytecodes and primitives that might modify existing data. These alternate forms cause the virtual machine to log the modification. This approach is more efficient than setting a flag when we enter an atomic block and checking the flag on every write, because the check would slow down non-atomic code. The log is a stack of modified addresses and their previous values. Thus when we roll back, the first writes are the last reversed. As Figure 4 illustrates, this ensures all locations revert to the value they held before the atomic block. If the log becomes large, we switch to a hashing scheme to avoid logging multiple writes to the same address (see Section 5.4).

We must also consider how logging and rollback interact with garbage collection: If we lose the last reference to an object during an atomic block, we still must not garbage collect it. Otherwise, after a rollback the program could have a reference to an object that no longer exists. We must also ensure that the garbage collector updates the log if it moves any logged data structures. Thus the log must be reachable to the garbage collector.

5.3.2 Code Duplication

AtomCaml creates two copies of each function: an atomic version that logs writes and buffers output, and a non-atomic version that does not. Thus our compiled code size is roughly twice that of an equivalent Objective Caml program. The compiler indicates which function should be called by inserting different `apply` bytecodes in atomic and non-atomic contexts. Since `atomic` is simply a function that calls the atomic version of its argument, the atomic contexts consist of the `atomic` function itself, and the atomic versions of all other functions. Thus, to compile AtomCaml code, the bytecode compiler merely needs to compile each function twice. The first compile generates the non-atomic version of the function, and proceeds exactly as a normal Objective Caml compilation would. The second compile generates the atomic version, and is identical to the first except that instructions that call functions or modify mutable data are replaced with their alternate, atomic forms. Calls inside atomic blocks will then automatically call the atomic version.

The virtual machine keeps track of these two function versions by adding an extra code pointer to every closure. This is depicted in Figure 5b. The normal `apply` bytecodes follow the original code pointer, and the atomic `apply` bytecodes follow the new code pointer. In Section 5.4, we discuss an alternate representation that eliminates this extra per-closure space at the expense of slower function calls inside atomic blocks.

5.3.3 The `atomic` Function

The `atomic` function lets the programmer pass in code that should be evaluated atomically. We implemented `atomic` as a C function that is built-in to the virtual machine runtime. It must be a part of the runtime because it is the one place where we transition from a non-atomic context to an atomic context. The compiler need not (and in general does not) know if a function-call target is `atomic`.

The version of `atomic` called from non-atomic contexts must perform the setup work necessary to enter an atomic section, call the argument function, and rollback the block if it failed to complete. The function first sets up the modified-memory log. It then calls a new version of the Objective Caml callback function that follows the atomic code pointer of the passed function. If the thread scheduler interrupts an atomic block,[8] it throws a special rollback exception. If the callback returns this exception, `atomic` rolls back the modifications in the log and executes any registered rollback actions (see Section 4.3). Otherwise, we empty the log and execute any registered commit actions. The version of `atomic` called from atomic contexts simply calls the passed function. As described in Section 4.1, nested `atomic`s are redundant, so this is sufficient.

5.4 Design Choices

In this section, we consider key choices we made in the AtomCaml implementation. We first discuss the representation of function

[8] Objective Caml experts may recall that the virtual machine normally prevents pre-emption during callbacks. AtomCaml allows pre-emption during callbacks, but only if all active callbacks are from the `atomic` function.

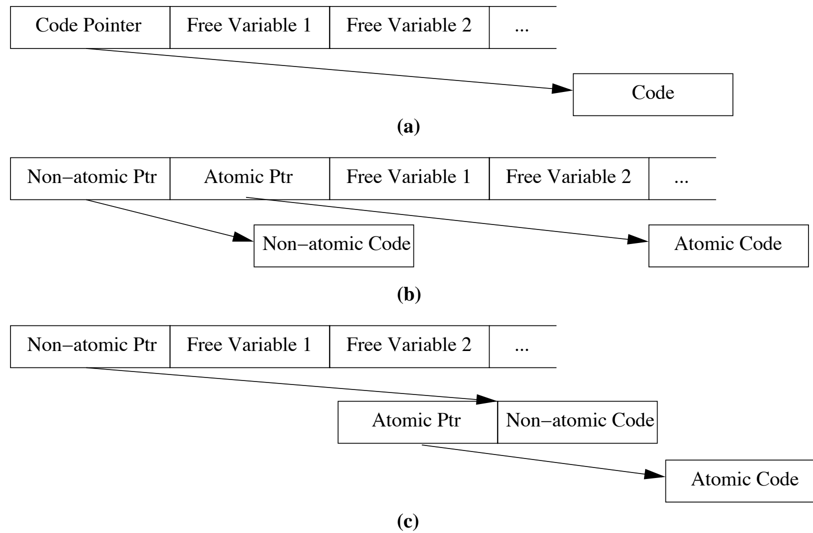


Figure 5. **a)** Objective Caml function closures. **b)** AtomCaml closures with two code pointers. **c)** AtomCaml closures with a single code pointer. The atomic code pointer is placed at a fixed offset from the non-atomic code.

closures and an alternate representation (Section 5.4.1). We then describe some optimizations to the logging mechanism that we chose to implement, and the associated tradeoffs (Section 5.4.2).

5.4.1 Closure Representations

Function closures consist of a code pointer and an environment containing the function's free variables (see Figure 5a). The most straightforward extension adds a second code pointer, as depicted in Figure 5b. This representation has the advantage of not adding any extra levels of indirection for function calls. However, the closures become larger, so closure creation is more expensive, even in non-atomic code.

Alternately, we could place the atomic-code pointer at a fixed offset from the start of the non-atomic code. We would then need only one code pointer in the function closure, as depicted in Figure 5c. This representation avoids enlarging the closures, thus closure creation is not slowed down. However, atomic functions calls will now have to follow two code pointers. Thus non-atomic code is sped up (due to cheaper closure creation), at the expense of slowing down function calls in atomic blocks.

As Section 6.5 shows, our test applications that use `atomic` run slightly faster (or at the same speed) with the first representation; thus we chose it for our implementation.

Objective Caml also uses special "flattened" representations for closures containing mutually recursive functions. Both our approaches extend naturally to such closures.

5.4.2 Optimizing Logging and Rollback

Generally, we want to reduce the work done for each atomic write, even if it means more expensive rollbacks, because we expect rollbacks to be rare. However, there are some cases where doing a little extra work when we log a write can save lots of time on a rollback. In particular, if we update the same location many times, we can save time on rollback by only logging the first update. However, checking for repeats has overhead; thus it only makes sense for atomic blocks that have an above-average chance of being rolled back, and that perform enough repeated writes that they stand to gain by eliminating duplicates. In our current implementation, we dynamically track the total number of writes to mutable data in a block to approximate both quantities. A block with a large number of writes has a higher chance of repeated writes, and will likely take longer to complete and thus have a higher chance of being rolled back. Thus, once the number of writes reaches our preset threshold (currently 50), we begin hashing newly logged addresses to check for duplicates. We also begin incrementally hashing the previously logged addresses to check for any duplicates that have already been inserted. Another option we plan to try in the future would attempt to identify repeated writes statically. For instance, a static analysis could determine that the write to `big_number` in the following loop needs to be logged only once:

```
atomic (fun () ->
  while (!big_number > 0) do
    big_number := !big_number - 1; ... done)
```

6. Experience

To understand the convenience and efficiency of AtomCaml, we have written or modified multithreaded libraries and applications. Section 6.1 describes common idioms we encountered in existing code and external C libraries and how these idioms appear when using atomic. Section 6.2 investigates the performance implications of our implementation on contrived microbenchmarks and non-atomic applications. Sections 6.3 and 6.4 describe more complete case studies involving one new and one existing application. Section 6.5 compares closure representations.

6.1 Common Idioms with Atomic

Usually, programming with atomic is much easier than programming with locks. With locks, one typically uses patterns like the following (which is a much-used utility function copied verbatim from PLANet [22, 21]):

```
let critical m thunk =
  try
    Mutex.lock m;
    let result = thunk() in
    Mutex.unlock m;
    result
  with e -> (Mutex.unlock m; raise e)
```

It is often but not always meaning-preserving to replace this function with:

```
let critical m thunk = atomic thunk
```

In many cases, a change like this one suffices. For example, the Objective Caml standard library provides an implementation of the Concurrent ML primitives [28]. The implementation keeps private data structures consistent with a "master lock" and atomic works just as well. It took only a few minutes to change the library, which we had never seen before. Similarly, code that wraps libraries such as hashtable implementations with a lock to implement a monitor-style abstraction is ideally suited for atomic. In one place in PLANet described below, making the simple change to atomic made a library more useful by avoiding a potential deadlock.

However, occasionally using atomic where current practice would use locks is incorrect or a bad idea. We consider three such scenarios, two of which we encountered during our case studies.

6.1.1 Condition Variables

Threads often communicate via *condition variables*, which have an associated lock and support the *wait*, *signal*, and *broadcast* operations[9] (see the `Condition` library in Objective Caml). A thread calling `wait` should hold the associated lock. `wait` then releases the lock and suspends the thread. Upon resumption, `wait` reacquires the lock before returning. `signal` resumes one thread waiting on the condition variable and `broadcast` resumes all of them. To our knowledge, prior work on atomic has not considered using this important idiom.

Code using `wait` generally has this form, where *lk* is a lock and *cv* is a condition variable:

```
critical lk (fun () ->
   e1;
   let rec loop () =
        if e2 then e5
        else (e3; wait cv lk; e4; loop ()) in
   loop ())
```

It is crucial that the call to `wait` suspends the thread so that another thread may modify shared state such that e_2 becomes true. But rolling back (as the atomic implementation of `critical` would do upon suspension) is incorrect if other threads need the effects of e_1, e_2, e_3, or e_4 to make progress. (If and only if these four expressions are pure does a conditional critical region [18] or `yield_r` suffice.)

Although there are often simpler solutions (it is rare that this pattern is used in its full generality), a general solution is *almost* the following:[10]

```
let f() = if e2 then Some e5 else (e3; None) in
let rec loop x =
   match x with
     None -> (wait' cv;
              loop (atomic (fun () -> e4; f())))
  |  Some y -> y in
loop (atomic (fun () -> e1; f()))
```

As needed, this code evaluates the correct expressions atomically and it does not call `wait'` (which we suppose takes a condition variable but not a lock) from within `atomic`. Unfortunately it has a race condition: Between the evaluation of e_2 and `wait'` *cv*, another thread could make e_2 true and signal `cv`; the waiting thread will never see a signal that precedes the wait. In the lock-based code, this race does not exist because the waiting thread holds the condition variable's lock until it suspends and a signaling thread must hold the lock.

To solve the problem, the waiting thread must start *listening* for a signal inside `atomic` (with `listen`) but *suspend* itself (with `wait`) outside atomic. The following interface and revised code suffices:

```
type condvar
type channel
val create : unit -> condvar

(*signal a channel that hasn't yet been signaled*)
val signal : condvar -> unit
val broadcast : condvar -> unit
val listen : condvar -> channel

(*suspends unless/until channel is signaled*)
val wait : channel -> unit

type 'a attempt = Wait of channel | Go of 'a


let f() =
  if e2 then Go e5 else (e3; Wait (listen cv) in
let rec loop x =
   match x with
     Wait ch -> (wait ch;
                 loop (atomic (fun () -> e4; f())))
  |  Go y -> y in
loop (atomic (fun () -> e1; f()))
```

In an atomic section, a thread can start listening for a signal, so a signal cannot be missed. The call to `listen` returns a *channel* on which signals occur; the `wait` operation suspends unless the channel has been signaled. If another thread "quickly" signals the channel, then the `wait` operation will simply not suspend.

Our implementation of condition variables is about 20 lines of AtomCaml; Appendix A contains its entirety. Note our use of `atomic` above leads to nested `atomic` evaluations, but these pose no problem.

6.1.2 External Calls (e.g., I/O)

If Caml code calls C code while holding a lock, then before replacing the lock acquisition/release with `atomic`, we must do one of the following to ensure that we will be able to safely roll back the block: (1) modify the Caml code, (2) manually verify the C code (at least as it is being called) is safe for atomic, (3) write an atomic version of the C code using the API described in Section 4.3. Which approach is easiest depends on the situation.

Modifying the Caml code is often not difficult. For example, this lock-based code closes or writes to a shared output channel held in a shared variable `f` that is guarded by `lk`:

```
fun close () =
  critical lk (fun () -> match !f with
    None -> ()
  | Some oc -> close_out oc; f := None)
fun output () =
  critical lk (fun () -> match !f with
    None -> () | Some oc -> output_string "ICFP")
```

We have modified the C code implementing `close_out` and `output_string` to have no effect until an atomic block completes. But suppose it was too difficult to provide this functionality for `close_out`, so we had it raise an exception instead. Then the programmer can still just write:

[9] These operations are sometimes called *wait*, *notify*, and *notifyAll*.

[10] Note it is also easy to abstract the pattern with a higher-order function taking thunks that evaluate e_1, e_2, e_3, e_4, and e_5.

```
fun close () =
  let th = atomic (fun () -> match !f with
    None -> (fun () -> ())
  | Some oc -> (f:=None; (fun () -> close_out oc))
  in th()
fun output () =
  atomic (fun () -> match !f with
    None -> () | Some oc -> output_string "ICFP")
```

Other times, we may know an operation is pure so we can move it outside of an atomic block. For example, the Objective Caml library uses C code for parsing, but parsing a constant (and well-formed) string is nonetheless a pure operation at the application level.

Determining if the C code is actually safe for atomic is usually easy: Getter functions (e.g., `pos_out`, which returns an output-channel's current position) are typically safe and other functions typically are not. In the latter case, it may be possible to delay effects until an atomic block commits.

6.1.3 Long Critical Sections

We have yet to find a situation where it seemed appropriate to hold a lock while performing a long computation but inappropriate to perform the computation atomically. This could happen if the computation was long enough to force the atomic block to roll back repeatedly. In this situation, it would be better to use locks or an idiom simulating them. We can "have our cake and eat it too" because our implementation of `atomic` is compatible with conventional lock implementations such as Objective Caml's `Mutex` library. Moreover, as a simple exercise we have written a library on top of `atomic` that provides simple locks, locks that check the releasing thread is the acquiring thread, reentrant locks, and readers/writer locks. The simple lock implementation is trivial:

```
type simple_lock = bool ref
let acquire lk = atomic (fun () ->
  if !lk then yield_r lk else lk := true)
let release lk = atomic (fun () -> lk := false)
```

6.2 Simple Benchmarks

We created several small tests to evaluate the performance of various aspects of AtomCaml. The tests were run on a 500MB, 2.26 GHz Pentium 4. Sections 6.3 and 6.4 present the overhead of two real applications.

As described in Section 5, our primary source of overhead in non-atomic code is closure creation. We measured this cost with a loop that creates 100,000,000 functions. AtomCaml required 2.945 sec. to execute this test and Objective Caml required 2.457 (a 19.9% overhead).[11] To see how this carries over to real applications, we compiled the AtomCaml compiler (which contains no atomic blocks) with both AtomCaml and Objective Caml, and compared the resulting compilers on two large Caml source files from the AtomCaml distribution: parser.ml and ctype.ml. The AtomCaml-compiled compiler took 1.6% longer to compile parser.ml (0.594 vs. 0.584 sec.) and 2.7% longer to compile ctype.ml (1.015 vs. 0.988 sec.). We also compiled an interpreter for the language described in [29] with both AtomCaml and Objective Caml. The AtomCaml version took 1.3% longer to run our largest test (3.483 vs. 3.440 sec.). Thus for "real" non-atomic code, AtomCaml adds about a 2% overhead.[12] These results are summarized in Figure 6a.

We also created three microbenchmarks to measure the cost of logging and rollback. Each has a loop that iterates 100,000 times. The first microbenchmark's loop body executes a series of writes (decrements of an int ref) followed by an empty atomic block. The second has the same writes inside the atomic block instead of outside; thus the difference with the first is the cost of logging. The third is like the second except the atomic block rolls back instead of completing;[13] thus the difference with the second is the cost of rollback (slightly underestimated since the enclosing loop body never completes). We ran each benchmark with 0, 10, 50, and 100 writes of the same top-level reference. Figure 6b summarizes the results; the running time for the first benchmark is "execution", the second "execution plus logging", and the third "execution plus logging plus rollback". Rollback was actually faster with 100 writes than with 50 because our duplicate-elimination logging optimization removed redundant writes from the log. Appendix B contains the data for this graph.

Finally, we compared the cost of executing an atomic block and executing the same code surrounded by a lock acquire and release. Our blocks consisted of a loop (implemented as a recursive function) that executes a variable number of writes. Our results are summarized in Figure 6c. With no writes, the atomic block took 2.15 times longer (0.118 sec. vs. 0.055 sec. for 100,000 executions). With 5 writes, the atomic block took 2.08 times longer (0.156 vs. 0.075 sec.), and with 10 writes, it took 1.82 times longer (0.182 vs. 0.100 sec.). The longer atomic blocks experienced less relative overhead (despite having to do more write-logging) because there are fixed costs associated with atomic entry and exit. For instance, we must allocate the log, perform a callback of the passed in function, and then check if a rollback is necessary. We evaluated the source of this overhead with another set of benchmarks that add callbacks to the lock acquire/release benchmarks described above. We determined that the callback accounts for 23.3% of the overhead in the 0 write case, 19.8% in the 5 write case, and 17.7% in the 10 write case.

6.3 Small Application: Web Cache

We also wrote a simple web cache application in AtomCaml. The application is initially given a page to cache. It searches that page for links and embedded objects (e.g., images), and caches them as well. If any of these objects are themselves HTML pages, we search them in turn. For simplicity, and for more consistent performance results, our application only works with locally stored web pages.

We implemented our web cache with a simple thread-safe bounded buffer library we wrote in AtomCaml. We create a bounded buffer for each HTML page and a producer thread to scan the page and insert any linked or embedded objects into the buffer. We also create a consumer thread to remove the objects from the buffer and cache them. If the object is an HTML page, the consumer creates a new bounded buffer and the two threads for it. Thus two threads share each bounded buffer, and every consumer thread shares the cache. Figure 7 depicts this architecture.

Our bounded buffer insert and remove functions are shown in Figure 2. Using `atomic` in these functions ensures no thread ever sees a bounded buffer with a partially-inserted or partially-removed object. Using `yield_r` lets us rollback the atomic block if we attempt to insert into a full (or remove from an empty) buffer, and to put the thread to sleep until the buffer is no longer full (or empty).

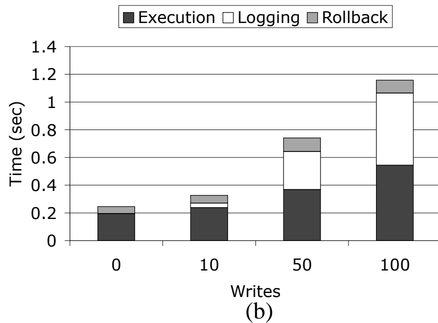
[11] Neither version experienced any major garbage collections. The number of minor collections was directly proportional to the size of the closures: 9155 for AtomCaml, and 6103 for Objective Caml.

[12] There are situations where closure-creation overhead is more significant. In particular, a large application with many top-level functions creates many closures when the application starts. If the application does little additional work, the start-up cost can dominate. For situations like these, it may be better to use the alternative closure representation described in Section 5.4.

[13] We use a call to `yield` at the end of the block to roll back. For a fair comparison, we included a call to yield immediately after the atomic block in the other two microbenchmarks.

	AtomCaml			Objective Caml			
Test	Min	Max	**Avg**	Min	Max	**Avg**	**Overhead**
Compile `ctype.ml`	1.007 sec	1.021 sec	**1.015 sec**	0.975 sec	0.996 sec	**0.988 sec**	**2.7%**
Compile `parser.ml`	0.591 sec	0.597 sec	**0.594 sec**	0.578 sec	0.592 sec	**0.584 sec**	**1.6%**
CDS Interpreter	3.466 sec	3.502 sec	**3.483 sec**	3.432 sec	3.456 sec	**3.440 sec**	**1.3%**
Web Cache	7.582 sec	7.758 sec	**7.689 sec**	7.563 sec	7.787 sec	**7.630 sec**	**0.8%**
PLANet ping	1.062 ms	1.709 ms	**1.198 ms**	1.024 ms	1.398 ms	**1.150 ms**	**4.2%**
“” send (in sec/MB)	0.0657	0.0683	**0.0667**	0.0710	0.0754	**0.0736**	**(–9.4%)**
“” receive (sec/MB)	0.0868	0.0925	**0.0898**	0.0820	0.0857	**0.0839**	**7.0%**

(a)



(b)

Block	Lock acquire and release	Atomic block	Overhead
0 Writes	0.055 sec	0.118 sec	115%
5 Writes	0.075 sec	0.156 sec	108%
10 Writes	0.100 sec	0.182 sec	82%

(c)

Figure 6. Results of tests evaluating AtomCaml's performance: Table (a) compares the performance of programs compiled with Objective Caml and with AtomCaml. Table (b) measures the overhead of logging and rollback for atomic blocks with 0, 10, 50, and 100 writes. Table (c) compares the execution time for a short atomic block with the execution time for the same block plus a lock acquire and release.

We implemented the same application in Objective Caml using locks and ran both versions with the first author's homepage as input. The web cache has 177 lines of code and three atomic blocks. The AtomCaml version took 7.689 seconds on average to fill a 100MB cache. The Objective Caml version took 7.630 seconds. Thus our overhead for this I/O-bound application was just 0.8%.

6.4 Large Application: PLANet

PLANet [22] is a prototype active network system that allows arbitrary network topologies, extensible routers using a domain-specific language for the extensions [21], and many other features. Separate threads interpret network packets, perform router updates, generate performance statistics, etc. PLANet was last used with version 2.2 of Objective Caml, so we first made minor changes so the code would run with version 3.08.1 (the version from which AtomCaml is derived).[14] We then replaced *all* lock-based synchronization with uses of atomic, though of course we could have left some uses of locks had we preferred. Although we modified all the code, we should note that the experiments we ran exercise some but not all of the system's synchronization.

Most synchronization used the `critical` function described in Section 6.1 and was therefore trivial to switch to atomic. Condition variables were used in several places; in all of them, the transformation described earlier sufficed. One place used Caml's Concurrent ML library; we changed this library not to use locks and the actual PLANet code needed no change. The PLANet code also implemented readers/writer locks on top of regular locks and used them to mediate access to a hashtable. We removed the readers/writer locks and wrapped the hashtable operations in atomic blocks.

At this point, we had five unsafe calls to C functions from within atomic blocks. We deemed one pure (so we moved it to before the atomic block), one delayable (so we moved it to after the atomic block), and three requiring atomic-safe versions of the C code (which we implemented). Two of the three functions performed output (which we buffered); the third killed a thread (which we delayed until the block commits). The atomic-safe versions use the rollback and commit callbacks described in Section 4.3.

In examining the code base, we also discovered three concurrency bugs. First, a “clock” module for performing periodic events would deadlock if one callback attempted to register another callback. None of the PLANet code would do this, but the module's interface suggests it is perfectly reasonable. Such library-reentrancy bugs are probably quite common. Second, the readers/writer locks had a glaring bug: An inverted test would cause a write-lock acquisition to block only if there were waiting readers *and* waiting writers. Third, the same library did not always allow simultaneous readers: When releasing a write-lock when no writers were waiting, the library would signal only one waiting reader rather than broadcasting to all of them. (So if one reader does not terminate, subsequent waiting readers will starve, which is not the desired behavior.) In our opinion, PLANet is well-written code; it is just very difficult to write and test concurrent applications.

Though it may be coincidence, our “port” to use atomic removed all three bugs, and would have even if we had not noticed them: The deadlock was a thread waiting on a lock it already held; this translates to a nested atomic block which is no problem. The readers-writer locks are not used in the atomic version of PLANet.

The resulting logging in the atomic version was small even though we did not make atomic blocks any smaller than the critical regions in the original code. We instrumented the run-time system to record the size of the logs and never found more than 41 logged writes in any atomic-block execution.

As for performance, the PLANet publications already indicate that the Objective Caml code's speed is typically lost in the noise: An active network with routers running in user space spends most of its time copying data from user-space to kernel-space and vice-versa. Nonetheless, we ran basic latency (“ping”) and throughput

[14] The most interesting change had nothing to do with changes to Caml: as of January 10, 2004, the number of seconds since January 1, 1970 no longer fits in a signed 31-bit integer.

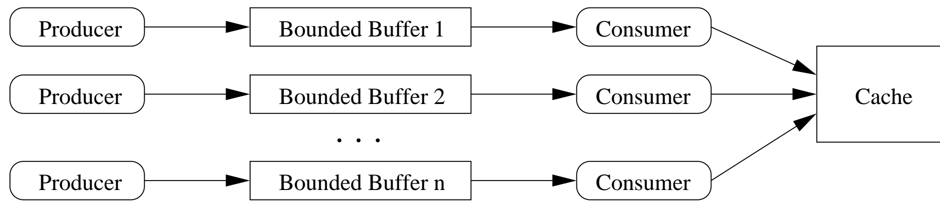


Figure 7. The architecture of our simple web cache. A producer thread scans an HTML page, inserting linked and embedded objects into a bounded buffer. A consumer thread removes objects from the buffer and caches them. If any object is an HTML page, a new producer, bounded buffer, and consumer are created.

("stream") tests for a trivial two-node network (two machines, each with 2.8 GHz Pentium 4 processors and 1 GB of RAM). The latency test took on average 4.2% longer per self-routed ping in the AtomCaml-compiled version (1.198 ms vs. 1.150 ms). However, as we see in Figure 6, the variation between tests was much larger than this difference, so it is probably not particularly meaningful. The throughput test measured the speed at which sender and receiver nodes process data. The AtomCaml compiled receiver node was 7.0% slower than the Objective Caml receiver, but the AtomCaml sender was 9.4% *faster*. There are several reasons that could explain why the locks version of the code is actually slower. For example, a thread could be pre-empted while it holds a lock for some data structure that every thread accesses. Or, the readers/writer lock bug that switching to atomic eliminated may have hurt performance.

6.5 Alternate Closure Representation

The results described above used the two-pointer closure representation depicted in Figure 5b. We also tested the single-pointer representation shown in Figure 5c. As expected, non-atomic code sped up slightly (0–2.5%): Compiling `parser.ml` and `ctype.ml` took 0.586 and 1.012 seconds, respectively, and the interpreter test took 3.407 sec. Applications using `atomic` were unchanged except for the ping test which slowed down by 7%: The web cache took 7.718 s, the ping test took 1.278 ms, and the send and receive tests took 0.0670 and .0894 s/MB respectively.

7. Conclusions and Future Work

We have designed, implemented, and evaluated AtomCaml, an extension to bytecode-compiled Objective Caml that supports atomic critical sections. Our design adds two first-class primitives, `atomic` and `yield_r`, and provides support for calling external C functions within atomic blocks. The `atomic` function takes a thunked block of code, evaluates it as if there was no interleaving of other threads, and returns the result. The `yield_r` function suspends the current thread (triggering a rollback if it occurs inside an atomic block) and allows the scheduler to keep it suspended until the reference bound to its argument changes. Our implementation uses logging and rollback to guarantee atomicity. Our approach is appropriate for any system that, like Objective Caml, enforces a uniprocessor execution model (i.e., that does not allow multiple threads to execute simultaneously). We evaluated the efficiency and convenience of AtomCaml by writing libraries, microbenchmarks, and a small application, and by porting a large multithreaded application. The overhead for "real" applications was small, and replacing locks with atomicity removed (at least) three concurrency errors from the application we ported.

The primary drawback to our *current implementation* is the requirement that threads not execute in true parallel. However, many language implementations (e.g., Objective Caml and DrScheme) do not have true shared-memory parallelism and a functional program needing such support would presumably also need a state-of-the-art parallel and/or concurrent garbage collector. Even when multiprocessors become commonplace, we believe many concurrent languages and applications will continue being run such that threads sharing memory are interleaved but not truly parallel. Moreover, many desktop applications (e.g., editors) need concurrency for responsiveness, but it is much less clear that they need the performance of parallel computing.

In addition, we are extending our ideas to settings with true shared-memory parallelism. We have begun preliminary work on our next project: AtomJava. AtomJava will bring the same strong atomicity guarantees present in AtomCaml to an object-oriented setting with support for true parallelism.

Acknowledgments

Manuel Fähndrich and Shaz Qadeer provided helpful early discussions about our approach. Joao Dias, Matthew Fluet, Michael Hicks, and Leaf Petersen provided feedback on earlier drafts. Michael Hicks suggested we look at PLANet.

References

[1] AtomCAML. http://www.cs.washington.edu/homes/miker/atomcaml.

[2] David Bacon, Robert Storm, and Ashis Tarafdar. Guava: A dialect of Java without data races. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 382–400, October 2000.

[3] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–233, 1992.

[4] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230, November 2002.

[5] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 56–59, October 2001.

[6] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–247, 2004.

[7] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 298–309, June 1998.

[8] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *ACM Conference on Programming Language Design and Implementation*, pages 258–269, June 2002.

[9] Cormac Flanagan and Martín Abadi. Types for safe locking. In *European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108. Springer-Verlag, 1999.

[10] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[11] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *ACM Symposium on Principles of Programming Languages*, pages 256–267, 2004.

[12] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 234–245, 2002.

[13] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *ACM Conference on Programming Language Design and Implementation*, pages 338–349, June 2003.

[14] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *ACM International Workshop on Types in Language Design and Implementation*, pages 1–12, January 2003.

[15] Matthew Flatt and Robert Bruce Findler. Kill-safe synchronization abstractions. In *ACM Conference on Programming Language Design and Implementation*, pages 47–58, Washington DC, June 2004.

[16] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (tcc). In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–13, 2004.

[17] Tim Harris. Exceptions and side-effects in atomic blocks. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[18] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, October 2003.

[19] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2005.

[20] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[21] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *ACM SIGPLAN International Conference on Functional Programming Languages (ICFP)*, pages 86–93, 1998.

[22] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott M. Nettles. PLANet: An active internetwork. In *IEEE Computer and Communication Society INFOCOM Conference*, pages 1124–1133, 1999.

[23] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[24] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, 1983.

[25] Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Jan Vitek, and Bin Xin. Preemptible atomic regions for Real-time Java. Technical report, Purdue University, 2005.

[26] V. Krishna Nandivada and Suresh Jagannathan. Dynamic state restoration using versioning exceptions. *Higher-Order and Symbolic Computation*. To appear.

[27] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, San Jose, CA, October 2002.

[28] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[29] Michael F. Ringenburg and Dan Grossman. Types for describing coordinated data structures. In *ACM International Workshop on Types in Language Design and Implementation*, pages 25–36, January 2005.

[30] Algis Rudys and Dan S. Wallach. Transactional rollback for language-based systems. In *International Conference on Dependable Systems and Networks*, June 2002.

[31] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynmaic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[32] February 27, 2005. http://www.securityfocus.com/.

[33] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, Special Issue(10):99–116, 1997.

[34] Avraham Shinnar, David Tarditi, Mark Plesko, and Bjarne Steensgaard. Integrating support for undo with exception handling. Technical Report MSR-TR-2004-140, Microsoft Research, December 2004.

[35] Olin Shivers, James W. Clark, and Roland McGrath. Atomic heap transactions and fine-grain interrupts. In *ACM International Conference on Functional Programming*, pages 48–59, 1999.

[36] Andrew P. Tolmach and Andrew W. Appel. A debugger for standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.

[37] February 27, 2005. http://www.us-cert.gov/.

[38] Christoph von Praun and Thomas R. Gross. Object race detection. In *ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 70–82, October 2001.

[39] Adam Welc, Antony L. Hosking, and Suresh Jagannathan. Preemption-based avoidance of priority inversion for Java. In *International Conference on Parallel Processing*, 2004.

[40] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In *European Conference on Object-Oriented Programming*, 2004.

[41] Jeannette M. Wing, Manuel Fähndrich, J. Gregory Morrisett, and Scott Nettles. Extensions to standard ML to support transactions. In *ACM SIGPLAN Workshop on ML and its Applications*, 1992.

A. AtomCAML Condition Variable Library

```
open Thread
type channel = bool ref
type condvar = channel list ref (*a queue would be
                                       fairer perhaps*)
let create () = ref []
let signal cv =
  atomic (fun () ->
    match !cv with
      [] -> ()
    | hd::tl -> (cv := tl; hd := false))
let broadcast cv  =
  List.iter
    (fun r -> r := false)
    (atomic (fun () ->
      let ans = !cv in cv := []; ans))
let listen cv = atomic (fun () ->
  let r = ref true in
  cv := r :: !cv;
  r)
let wait ch = atomic (fun () ->
  if !ch then yield_r ch else ())
```

B. Logging and Rollback Microbenchmarks

Writes	Execution	Execution Logging	Execution Logging & Rollback
0	0.194	0.195	0.245
10	0.238	0.271	0.327
50	0.368	0.643	0.741
100	0.543	1.064	1.158