SC-Haskell: Sequential Consistency in Languages That Minimize Mutable Shared Heap

Michael Vollmer\textsuperscript{1}  Ryan G. Scott\textsuperscript{1}  Madanlal Musuvathi\textsuperscript{2}  Ryan R. Newton\textsuperscript{1}

Indiana University (USA)\textsuperscript{1},  Microsoft Research (USA)\textsuperscript{2}

\{vollmerm, rgscott, rrnewton\}@indiana.edu  madan@microsoft.com

Abstract

A core, but often neglected, aspect of a programming language design is its memory (consistency) model. Sequential consistency (SC) is the most intuitive memory model for programmers as it guarantees sequential composition of instructions and provides a simple abstraction of shared memory as a single global store with atomic read and writes. Unfortunately, SC is widely considered to be impractical due to its associated performance overheads.

Perhaps contrary to popular opinion, this paper demonstrates that SC is achievable with acceptable performance overheads for mainstream languages that minimize mutable shared heap. In particular, we modify the Glasgow Haskell Compiler to insert fences on all writes to shared mutable memory accessed in nonfunctional parts of the program. For a benchmark suite containing 1,279 programs, SC adds a geometric overhead of less than 0.4% on an x86 machine.

The efficiency of SC arises primarily due to the isolation provided by the Haskell type system between purely functional and thread-local imperative computations on the one hand, and imperative computations on the global heap on the other. We show how to use new programming idioms to further reduce the SC overhead; these create a virtuous cycle of less overhead and even stronger semantic guarantees (static data-race freedom).

Keywords: memory models, functional programming, sequential consistency

1. Introduction

The prospect of sequential consistency (SC)\textsuperscript{21} as a concurrency abstraction for programmers has been debated against a backdrop of mainstream C++ and Java programs. These languages have settled on memory models based on DRF0\textsuperscript{26,8} that guarantee SC for data-race-free programs. However, as Adve and Boehm point out\textsuperscript{3}, this is far from a satisfying solution due to the inherent difficulties in enforcing data-race-freedom and in pinning down the semantics of programs with data races.

This paper rests on the hypothesis that high-level languages should shield all programmers from the counter-intuitive effects of compiler and hardware reorderings by providing SC. The key challenge is that doing so requires the language runtime to insert hardware fences to counter the relaxed memory models of current hardware architectures. Doing so for shared-memory imperative languages, such as C++ and Java, either incurs unacceptable overheads or requires sophisticated whole-program analyses\textsuperscript{36,38,5} that modern language runtimes are loath to implement.

This is not necessarily true for language paradigms that deemphasize access to shared mutable heap when possible. For instance, consider Haskell, Erlang, and Rust, which are designed for functional programming, message passing, and linear ownership, respectively. Programs in these languages naturally use fewer shared heap locations. Moreover, the language type system provides an opportunity to make SC mainstream with acceptable overheads. The goal of this paper is to evaluate this hypothesis for the Haskell language. As functional programming primitives are gradually introduced into mainstream imperative languages, we believe that lessons learned from our efforts here could influence future efforts to tighten the memory models of Java and C++.

Despite its fundamental importance, memory model considerations are usually an afterthought during language design. The current Haskell specification does not formally specify a memory model, a sore spot in a language that otherwise provides a mathematically grounded semantics. We show that this oversight is more than a minor annoyance and demonstrate how a weak memory model can break the fundamental abstractions of Haskell. In particular, in Section\textsuperscript{3} we show that the current GHC implementation on ARM is \textit{type-unsafe} — a particular sharing pattern in the imperative parts of a Haskell program can break the type safety of the functional parts of the program, a core language guarantee.

To comprehensively resolve such issues, this paper proposes SC-Haskell, a refinement of the existing Haskell lan-
The contributions of this paper are:

1. (parallel) functional programming [28],
2. thread-private references (ST type) [22],
3. software transactional memory (STM type) [17], and
4. libraries that encapsulate alias-free partitions of shared mutable state [19].

In all these cases, the type system serves as a firewall preventing code using these mechanisms from modifying unsynchronized (non-transactional, non-atomic) shared mutable variables. Thus, program transformation to ensure SC can be type directed by systematically ignoring all store operations that originate from the four alternate facilities above. Thus our first question is whether the presence of these facilities has already made reliance on unsynchronized shared-heap writes statistically rare in existing Haskell programs, and thus the surface area for SC-enforcement small.

The answer to this question is a resounding “yes”. As described in section 7.2 we find occurrences of unsynchronized, global-heap-mutating code in less than two percent of one hundred thousand publicly available Haskell modules in the Hackage package manager, representing 17 million lines of open source code. Targeting x86-64 architectures, if we attach memory barriers to all writes to shared mutable memory, only 483 benchmarks of 1,279 we surveyed issued any barriers at all (Section 7.3). For the 483 with barrier overhead, the geometric mean slowdown was only 0.4%. Further, only 12 benchmarks slow down more than 10%.

We believe SC is the right choice for the Haskell language moving forward, and, further, that SC-Haskell showcases a methodology for language design that emphasizes modular construction of effects on memory regions. Mixed-paradigm functional and object-oriented languages of the future have the option of reducing reliance on a monolithic shared memory, and instead using separate mechanisms for immutable data, thread-local state, and transactional state, just as Haskell does. The memory models for these languages can be much simpler to specify and simpler for the programmer to use. Finally, for the language designer, separate, modular memory effects can make proving sequential consistency easier, by showing that a specialized memory area is disjoint from the global heap (Section 4).

The contributions of this paper are:

- A formal operational semantics for core Haskell against a weak memory model following total-store-order (TSO). Using this we can show that our implementation strategy for global mutable state preserves SC while not adding overhead to thread-private or transactional state in the ST and STM types, respectively (Section 4).
- To our knowledge, the first practical demonstration that a mainstream concurrent language can provide SC with low overheads on a wide swath of application code. We demonstrate this with an empirical evaluation over a thousand benchmark programs (Section 7).
- A case study where we seek out examples of SC-Haskell adding barriers to core libraries, then demonstrate how these libraries can be rewritten to both avoid synchronization overheads and have much stronger semantic guarantees. In Section 6 we introduce two novel techniques for enforcing locking discipline and thread-private memory respectively.

2. Background: Mutability in Haskell

The first thing to know about Haskell is that functions and expressions are pure, i.e. lacking externally visible side effects. Thus a function of type \( \text{Int} \rightarrow \text{Int} \) takes and returns a number but cannot print to the screen or modify memory.

Ultimately, such effects are a necessary part of programming, and they appear in Haskell in a manner explicitly marked in the types. Specifically, they are expressed in Haskell using monadic actions: explicit, first-class descriptions of effects to perform. Thus a function of type \( \text{Int} \rightarrow \text{IO} \text{Int} \) takes and returns an integer, while also performing additional input/output side effects. Semantically, we view this function is as producing an “IO Int” object: an abstract description of the actions to perform later on. IO actions are realized only at the “main” function of a complete program. The main function provides the “imperative spine” of a program, even if the vast majority of the work happens in pure functions called from this spine.

Because they are first class values, IO actions can be put in a list or passed as inputs to functions. But for most programs this is irrelevant. Rather, a function in the IO monad (i.e., returning an IO action) is compiled to assembly code very similar to that produced from imperative code in other managed languages. From the point of view of the compiler, the monadic treatment is not very different than a type-effect system [24].

**Mutable heap locations** Within the IO monad, Haskell programmers use a simple library API to build and use mutable references:

```haskell
newIORef   :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

In these type signatures, the lower case “a” is a type variable indicating that the above functions are polymorphic and a IORef a is a mutable reference to an object of type a. Thus, newIORef takes an object of type a and returns a reference to a, with the side effect (indicated by IO) of allocating memory to hold the reference. The function readIORef takes
such a reference and returns the value of the object. Haskell functions are usually *curried*, but the type of `writeIORef` could just as well be written `(IORef a, a) → IO ()`; it takes a reference and a value and returns a *monadic action* that writes the value to the reference. Here, `()` is the unit type, equivalent to “void” or an empty tuple.

The monadic actions returned by the above functions are typically combined using `do` notation, e.g.:

```haskell
-- A multi-line do-expression with type "IO Int":
do r ← newIORef 0 ⏯ Initial value is zero
   writeIORef r 3 ⏯ Modify value
   readIORef r ⏯ Returns three
```

When Haskell gained concurrency [32], IORefs could be accessed by multiple threads, by using `forkIO` to create language-level lightweight threads:

```haskell
forkIO :: IO () → IO ThreadId
```

Unfortunately, no memory model was defined to specify which return values can be obtained from a `readIORef`. Indeed, to this date, the documentation says only that:

> **IORef operations may appear out-of-order to another thread, depending on the memory model of the underlying processor architecture**

Our position is that `writeIORef`, *with these semantics, is harmful and unnecessary*: harmful because of debugging difficulties and even bugs on some architectures (Section 3); unnecessary, because it is possible to simply rely on other mechanisms. For starters, atomic operations are already provided for IORefs:

```haskell
atomicWrite :: IORef a → a → IO ()
atomicModify :: IORef a → (a → (a, b)) → IO b
```

Both were implemented in GHC with compare-and-swap. Both incur significant overhead and should be used sparingly, but they need not change to support SC.

**Intercepting mutable state modifications** Because mutable references do not have built-in syntax, it is straightforward to modify their behavior by modifying the library that exposes the above API. The same argument applies to Haskell’s other mutable heap objects: (un)boxed arrays. By intercepting all calls that write to mutable, potentially-shared memory, we can enforce a new memory model. For example, in this work, we attach a store-load barrier to each `writeIORef` (turning it effectively into `atomicWrite`), which yields sequential consistency on any TSO architecture.

### 2.1 Thread-local state: ST monad

Before we go further in changing the semantics of IO-based state, first we look at the alternative mechanisms that allow programmers to avoid IO-based state in the first place. If a programmer needs to, e.g., implement an efficient in-place sort, then there are other monads to turn to in place of `IO`.

A State Thread (ST) computation [22] enables private, thread-local mutable state. An action of type `ST s a` returns a value `a` while performing side effects on a memory region identified by `s`. Mutable memory managed by the ST monad includes references and arrays, but these data are not allowed to escape the state thread. That is, they cannot be read or changed by another ST computation with a different `s` parameter. Furthermore, ST computations cannot fork new threads, so mutable data structures in an ST computation are guaranteed to be used by only one thread.

In order to enforce this, the mutable datatypes—such as the `STRef` counterpart of `IORef`—must also be tagged with an `s` parameter, becoming `STRef s a` instead of `IORef Int`. The function for mutating a reference then becomes:

```haskell
writeSTRef :: STRef s a → a → ST s ()
```

Here the reference’s `s` parameter must match the returned ST action. Eventually, we can eliminate the `s` parameter and embed an ST computation in pure code using `runST`:

```haskell
runST :: (∀ s. ST s a) → a
```

Here is the old trick at the heart of the ST technique: the higher-rank type (indicated by the explicit “∀ s” quantifier) is what enforces the requirement that no mutable data may escape the scope of the ST computation. That is, because of the limited scope of the “∀ s” quantifier, if the programmer attempts to return a value such as an `STRef s a`—i.e. smuggled inside of `runST`’s return value—the compiler will report a type error. The `s` cannot escape its scope.

Critically, in order to enforce an SC memory model for Haskell, we do not need to add any additional synchronization to `STRef` operations, because they are guaranteed by the type system to be used in one thread, in which case SC is already preserved by all major architectures [3] including the x86 TSO architectures we focus on in our experiments.

Indeed, as we will see in Section 6 and Section 7, shifting most of the work in a program to (1) pure functions, or (2) ST computation, reduces the overhead of SC. But there’s also a third category of mutable memory in Haskell—memory which *already* includes implicit synchronization.

### 2.2 Synchronized mem: STM, MVars, and beyond

In addition to the ST monad, Haskell programmers have at their disposal the STM monad, for *software transactional memory* [17]. Like other STM systems, transactions execute atomically. They are also restricted by the type system to be of a distinct type: `STM a` rather than `IO a`, and they mutate only references of type `TVar t`, never `IORef t`.

STM requires its own synchronization and retry strategy, and a correct implementation naturally preserves sequential consistency. As one of the major concurrency mechanisms in modern Haskell we include it in our formal treatment (Section 4) to show how it interacts with our language-level model of relaxed memory.

---


[2] by the program order relation
Haskell also has other forms of data with built-in synchronization, such as MVars, which include a full/empty bit and blocking operations, or the synchronization variables supported by a family of Par monads \[28, 25, 20\] that complement the existing ST and STM monads. We omit these from our formalism, though the treatment for MVars and Par would follow that of STM and ST respectively—no additional synchronization is needed to achieve SC. The real target of our SC-enforcement effort must be IO actions.

2.3 Safe Haskell

Ideally, the scope for enforcing a memory model would be “every program written in language X”, i.e. all IO actions in all Haskell programs. However, even “safe” languages typically contain backdoors. For instance, Rust has the unsafe block, and Java has the JNI, which allows programmers to run unsafe C code. Haskell also has backdoors and under-the-hood primitives. In particular, GHC exposes several functions which can subvert programmer expectations, such as unsafePerformIO, which can break purity.

To avoid these pitfalls (except in specialized expert code), GHC has a Safe Haskell mode \[40\]. Safe Haskell is a subset of Haskell code for which the compiler verifies that certain key properties of a safe programming language holds, including type safety, referential transparency, and encapsulation of modules, by disallowing certain exotic language features and restricting the use of certain unsafe modules.

Safe Haskell provides a natural target for enforcing our memory model. We guarantee that compiling a module or package with the -safe flag results in an SC program. Libraries that use raw, lower level primitives (which may leak the weak memory model) are by default considered “Unsafe” and thus not available to Safe programs. On the other hand, select libraries can be marked as “Trustworthy”, and become part of the trusted code base if they use unsafe features internally but expose only safe external interfaces.

3. Example: GHC type unsafety bug

In general, the “leave it to the architecture” approach to memory models is unacceptable, particularly for languages such as Haskell that otherwise provide a mathematically grounded semantics. However, this oversight is more than a specification problem. We found that the current version of GHC is not type safe on ARM. To see why, let’s look at two ways that values can flow between threads.

First, purely functional computation can contain parallelism, where multiple threads share the same (pure) data structures. Due to lazy evaluation, a delayed computation (thunk) updated on one thread can issue writes that mutate the heap, which are then read by another thread. This scenario is correctly handled, with a memory fence preceding the write that commits a thunk update. Thus writes that initialize a data structure will be visible.

A second way data can flow between threads is IORefs:

\[
\text{-- Force allocation/initialization to happen on the child thread:} \\
\{−# NOINLINE mk #−\} \\
mk x = \text{MkMyStruct } x \\
go = \text{do } r \leftarrow \text{newIORef 0} \\
\]

\[
\text{-- The $!$ operator is strict application, no thunks:} \\
\text{forkIO } (\text{writeIORef r$! mk 3}) \\
\text{readIORef r} \\
\]

In this program the readIORef may or may not observe the new value written to r. Unfortunately, here GHC has a bug as of version 7.10.3. The MkMyStruct heap value must be allocated and its initialization visible to the parent thread’s readIORef. Yet GHC does not currently generate a fence on IORef update in the ARM backend. Thus the read may return an uninitialized value. We plan to fix the bug in GHC 8.2.

Java handles the similar problem of initializing final fields in object constructors by adding a fence at the end of such constructors. This solution, however, is unacceptable in our case as most objects are created in pure computations that never escape to the imperative spine of a Haskell program. Note that our fix erases the difference between writeIORef and atomicWriteIORef and automatically provides SC semantics on TSO architectures.

4. Formal semantics

In this section we introduce a formal model for SC-Haskell as implemented on a weak-memory, TSO architecture. In contrast, the simplest high-level model of SC-Haskell would take for granted a global, sequentially consistent memory and not address the question of how writes and reads are implemented. Here, we instead expose the store buffer which enables relaxed memory and closely follows current x86 hardware. This enables us to explicitly model the implementation strategy of attaching barriers to IO writes (but not ST or STM writes). The “lower level” relaxed-memory operational semantics in this section would also be useful for explicitly reasoning about unsafe code or new extensions.

SC-Haskell is designed such that all well typed programs are sequentially consistent, that is, all programs would produce the same observable behavior under a semantics without store-buffers. The language we present is a combination of a core, purely functional language with a series of extensions for performing different kinds of side effects: IO, ST, and STM. Both purely functional and imperative programs are expressable in this language. To keep the size of the model reasonable, the only kind of mutable memory in the semantics we present are mutable references (not arrays).
Further, we omit features of Haskell that do not directly relate to concurrency.

4.1 Abstract syntax

The abstract syntax for SC-Haskell is given in Figure 2. It is mostly standard. As in [28] and other semantics for subsets of Haskell, we consider monadic actions—e.g., writing to a mutable reference—to be values.

Our subset of SC-Haskell has syntax for `fork`, `runST`, operations on mutable references, and syntax for STM computations. The operations on references are parameterized by a subscript `m`, which is the monad in which the action occurs. Thus `write_m` is a shorthand for `writeIORef`, and so on.

States in SC-Haskell consist of a parallel composition of threads, `P ∥ ... ∥ Q`, where each thread contains a store buffer and a program term. While relaxed memory models are often formalized in a processor-centric rather than thread-centric fashion, it is reasonable to treat store buffers as per-thread (as previous authors have [9]). This is because a realistic implementation issues a memory fence when context switching between threads, which prevents sharing or aliasing of thread-local store buffers.

No special syntax is necessary for introducing fences. The `atomic` syntax, which is used to launch an STM transaction from within the IO monad, is used also wherever a memory fence is needed. Essentially, fence can be defined as an empty transaction: `fence = atomic (return ())`.

4.2 Static semantics for SC-Haskell

The static semantics for SC-Haskell are given in Figure 1. It is straightforward, and is similar previous formulations [22].

To support the `∀` quantifier in the typing judgment for `runST`, the static semantics have both types `T` and type schemes `S`:

```
| Type Variables
| t, s ∈ Type Variables

| Types T ::= t | T_1 → T_2 | () |
| ST T_1 T_2 | STRef T_1 T_2 |
| IO T | IORef T |
| STM T | TVar T |

Schemas S ::= T | ∀t.S
```

Type judgments could be included for reading and writing to references, but they are generally not interesting. Reading an IORef, for example, is a function that takes an r of type `(IORef a)` and returns a value of type `IO a`. `runST` is given its own typing judgment (RUN) to simplify the type system.

4.3 Dynamic Semantics

The main semantics for SC-Haskell are given as a small-step operational semantics via two relations: `→` for pure computation, and `⇒` for monadic computation (Figure 4).

```
APP
Γ ⊢ M_1 : T_1 → T_2  Γ ⊢ M_2 : T_3
----------------------
Γ ⊢ (M_1 M_2) : T_2
```

```
LAM
Γ, x : T_1 ⊢ M : T_2
Γ ⊢ λx → e : T_1 → T_2
----------------------
Γ ⊢ atomic M : IO T
```

```
VAR
Γ, x : S ⊢ x : S
----------------------
Γ ⊢ M : ∀t.S
```

```
SPEC
Γ, x : S ⊢ x : S
----------------------
Γ ⊢ M : S[T/Γ] t ∉ FV(T)
```

```
GEN
Γ ⊢ M : S
Γ ⊢ M : ∀t.S
----------------------
Γ ⊢ M : t ∉ FV(Γ)
```

```
RETURN
Γ ⊢ M : T
----------------------
Γ ⊢ return M : m T
```

```
BIND
Γ ⊢ M : m T_1
Γ ⊢ N : T_1 → m T_2
----------------------
Γ ⊢ (M >>= N) : m T_2
```

```
RUN
Γ ⊢ M : ∀t.ST t T
Γ ⊢ runST M : T
t ∉ FV(T)
----------------------
Γ ⊢ fork M : IO ()
```

```
FORK
Γ ⊢ M : IO ()
----------------------
Γ ⊢ fork M : IO ()
```

Figure 1: Static semantics for SC-Haskell

4.3.1 Reduction rules

Most of the reduction rules in Figure 4 are straightforward, but we nevertheless explain them in turn below. For brevity we omit additional features of the STM monad, such as retrying or transaction composition. A full operational semantics for STM is given in [17].

Reading from mutable references happens in one of two ways. In `READ-LOCAL`, if `r` is mapped to `π` in the thread’s local buffer `Φ`, then `read_m r` evaluates to `return M` (and a side condition ensures that the most recent occurrence of `r` is used). Otherwise, in `READ-GLOBAL`, when `r` does not occur in `Φ`, it is looked up in the heap, `Θ`.
A FLUSH rule non-deterministically moves entries from a thread’s local buffer to the global store. It pulls elements out of the right side of Φ, which acts as a FIFO buffer. The only way entries are added to the global store in the language is through FLUSH, so only this rule must be considered when reasoning about writes to the store.

When creating a new reference in NEW, a fresh reference r is created, and local buffer Φ is extended on the left with a new mapping. This is where Δ is needed, to enable globally unique names that do not occur in the store or any thread’s store buffer. The new serves for all three monads.

WRITEIO reduces a write action write\textsubscript{IO} r M to atomic (return ()), while appending the write of m to r to the local buffer. Because writes are performed purely for side effects, they return a unit value, or ().

In rule FORKIO, on fork M a single thread forks into two threads, with the child thread added to the pool to execute the term m. Child threads are executed for effect—they never return a value, and there is no thread-join. Before forking, a thread must have flushed its buffer.

Rule WRITE\textsubscript{STM}(M) is similar to WRITEIO, but leaves out the memory fence. Because the type system ensures that STRefs are thread-local, it is not necessary to ensure that writes to STRefs move to the global store. Likewise, STM writes are synchronized by other means, whose implementation is not modeled explicitly, but is abstracted into the ATOMIC rule.

RUN\textsubscript{ST} states that if a term k reduces to a result k via some ST computation, then the pure term run\textsubscript{ST} M can be reduced to to k, effectively turning an ST computation into a pure computation. One interesting aspect is that when a run\textsubscript{ST} completes, all remaining state in the store buffer and the (local) store become garbage.

In ATOMIC, if k reduces via a multi-step STM computation to return N, then atomic M can be reduced in the IO monad to return N. The rule also enforces that the thread-local buffer was flushed during the STM computation.

### 4.4 Sequential Consistency: Proof sketch

Toward proving SC, we propose a new reduction relation \( \equiv \), which corresponds to reduction relation \( \equiv \) minus the use of local store buffers. A translation function takes a configuration \( P; Δ; Θ \) to an equivalent configuration that lacks a store buffer:

\[
\overline{T}[P || Q] = T[P] || T[Q]
\]

\[
\overline{T}[\langle \rangle ; M; Δ; Θ] = M; Δ; Θ
\]

\[
\overline{T}[\langle Φ(r, N) \rangle ; M; Δ; Θ] = T[\langle Φ \rangle ; M; Δ; Θ][r \mapsto N]]
\]

It is important that \( T[\] \) collapses Φ into Θ by copying elements from right-to-left, just as FLUSH moves writes from the right of the buffer into the store. A buffer may contain several writes to a reference r, and the final state of Θ should reflect those writes happening in the correct order.

\[
x, y \in \text{Variables} \quad r \in \text{Refs}
\]

Terms \( M, N ::= x | V | M N | \text{runST} M | \cdots \)

Values \( V ::= λx \rightarrow M | \text{return} M | M >>= N | \text{fork} M | \text{atomic} M | \text{new}_m M | \text{write}_m r N | \text{read}_m r \)

Monads \( m ::= IO | ST | STM \)

States \( P, Q, R ::= \langle Φ \rangle M | P || Q \)

Buffer \( Φ ::= \cdot | Φ | Φ (r, M) \)

Figure 2: Abstract syntax for SC-Haskell programs, and for abstract machine states.

\[
P || Q \equiv Q || P \quad P || (Q || R) \equiv (P || Q) || R
\]

\[
P || R; Δ; Θ \equiv \equiv m \quad Q || R; Δ'; Θ' \equiv \equiv m
\]

\[
P \equiv P' \quad P; Δ; Θ \equiv \equiv m Q' || Δ'; Θ' \equiv \equiv m Q' \equiv Q
\]

Figure 3: Congruence rules for states

For simplicity we omit most of the details of \( \equiv \).

The rules are a direct simplification of those in Figure 4.

For example, translating the ATOMIC rule simply involves removing the barriers from the configurations:

\[
\text{ATOMIC'}
\]

\[
M; Δ; Θ \equiv \equiv m \quad \text{return} N; Δ'; Θ'
\]

\[
\text{atomic M; Δ; Θ} \equiv \equiv m \quad \text{return} N; Δ'; Θ'
\]

Lemma 4.1 \( T[\langle Φ \rangle \text{read}_m r; Δ; Θ]\) is equivalent to \( \text{read}_m r; Δ; Θ \) in \( \equiv \).

Proof Follows from the definition of \( T[\] \).

Lemma 4.2 \( T[\langle Φ \rangle \text{write}_\text{ST} r M; Δ; Θ]\) is equivalent to \( \text{write}_\text{ST} r M; Δ; Θ \) in \( \equiv \).

Proof Follows from the definition of \( T[\] \).

Lemma 4.3 ATOMIC (return ()) is equivalent to return () under the \( \equiv \) relation.

Proof Follows from the ATOMIC’ rule.

Theorem 4.4 Given a term M and a name environment Δ such that \( \langle \rangle ; M; Δ; \equiv \equiv m \quad Φ \rangle \text{return} N; Δ; Θ \), then \( T[\langle \rangle ; M; Δ; \equiv \equiv m \quad Φ \rangle \text{return} N; Δ; Θ] \).

Proof Sketch RUN\textsubscript{ST} is defined so that non-interference is “baked in”—we start with an empty buffer and store, and the types ensure that only thread-local references are written to them. From lemma 4.1 and lemma 4.2, reading from and writing to the local buffer is equivalent to reading from and
\[
\begin{align*}
M & \rightarrow M' \\
MN & \rightarrow M'N \\
\text{PURE} & \quad M \rightarrow N \quad \langle \Phi \rangle [\text{E}[M]; \Delta; \Theta] \xrightarrow{m} \langle \Phi \rangle [\text{E}[N]; \Delta; \Theta] \\
\text{BIND-RETURN} & \quad \langle \Phi \rangle [\text{E}[\text{return } M \Rightarrow N]; \Delta; \Theta] \xrightarrow{m} \langle \Phi \rangle [\text{E}[N M]; \Delta; \Theta] \\
\text{NEW} & \quad \langle \Phi \rangle [\text{E}[\text{new } m M]; \Delta; \Theta] \xrightarrow{m} \langle \Phi \rangle [\text{E}[\text{return } r]; \Delta, r; \Theta] \quad r \notin \Delta \\
\text{READ-LOCAL} & \quad \langle \Phi_1 \rangle [\text{E}[\text{read } m r]; \Delta; \Theta] \xrightarrow{m} \langle \Phi_1 \rangle [\text{E}[\text{return } M]; \Delta; \Theta] \\
\text{READ-GLOBAL} & \quad \langle \Phi \rangle [\text{E}[\text{read } m r]; \Delta; \Theta] \xrightarrow{m} \\
& \quad \langle \Phi \rangle [\text{E}[\text{return } M]; \Delta; \Theta] \quad r \notin \text{Dom}(\Phi) \\
\text{FLUSH} & \quad \langle \Phi \rangle [\text{E}[\text{return } M]; \Delta; \Theta] \xrightarrow{m} \langle \Phi \rangle [\text{E}[\text{return } r]; \Delta; \Theta] \\
\text{WRITEIO} & \quad \langle \Phi \rangle [\text{E}[\text{write } M]; \Delta; \Theta] \xrightarrow{m} \langle \Phi \rangle [\text{E}[\text{return } ()]; \Delta; \Theta] \\
\text{FORKIO} & \quad \langle \Phi \rangle [\text{E}[\text{fork } M]; \Delta; \Theta] \xrightarrow{m} \langle \Phi \rangle [\text{E}[\text{return } ()]; \Delta; \Theta] \\
\text{WRITEST(M)} & \quad \langle \Phi \rangle [\text{E}[\text{write } m r M]; \Delta; \Theta] \xrightarrow{m} \langle \Phi \rangle [\text{E}[\text{return } ()]; \Delta; \Theta] \\
\text{RUNST} & \quad \langle \Phi \rangle [\text{return } M]; \Delta; \Theta] \xrightarrow{m} \langle \Phi \rangle [\text{return } N]; \Delta'; \Theta' \\
\text{ATOMIC} & \quad \langle \Phi \rangle [\text{atomic } M]; \Delta; \Theta] \xrightarrow{m} \langle \Phi \rangle [\text{return } N]; \Delta'; \Theta' \\
\end{align*}
\]

Figure 4: Reduction rules for the pure and monadic relations

writing to the store after translation. Reads directly from the store are unchanged.

**Theorem 4.5** Given a state \( P \) and store \( \Theta \) such that \( P; \Delta; \Theta \xrightarrow{\text{storeLoadBarrier#}} \langle \Phi \rangle [\text{return } Q; \Delta'; \Theta'] \), then \( \mathcal{T}[P; \Delta; \Theta] \xrightarrow{\text{storeLoadBarrier#}} \langle \Phi \rangle [\text{return } Q; \Delta'; \Theta'] \).

**Proof Sketch** Similar to theorem 4.3

**Theorem 4.6** Given a state \( P \), a name environment \( \Delta \), and store \( \Theta \) such that \( P; \Delta; \Theta \xrightarrow{\text{storeLoadBarrier#}} \langle \Phi \rangle [\text{return } Q; \Delta'; \Theta'] \), then \( \mathcal{T}[P; \Delta; \Theta] \xrightarrow{\text{storeLoadBarrier#}} \langle \Phi \rangle [\text{return } Q; \Delta'; \Theta'] \).

**Proof Sketch** We proceed by induction on the monadic reduction relation \( \xrightarrow{\text{storeLoadBarrier#}} \). Assuming, for all reductions \( R \) of size \( N \) in the relation \( \xrightarrow{\text{storeLoadBarrier#}} \), there exists an equivalent reduction \( R' \) in the relation \( \xrightarrow{\text{storeLoadBarrier#}} \).

The cases for reduction rules \( \text{BIND-RETURN} \), \( \text{NEW} \), \( \text{FLUSH} \), \( \text{FORKIO} \), \( \text{WRITEST(M)} \), and \( \text{PURE} \) are straightforward.

In the case of rules \( \text{READ-LOCAL} \) and \( \text{READ-GLOBAL} \), either a reference \( r \) occurs in \( \Phi \), or it is absent from \( \Phi \) and present in \( \Theta \). The latter case is trivial, and the former case follows from lemma [4.1]

In the case of \( \text{WRITEIO} \), we know by lemma [4.3] that \( \text{WRITEIO} \) is equivalent to \( \text{WRITEST(M)} \). From there we can use lemma 4.2.

The **ATOMIC** case can be proven from theorem 4.5

5. **Implementation**

Our changes to GHC were relatively minor. It is possible to narrow down the safe uses of global heap mutation to a handful of functions in the Haskell standard libraries. Therefore, patching those functions to insert memory barriers is sufficient to make GHC sequentially consistent. This precisely follows the formal model of the previous section.

The abstraction of SC is useful to the programmer only if the provided granularity is at least as large as individual program variables. Haskell IORefs have the nice property that sharing occurs at the granularity of a reference and all hardware architectures enable (properly-aligned) addresses to be read and written atomically. Implementing unboxed arrays (Data.Vector.Unboxed) requires care, e.g., to provide atomic 64-bit accesses on 32-bit architectures. In GHC, vectors use type families (type-level functions), so their representation can be changed based on the element type. This makes it straightforward to implement an additional level of boxing for Vector Double on 32-bit architectures.

5.1 **Barrier insertion**

We added a new primitive operation to GHC 7.10.3 called storeLoadBarrier#, which, when compiled down to x86 assembly, emits a lock instruction followed by an addq. Although storeLoadBarrier# can work in any state-transforming monad, which includes ST, in practice we only use storeLoadBarrier# in the presence of IO. Hence, we need only introduce one more function, storeLoadBarrier :: IO (), which encapsulates storeLoadBarrier# in an IO context.

Equipped with storeLoadBarrier, we searched through base, Haskell’s standard library, for any Safe or Trustworthy modules with functions that expose the ability to mutate the global heap. This includes the API for IORefs as well as other primitive data structures, such as mutable arrays. For each function that directly writes to global memory, we inserted a storeLoadBarrier to ensure that SC is upheld.

5.2 **Low-level libraries**

PATCHING base is not the end of the story, since many Haskell libraries bypass base and use mutable references from another library called primitive. Libraries which utilize
primitive include the widely used vector library. To ensure that we obtained an accurate picture of how SC affects Haskell libraries in the wild, we also patched the primitive library to make use of `storeLoadBarrier` after writes.

Crucially, primitive abstracts over both ST and IO with a type parameter `m`. In our modification, we emit barriers only when `m` ultimately equals IO. A consequence of this is that programmers can use `primitive` to write functions that are polymorphic over whether they will need a barrier on writes to enforce SC. For the most part, adding barriers did not affect the existing Safe Haskell properties of base (though we did need to change the type of `stToIO`, as described in Appendix A.1).

6. Case studies: auditing fenced references

Before diving into the results, we examine uses of non-atomic writes found in commonly used Haskell code. While uses of mutable memory are relatively rare (see Section 7.2) in both core libraries and large-scale applications, one can still find code which mutates memory in the IO monad.

Our large-scale evaluation in Section 7 shows a worst case SC-scenario—only adding the barriers described in Section 5 nothing more. We believe that while the basic technique is already low overhead, that this is only the beginning of a transition for the Haskell ecosystem as a whole. Ultimately, it should be possible to rewrite virtually every use of a `writeIORef` to use another alternative that either (1) removes the overhead, and/or (2) enforces even stronger semantics, such as statically-guaranteed data-race freedom. Specifically, there are many techniques one can employ to transition IORef-heavy code over to ST-style mechanisms.

6.1 Buggy uses of `writeIORef`

Auditing the uses of `writeIORef` in Haskell code can reveal bugs (data races) that expose non-SC behavior. One source of problems is APIs which are unclear about thread safety guarantees. The standard `System.Random` library exposes a global random number generator (as well as the ability to coin local RNGs). The global RNG can be used and updated atomically, which internally uses `atomicModifyIORef`, and it can be read with `getStdGen` and written with `setStdGen`:

```
setStdGen :: StdGen → IO ()
ssetStdGen sgen = writeIORef theStdGen sgen
```

Here the update is non-atomic! Yet the documentation makes no mention that this part of the API is non-thread safe. Therefore, enforcing SC in GHC fixes a bug in random, making the write atomic, and likely fixes bugs in other packages that use `writeIORef` or `modifyIORef` carelessly.

In many cases, the extra memory fence will be enough to provide SC semantics, but not enough to statically enforce a correct thread-safe behavior. In those cases, we can build new type-safe mechanisms to ensure stronger safety properties. We cover two examples in the next two subsections.

6.2 Handle: informal, unenforced locking protocols

The `base` library features a `Handle` datatype used for managing file handle operations. `Handle` uses IORefs in a way such that they cannot be trivially converted to STRefs. The (abridged) definition of `Handle` is:

```
data Handle =
  FileHandle FilePath (MVar Handle__)
data Handle__ =
  Handle__ (IDRef (Buffer Word8))
  (IDRef (Buffer CharBufElem))
  (IDRef (BufferList CharBufElem))
```

A file handle is essentially a group of IORefs that are mutated in response to file I/O. These IORefs are used by multiple threads, but they are only reachable from the MVar in a `Handle`, and that the MVar serves as a lock that protects the IORefs. As mentioned earlier, an MVar is a synchronization variable which obeys a blocking interface:

```
newMVar :: a → IO (MVar a)
putMVar :: MVar a → a → IO ()
takeMVar :: MVar a → IO a
```

An MVar is either full or empty—equivalent to a length-one blocking FIFO. When used as a lock, taking from an MVar is “lock”, and putting it back is “unlock”. As in most imperative languages, this locking mechanism is not enforced by the type system, and thus nothing prevents a programmer from storing a reference to the `Handle__` outside the MVar and performing non-atomic operations on its IORefs.

While SC-Haskell will ensure that all IORef operations are SC, it’s still not a satisfactory solution. Now `Handle` operations must bear the extra overhead of memory fences unnecessarily, since multiple threads shouldn’t be modifying mutable state concurrently in the first place! We wish to enforce the correct locking behavior and avoid the overhead.

**MVarLock solution:** Our proposed solution is to introduce a new data type:

```
newtype MVarLock s a = MVarLock (MVar a)
```

This is just an MVar with an extra type parameter `s`. The `s` represents pieces of state, most notably the state in an ST computation. Holding an MVarLock gives the authority to modify both the underlying value inside the MVar as well as any other state tagged with `s`.

We can refactor `Handle` to make use of MVarLocks:

```
data Handle = ∀ s.
  FileHandle FilePath (MVarLock s Handle__)
data Handle__ s =
  Handle__ (STRef s (Buffer Word8))
  (STRef s (Buffer CharBufElem))
  (STRef s (BufferList CharBufElem))
```

Finally, we need a way to modify the state held by an MVarLock. This is accomplished with:

```
withMVarLock :: MVarLock s a → IO b
  → (a → ST s (a, b))
  → IO b
withMVarLock (MVarLock mv) fn =
```
Of course, this function itself must be a part of the trusted codebase due to the use of unsafeSTToIO. Likewise, MVarLock is an abstract data type, such that the end user cannot access the underlying, raw MVar value.

In withMVarLock, the ST computation serves as a transaction to execute while holding the lock. This goes beyond runST, because the state lives across multiple sessions, intermixed with IO. But the basic idea is the same: The ∀s in the \texttt{Handle} declaration ensures statically that no references to the mutable state can escape. Dynamically, withMVarLock ensures that the state is modified only while holding the lock.

While analogous to the formulation of locking in Rust, MVarLock is actually more general, because it allows modifying state that is stored outside of the value pointed to the lock itself (i.e. the \texttt{a} in MVarLock \texttt{s a}). To our knowledge, this is a novel formulation of type-enforced locking, which is lighter-weight than static solutions that require full permission types \cite{29} or session types \cite{39}.

### 6.3 A web server with unnecessary IORefs

In the evaluation (Section \ref{evaluation}), we use a web server as an example application to measure SC-Haskell’s overhead, because it is necessarily IO intensive, and a likely candidate for high SC-overhead. In examining the \texttt{snap} web server\footnote{http://snapframework.com/}, we found that because most of the application involves IO, there are in fact several uses of IORefs that could potentially be avoided. For instance, inside the large record type that represent client requests, the request body is an IORef:

```
data Request = Request { ... 
  rqBody :: IORef SomeEnumerator 
  ... } 
```

Following the decision to use this and other IORef fields, there are 29 uses of writeIORef across the three core snap packages\footnote{snap-0.14.0.7, snap-core-0.9.8.0, snap-server-0.9.5.1}. The request body is either initialized or rewritten in several places. Because snap is a multi-threaded application, it would increase confidence that accesses to \texttt{rqBody} are non-racy by simply using atomicModifyIORef to update \texttt{rqBody}. But this could introduce substantial overhead because an atomic modification is roughly ten times as expensive as an unsynchronized write (Table \ref{table:atomic-modification}).

A better solution in this case is to enforce \texttt{snap}’s invariant that separate clients are handled by separate threads, and each client’s data structures are modified only by a single thread. We propose a simple solution—not to use locks or atomic updates—but instead to dynamically enforce that the thread accessing the state is the same thread that created it:

```
do x ← takeMVar mv 
  (a,b) ← unsafeSTToIO (fn x) 
  putMVar mv a
  return b
```

Here we create a token \texttt{TLState s}, which binds together the thread ID (dynamic value) and the type of the state. Typically, the \texttt{b} value above must use an existential type to capture both the \texttt{TLState} and the references it protects. For instance, this data type that stores a token together with a mutable reference to an integer:

```
data Pk = ∀s. Pk (TLState s) (STRef s Int64)
```

We can create a new \texttt{pk} value, hiding \texttt{s}, as follows:

```
do e ← newTLState (λtls → 
  do r ← newSTRef 3
    return (Pk tls r))
```

And then peek inside the value and modify it:

```
case e of
  Pk t r → withTLState t (writeSTRef r 4)
```

To our knowledge, \texttt{TLState} is novel and does not previously occur in the literature or folklore. Yet it is a simply way to enforce that thread-private state stays thread-private. Further, while Table \ref{table:atomic-modification} shows that the overhead of \texttt{withTLState} is about the same as a memory fenced writeIORef (7 or 8 nanoseconds), there are several benefits:

1. The measurement for a fenced IORef is optimistic because, architecturally, the cost of a fence depends on how full the store buffer is. When benchmarking fenced writes in a loop, the buffer is always empty.
2. The cost of \texttt{withTLState} can be amortized by checking the token once and then performing an arbitrary number of \texttt{ST} operations against the state. E.g. \texttt{withTLState} can be lifted outside of loops.
3. \texttt{TLState} provides stronger semantics than an SC-preserving IORef, guaranteeing data-race freedom rather than SC.

In summary, MVarLock and \texttt{TLState}, enable small reductions in overhead. However, there’s limited room to improve, as most Haskell programs have little performance impact for SC, as we’ll see in the next section. Rather, these techniques provide a continuum for improvement in semantic guarantees: SC becomes the new baseline, and static data-race freedom becomes the goal, with a whole ecosystem full of code that needs to be pushed up the slope.

### 7. Evaluation

Unless otherwise mentioned, we run our benchmarks on a cluster of identically configured dual-socket Xeons (E5-2670 2.60GHz) with 32GB of memory. Not all benchmarks are parallel, but all are compiled to support parallelism and linked with the threaded runtime system; thus they pay the cost for SC whether or not they are multithreaded.

For all our results, we use the \texttt{criterion} package\footnote{http://criterion.readthedocs.io/en/stable/} to report the marginal cost of performing one additional operation. This is done by repeatedly measuring 20 seconds worth
of a particular computation and fitting a linear-regression model to the timing results. This methodology can perform timing measurements at nanosecond granularities.

We also include limited results from an NVIDIA Jetson TK1 board, with a quad-core ARM Cortex A15 CPU. We were not able to run more extensive benchmarks due to the limited portability of many Haskell libraries.

7.1 Microbenchmarks

Reference overheads First, we look at the overheads of individual IORef operations as shown in Table 1. One takeaway is that MVar-based locks are expensive. A spinlock could probably do somewhat better, but an atomicModifyIORef is already more efficient. Thus MVarLock should only be used when the program already has MVars, as in the Handle case study. The cost of the TLState operation depends on how efficiently the thread-ID operation is implemented, and whether storing the token value inhibits other compiler optimizations.

Trust the standard library? We described the Handle implementation in Section 6.2. It is one of fourteen components of the standard library that we found which use IORefs internally. We audited these and decide that several, including Handle can become part of the trusted code base. They need no fences to guarantee SC. Others, like System.Random, expose get and set methods and require barriers.

We tested the performance impact of trusting versus not trusting the Handle/ByteString implementations. We tested the overhead of getLine, which internally updates several IORefs while holding the MVar lock. We found that each getLine call executes exactly two memory fences if run untrusted. In this case, the optimization to eliminate the memory fences—which are redundant with taking the MVar lock—has a small but measurable impact. Using linear regression, we computed the cost of each getLine as 220ns with the barriers and 212ns without (with an \( R^2 \) fit of 1.000). In this case, we tried to maximize the overhead by making the lines themselves very short: two characters. The result reinforces the intuition that “stray” fences are unlikely to incur substantial overhead unless they are inside a tight inner loop: true IO or other substantial work in the loop is likely to amortize the cost of stray fences.

<table>
<thead>
<tr>
<th>Method</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>writeSTRef</td>
<td>1.67ns</td>
</tr>
<tr>
<td>writeIORef (w/barrier)</td>
<td>7.24ns</td>
</tr>
<tr>
<td>atomicModifyIORef</td>
<td>11.2ns</td>
</tr>
<tr>
<td>MVarLock/writeIORef</td>
<td>28.1ns</td>
</tr>
<tr>
<td>TLS Check/writeIORef</td>
<td>7.78ns</td>
</tr>
<tr>
<td>read ThreadId only</td>
<td>1.81ns</td>
</tr>
</tbody>
</table>

Table 1: Reference update costs. Here we use linear regression to compute the marginal cost of one loop iteration that writes a boxed integer into a mutable reference.

7.2 Static code survey: Hackage

Hackage is the central repository of open-source Haskell packages on the Internet. At the time of writing, Hackage contains 9,017 packages, which contain 17,435,314 lines of Haskell code. We searched all Hackage packages to ascertain the number of uses of synchronized writeIORef and modifyIORef functions, in order to estimate the approximate frequency of packages susceptible to non-SC behavior and to overhead under with SC-Haskell.

Of the 9,017 packages, 379 (4.20%) contain at least one use of modifyIORef, and 816 (9.05%) contain at least one use of writeIORef. Note that under TSO, SC-Haskell does not insert fences on packages that only contain readIORef.

A package typically contains multiple modules such as Data.Map and Data.Set. In Table 2 we measure the occurrences of the IORef operations as a percentage of modules and lines of code. One thing that we see is that even among modules that had writeIORef, the number of occurrences per module was less than three on average. writeIORef is used sparingly, which is consistent with our expectation that it is usually used for configuration, or coarse-grained communication. Thus we predict that the probability that writeIORef is used in an inner loop (such that fence overhead will be substantial) is low. But to evaluate this, we need benchmarks that evaluate a large cross-section of Haskell code.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Lines</th>
<th>Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>writeIORef</td>
<td>4,450 (0.03%)</td>
<td>1,504 (1.47%)</td>
</tr>
<tr>
<td>modifyIORef</td>
<td>1,631 (0.009%)</td>
<td>564 (0.55%)</td>
</tr>
<tr>
<td>Total non-atomic</td>
<td>6,028 (0.03%)</td>
<td>1,767 (1.73%)</td>
</tr>
<tr>
<td>atomicModifyIORef</td>
<td>1,263 (0.007%)</td>
<td>460 (0.50%)</td>
</tr>
<tr>
<td>atomicWriteIORef</td>
<td>53 (0.0003%)</td>
<td>22 (0.02%)</td>
</tr>
<tr>
<td>Total atomic</td>
<td>1,310 (0.008%)</td>
<td>469 (0.46%)</td>
</tr>
</tbody>
</table>

Table 2: Frequency of IORef operations within all packages.
for \( n \)-body, the runtime was 0.8% slower. The overall geometric mean runtime for SC-Haskell was 3.3% slower than stock, however. While \( n \)-nucleotide and \( n \)-body are long-running, \texttt{nofib} unfortunately contains many short-running benchmarks that contribute to noise in this geometric mean. For example, the \texttt{ic88} benchmark ran for 0.212 seconds on stock GHC and 0.229 seconds on SC-Haskell.

We were also able to evaluate \texttt{nofib} on ARM\(^7\) Here we added additional fences to all \texttt{readIORef} operations. The overhead remained low: the overall geometric runtime for SC-Haskell was 1.9% slower than stock.

Noise notwithstanding, the main take away is a negative result—that this repository of large benchmark applications does not make substantial use of mutable references or arrays in a way that would incur an unacceptable SC penalty.

**Stackage:** A broader set of benchmarks comes from analyzing Stackage. Stackage is an online repository of curated Haskell libraries that have been confirmed to compile when built against the other libraries in Stackage. Therefore, a simple and effective way to run many working Haskell benchmarks in the wild is to download all packages from Stackage and filter the ones that contain benchmarks. These contain everything from data structure benchmarks to application benchmarks (such as the \texttt{pandoc} package).

We ran the benchmarks from the LTS-5.16 Stackage resolver, which uses GHC 7.10.3, the same version of GHC that we modified. From that resolver, we uncovered 61 benchmark suites that ran to completion, with a grand total of 1,279 individual benchmarks. Of those, 19 benchmark suites (483 individual benchmarks) emitted at least one write barrier when compiled with SC-Haskell. In fact, few of these emitted many barriers-per-iteration; the worst offenders are listed in Table 3. We ran benchmarks against stock GHC 7.10.3 (which we refer to as LTS-GHC) as well as SC-Haskell, averaging the results of eight full runs.

Across the 483 benchmarks with added fences, the geometric mean runtime for SC-Haskell was 0.4% slower than LTS-GHC. This is encouraging, but this result hides a large benchmark-specific variance that we discuss below.

---

\(^7\) One \texttt{nofib} test had to be disabled on ARM due to a codegen bug.
8. Related works and conclusion

Defining the memory consistency models of shared-memory programming systems is a long-studied area (see, for instance, [2]). Sequential consistency [21] is widely accepted as the memory model that programmers should work with. For instance, foundational works such as Lamport [21] and Shasha-and-Snir [36] equate “correct” execution with an SC execution. Moreover, DRF0 [4] together with its manifesta-
tions in the recent standardization of Java [26] and C++ [8] require programmers to follow a data-race-free programming discipline in order to achieve SC.

One way to provide SC is to build a SC-preserving compiler [27] that targets an SC processor. While efficient research proposals exist for efficient SC hardware [15, 34, 12, 6, 23, 37, 13], current commercial processors do not implement SC and it is unclear if future ones will do so.

Barring hardware SC support, the only option is for programmers, compilers, and/or language runtimes to insert sufficient fences to provide the desired semantics at the programming language interface. Recently, mainstream languages such as C++ and Java have finalized a memory model based on DRF0, with the intention that programmers follow a data-race-free programming discipline. However, enforcing data-race-freedom is hard statically [33, 30] and dynamically [14]. Without tool support, even expert programmers are prone to inadvertently introduce weak-memory-model behaviors in their programs.

Sasha and Snir [36] propose a static program analysis to insert as few fences as needed in imperative programs so as to ensure SC. Building on this, Sura et al. [38] and Kamil et al. [18] present compiler techniques for automatically ensuring SC in Java programs, and Alglave et al. present a scalable tool for automatically inserting fences in C programs [5]. All of these techniques rely on a whole-program analysis that is difficult to scale beyond moderately-sized programs.

Given these difficulties, a viable approach for efficiently providing SC is to restrict the use of shared-memory in programming languages [11, 10, 7, 16]. Boyapati et al. [11, 10] use ownership types to only allow a shared-memory access when holding appropriate locks. Deterministic Parallel Java [7] statically guarantees determinism and data-race-freedom (and hence SC) through a sophisticated type-and effect system. Similarly, Gordon et al. [16] propose a type system where mutations to objects are only performed by threads that own a unique pointer to that object. However, these approaches create a significant programmer burden in terms of type annotations to ensure SC.

This paper observes that a functional programming language, such as Haskell, naturally controls the mutable updates. The language runtime can guarantee SC by simply inserting fences at all potentially shared-memory accesses, because there aren’t very many. The language designer can add new memory effects by adding new monads, and can prove SC by proving that the new effects do not interfere with the main, IO monad, evaluation. This paper, to the best of our knowledge, is the first to demonstrate the empirical feasibility of this approach.


A. Appendix

A.1 Safe-Haskell changes: stToIO

For the most part, adding barriers did not effect the existing Safe Haskell properties of base—that is, modules that were marked Safe and Trustworthy continued to be so after our changes. There is one exception, though: a function with the type signature

\[
\text{stToIO :: ST RealWorld \to IO a}
\]

that is used within a module marked Trustworthy. This poses a problem for an SC-aware definition of trustworthiness, because use of stToIO can violate a critical assumption about ST. Namely, we have previously assumed that all ST computations occur within a single thread, but if ST is allowed to be lifted up to IO, that assumption no longer holds.

Here is an example of problematic code:

```haskell
main :: IO ()
main = do
  x ← stToIO $ newSTRef (0 :: Int)
  y ← stToIO $ newSTRef (0 :: Int)
  forkIO $ do
    stToIO $ writeSTRef x 1
    y' ← stToIO $ readSTRef y
    putStrLn $ "T2: x=1, y=" ++ show y'
  stToIO $ writeSTRef y 1
  x' ← stToIO $ readSTRef x
  putStrLn $ "T1: y=1, x=" ++ show x'
```

This code contains two threads contending on the values of STRefs, and since writeSTRef does not emit barriers, this can result in data races that expose non-SC behavior!

The simplest solution to this problem is to modify the type signature:

\[
\text{stToIO :: } (\forall s. \text{ST } s \to \text{IO } a)
\]

to disallow the code above, and export the original definition of stToIO from a module marked as Unsafe.

A.2 nofib barriers

The below tables include the largest barrier-counts performed by nofib benchmarks, before and after including our optimized version of file handles (both SC-Haskell).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Barriers</th>
<th>Benchmark</th>
<th>Barriers</th>
</tr>
</thead>
<tbody>
<tr>
<td>k-nucleotide</td>
<td>20</td>
<td>n-body</td>
<td>35</td>
</tr>
</tbody>
</table>

Table 5: Number of memory barriers performed on nofib benchmarks, without the barriers resulting from writing to file handles.

B. Artifact description

The artifact bundled with this paper is a virtual machine image set up with two copies of the Glasgow Haskell Compiler: an unmodified (“vanilla”) build of GHC 7.10.3, and a build of GHC 7.10.3 with the modifications we describe in the paper.

The virtual machine image is based on Ubuntu 16.04 and includes example programs and benchmark programs. Scripts are included in the virtual machine to automatically run benchmarks and compare their results.

The VM is 5.7GB, and can be downloaded from the following location: [http://doi.org/10.5072/FK22J6F6C](http://doi.org/10.5072/FK22J6F6C) (md5sum 2797486cf47e977b0ef979bc2ebfd660), or by using the DOI to find the current URL.

B.0.1 Hardware dependencies

To successfully complete the benchmarks it is recommended that at least 4GB of RAM is available to the virtual machine (and we have set the default to 8GB, and two cores). Running on one core may require less memory.

B.0.2 Software dependencies

The software dependency is VirtualBox, which is freely available.

B.0.3 Datasets

We used freely available Haskell benchmarks from Stackage, an online source of stable Haskell packages. We used the benchmarks from the LTS-5.16 set, available at [https://www.stackage.org/lts-5.16](https://www.stackage.org/lts-5.16).

B.1 Installation

All required software is already installed on the virtual machine image. The virtual machine uses an account whose password matches the account name (and thus it should not be exposed to a network).

B.2 Experiment workflow

The VM contains step-by-step instructions to run the benchmarks mentioned in Section 7. A brief summary is included here, and full instructions are given in evaluation.md in the home folder of the virtual machine image.

B.2.1 nofib results

The nofib test suite (Section 7.3) is located in `~/nofib`. We ran this with both the original and modified versions of GHC. To compute the results, type the following commands:

```bash
cd ~/nofib
```

Table 4: Number of memory barriers performed on selected nofib benchmarks, unoptimized mode including the barriers upon writing to file handles.
source use_vanilla_ghc
make run
mv nofib-log nofib-vanilla-log
source use_sc_ghc
make run
mv nofib-log nofib-sc-log
nofib-analyse/nofib-analyse \
      nofib-vanilla-log nofib-sc-log

(Note that the “use_” scripts are found in ~/opt/bin and are thus not specific to any directory.)

The nofib-analyse command prints out a chart which compares the compilation and performance characteristics of each program in the test suite when compiled with each of the two versions of GHC. It produces a lot of output, but the “Elapsed” column of the first section is what we are primarily interested in. Positive or negative changes represent increases or decreases in runtime in the second log compared to the first.

Note also that the above instructions execute a single run of nofib. In Section 7.3, we ran nofib 20 times for both the modified and unmodified compiler, and computed the mean runtime for each benchmark in each mode. This especially helps to compensate for noise from a minority of short-running benchmarks.

B.2.2 Stackage results

Next, instructions for running our Stackage benchmarks from Section 7.3. The Makefile we provide runs only a portion of the Stackage benchmarks, as running them all would take about a day. If you want to run all the benchmarks, you can edit ~/sc-haskell/benchmarking/Makefile and remove the parts from the vanilla and sc targets that say NODE_NUMBER=1 TOTAL_NODES=67.

cd ~/sc-haskell/benchmarking
make vanilla
make sc
make compare

Similarly to nofib-analyse, running make compare will print out a detailed comparison of the runtime differences between the benchmarks run with the unmodified and modified versions of GHC, and gives a geometric mean runtime difference at the bottom.

B.3 Evaluation and expected result

For both the nofib and Stackage benchmarks, the expected result will be the difference in runtime between code generated by vanilla GHC and SC GHC.

Limits of virtualized evaluation  Be warned that virtual-machine-based evaluation is non-ideal for this experiment. The purpose of this experiment is to measure small differences in runtime between two variants of a compiler, with programs executing on dedicated machines under a job control system (as found in HPC deployments). We expect that jitter from virtualization may dominate the differences in runtime. Thus VM-based results can reconfirm that the sequentially-consistent compiler (SC-Haskell) did not add a large amount of overhead, they are unlikely to provide an accurate quantification of the effect.

B.4 Experiment customization

The artifact includes both versions of the compiler, and can thus be reused to evaluate other programs than those we have included.

The instructions on the image also describe how to inspect the assembly language output of the compiler when called on an arbitrary program. The example program (in the ~/examples folder) can serve as a starting point to experiment with what code the compiler produces for different Haskell programs.

cd ~/examples
source use_sc_ghc
ghc IORefExample1.hs -O2 -ddump-asm

B.5 About TechEmpower

We included results from the TechEmpower web benchmarks in section 7.4 but opted to exclude them from the virtual machine, as setting up the benchmarks requires three virtual machines to be running, which makes it impractical to encapsulate within this virtual machine.