CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency

JAROSLAV ŠEVČÍK, University of Cambridge
VIKTOR VAFEIADIS, MPI-SWS
FRANCESCO ZAPPA NARDELLI, INRIA
SURESH JAGANNATHAN, Purdue University
PETER SEWELL, University of Cambridge

In this paper, we consider the semantic design and verified compilation of a C-like programming language for concurrent shared-memory computation above x86 multiprocessors. The design of such a language is made surprisingly subtle by several factors: the relaxed-memory behaviour of the hardware, the effects of compiler optimisation on concurrent code, the need to support high-performance concurrent algorithms, and the desire for a reasonably simple programming model. In turn, this complexity makes verified (or verifying) compilation both essential and challenging.

In this paper we describe ClightTSO, a concurrent extension of CompCert's Clight in which the TSO-based memory model of x86 multprocessors is exposed for high-performance code, and CompCertTSO, a verifying compiler from ClightTSO to x86 assembly code, building on CompCert. CompCertTSO is verified in Coq: for any well-behaved and successfully compiled ClightTSO source program, any permitted observable behaviour of the generated assembly code (if it does not run out of memory) is also possible in the source semantics. We also describe some verified fence-elimination optimisations, integrated into CompCertTSO.

Categories and Subject Descriptors: C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: Parallel processors; D.1.3 [**Concurrent Programming**]: Parallel programming; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]

General Terms: Reliability, Theory, Verification

Additional Key Words and Phrases: Relaxed Memory Models, Verifying Compilation, Semantics

Contents

We acknowledge funding from EPSRC grants EP/F036345 and EP/H005633, ANR grant ANR-06-SETI-010-02, and INRIA program *Équipes Associées MM*.

1. INTRODUCTION

1.1. Context

Multiprocessors are now ubiquitous, with hardware support for concurrent computation over shared-memory data structures. But building programming languages with well-defined semantics to exploit them is challenging, for several inter-linked reasons.

At the hardware level, most multiprocessor families (e.g., x86, Sparc, Power, Itanium, and ARM) provide only *relaxed* shared-memory abstractions, substantially weaker than sequentially consistent (SC) memory [Lam79]: some of the hardware optimisations they rely on, while unobservable to sequential code, can observably affect the behaviour of concurrent programs. Moreover, while for some multiprocessors it has long been clear what the programmer can rely on, e.g. the Sparc *Total Store Ordering* (TSO) model [Spa], for others it has been hard to interpret the vendor's informal-prose architecture specifications [SSZN+09; SSA+11]. For x86, we recently proposed *x86-TSO* [SSO+10; OSS09] as a rigorous and usable semantics; we review this in §2.

Compilers also rely on optimisations for performance, and again many common optimisations (e.g., common subexpression elimination, and so on) preserve the behaviour of sequential code but can radically change the behaviour of concurrent programs. Moreover, for good performance one may need concurrency-specific optimisations, for example to remove redundant fence instructions.

Hence, when designing a concurrent shared-memory programming language, where one must choose what memory semantics to provide, there is a difficult tension to resolve. A strong model (such as sequential consistency) would be relatively easy for programmers to understand but hard to implement efficiently, because compiler optimisations will not always be sound and because expensive processor-specific memory fences (or other synchronisation instructions) will be needed to enforce ordering in the target hardware. A second alternative is to take a *data-race-free* (DRF) approach [AH90], with an SC semantics but regarding programs containing races as undefined, relying on synchronisation from the implementations of lock and unlock (or, in C++0x, certain atomic primitives). Precisely defining a satisfactory DRF programming language model is a technical challenge in itself, as witnessed by the complexities in establishing a Java memory model that admits all the intended optimisations [Pug00; MPA05; CKS07; SA08; TVD10], and the ongoing work on C++0x [BA08; BOS+11]. When it comes to concurrent systems code and concurrent data structure libraries, however, for example as used in an OS kernel and in `java.util.concurrent` [Lea99], it seems that neither of the above are appropriate, and instead a weak model is essential. Compiler optimisations are not the main issue here: these low-level algorithms often have little scope for optimisation, and their shared-memory accesses should be implemented exactly as expressed by the programmer. But for good performance it is essential that no unnecessary memory fences are introduced, and for understanding and reasoning about these subtle algorithms it is essential that the language has a clear semantics. Moreover, such algorithms are intrinsically racy. Such code is a small fraction of that in a large system, but may have a disproportionate effect on performance [Boe05], as illustrated by an improvement to a Linux spinlock, where a one-instruction change to a non-SC primitive gave a 4% performance gain [Lin99]. Recognising this, both Java and C++0x aim to provide a strong model for most programming but with low-level primitives for expert use.

In the face of all this intertwined semantic subtlety, of source language, target language, compilation between them, and the soundness of optimisations, it is essential to take a mathematically rigorous and principled approach to relaxed-memory concurrency: to give mechanised semantics for source and target languages and to consider verified (or verifying) compilation between them. In the sequential setting, verifying

compilation has recently been shown to be feasible by Leroy et al.'s CompCert, a verifying compiler from a sequential C-like language, Clight, to PowerPC assembly language [Com09; Ler09a; Ler09b; BL09]. In this paper, we consider verifying compilation in the setting of concurrent programs with a realistic relaxed memory model.

1.2. Contributions

Our first contribution is the design and definition of *ClightTSO* (§3). ClightTSO is not intended to be a general-purpose programming language, but rather a language in which concurrent algorithms can be expressed precisely, and, more importantly, as a test case for reasoning about relaxed-memory computation. It essentially exposes the x86 hardware load and store operations and synchronisation primitives to the programmer, so ClightTSO loads and stores inherit the hardware relaxed-memory TSO behaviour, but can be implemented without memory fences or atomic instructions. (As we discuss in §8, in a full language one would expect to augment these with assumed-local accesses that the compiler is permitted to optimise away, for high-performance sequential code, but that is not our focus here.) The semantic design of ClightTSO turns out to involve a surprisingly delicate interplay between the relaxed memory model, the behaviour of block allocation and free, and the behaviour of pointer equality.

Our second contribution is one of *semantic engineering* (§4). Relaxed memory models are complex in themselves, and a verifying compiler such as CompCert is complex even in the sequential case; to make verifying compilation for a concurrent relaxed-memory language feasible we have to pay great attention to structuring the semantics of the source and target languages, and the compiler and any correctness proof, to separate concerns and re-use as much as possible. We factor out the TSO memory from each language and build small-step 'labellised' semantics, allowing most of the proof to be done by threadwise simulation arguments. A key question for each compiler phase is the extent to which it changes the memory accesses of the program. For many of our phases (7 of 17) the memory accesses of source and target are in exact 1:1 correspondence. Moreover, for four phases the memory accesses are identical except that some values that are undefined in the source take particular values in the target; and one phase (register allocation) has no effect on memory accesses except that it removes memory loads to dead variables. For all these, the correctness of the phase is unrelated to the TSO nature of our memory. That leaves two phases that change memory accesses substantially, and whose proofs must really involve the whole system, of all threads and the TSO memory, and three phases that leave memory accesses in place but change the fences.

Thirdly, we present evidence that our approach is effective (§5). We have implemented a verifying compiler, CompCertTSO, from ClightTSO to x86 multiprocessors, taking CompCert as a starting point. We have proved correctness, in Coq [Coq], for all the CompCertTSO phases between ClightTSO abstract syntax and x86 symbolic assembly code. In addition, we have successfully run the compiler on a number of sequential and concurrent benchmarks, including an implementation of a non-trivial lock-free algorithm by Fraser [Fra03].

Fourthly, we consider compiler optimisations to optimise barrier placement, and the verification thereof (§6). There are many opportunities to perform fence removal optimisations on x86. In particular, if there are no writes between two memory fence instructions, the second fence is unnecessary. Dually, if there are no reads between the two fence instructions, then the first fence instruction is redundant. Finally, by a form of partial redundancy elimination [MR79], we can insert memory barriers at selected program points in order to make other fence instructions redundant, with an overall effect of hoisting barriers out of loops and reducing the number of fences along some execution paths without ever increasing it on any path. The correctness of

one of our optimisations turned out to be more much interesting than we had anticipated and could not be verified using a standard forward simulation [LV95] because it introduces unobservable non-determinism. To verify this optimisation, we introduce *weak-tau simulations*, which in our experience were much easier to use than backward simulations [LV95]. In contrast, the other two optimisations were straightforward to verify, each taking us less than a day's worth of work to prove correct in Coq.

Finally, we describe some experiments running the compiler (§7), reflect on the formalisation process and on the tools we used (§8), discuss related work (§9), and conclude (§10). The proof effort for each compiler phase was broadly commensurate with its conceptual difficulty: some have essentially no effect on memory behaviour, and needed only days of work; a few were much more substantial, really changing the intensional behaviour of the source and with proofs that involve the TSO semantics in essential ways.

This paper extends conference papers in POPL 2011 [SVZN$^+$11] and SAS 2011 [VZN]. The first paper reported on the correctness proof for key phases of the compiler only, whereas now our main theorem is correctness of the entire compiler, adding the (substantial) MachAbs to MachConc phase, and the (more straightforward) Cminor to CminorSel to RTL phases. The second paper described our fence elimination optimisations. The discussion and semantic details have also been expanded throughout. Our development, all mechanised in Coq, is available online (`www.cl.cam.ac.uk/users/pes20/CompCertTSO`).

2. BACKGROUND: x86-TSO

We begin by recalling the relaxed-memory behaviour of our target language, x86 multiprocessor assembly programs, as captured by our x86-TSO model [SSO$^+$10; OSS09]. The classic example showing non-SC behaviour in a TSO model is the store buffer (SB) assembly language program below: given two distinct memory locations x and y (initially holding 0), if two hardware threads (or processors) respectively write 1 to x and y and then read from y and x (into register EAX on thread 0 and EBX on thread 1), it is possible for both to read 0 *in the same execution*. It is easy to check that this result cannot arise from any SC interleaving of the reads and writes of the two threads, but it is observable on modern Intel or AMD x86 multiprocessors.

SB

Thread 0	Thread 1
MOV [x]←1	MOV [y]←1
MOV EAX←[y]	MOV EBX←[x]
Allowed Final State: Thread 0:EAX=0 ∧ Thread 1:EBX=0	

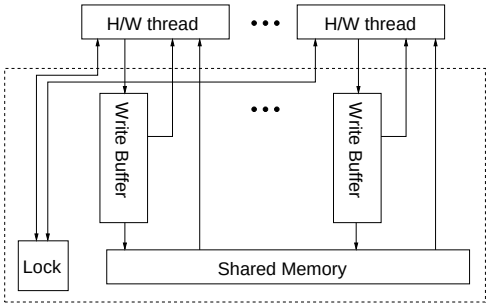


Fig. 1. x86-TSO block diagram

type, $ty ::=$ `void` | `int (`*intsize*`,`*signedness*`)` | `float (`*floatsize*`)` | `pointer (`ty`)`
| `array (`ty`,`*len*`)` | `function (`ty^*`,`ty`)` | `struct (`id`,`ϕ`)` | `union (`id`,`ϕ`)` | `comp_pointer(`id`)`
| `(`ty`)`

fieldlist, $\phi ::=$ `nil` | `(`id`,`ty`)::`ϕ

unary_operation, $op_1 ::=$ `!` | `~` | `-`

binary_operation, $op_2 ::=$ `+` | `-` | `*` | `/` | `%` | `&` | `|` | `^` | `<<` | `>>` | `==` | `!=` | `<` | `>` | `<=` | `>=`

expr, $e ::= a^{ty}$

expr_descr, $a ::= n \mid f \mid id \mid$ `*`e | `&`e | $op_1\ e$ | $e_1\ op_2\ e_2$ | `(`ty`)`e | e_1`?`e_2`:`e_3 | e_1`&&`e_2 | e_1`||`e_2
| `sizeof (`ty`)` | e`.`id

opt_lhs ::= | `(`id`:`ty`)=`

atomic_statement, *astmt* ::= `CAS` | `ATOMIC_INC`

statement, $s ::=$ `skip` | e_1`=`e_2 | *opt_lhs* e'`(`e^*`)` | s_1`;`s_2 | `if (`e_1`) then` s_1 `else` s_2
| `while (`e`) do` s | `do` s `while (`e`)` | `for (`s_1`;`e_2`;`s_3`)`s | `break` | `continue` | `return` *opt_e*
| `switch (`e`)`*ls* | l`:`s | `goto` l | `thread_create(`e_1`,`e_2`)` | *opt_lhs astmt*`(`e^*`)` | `mfence`

labeled_statements, *ls* ::= `default :`s | `case` n`:`s`;`*ls*

fndefn_internal ::= ty id`(`*arglist*`){`*varlist* s`}`

program ::= *dcls fndefns* `main =`id

Fig. 2. ClightTSO abstract syntax (excerpts)

Microarchitecturally, one can view this behaviour as a visible consequence of store buffering: each hardware thread effectively has a FIFO buffer of pending memory writes (avoiding the need to block while a write completes), so the reads from y and x can occur before the writes have propagated from the buffers to main memory.

In addition, it is important to note that many x86 instructions involve multiple memory accesses, e.g. an increment INC [x]. By default, these are not guaranteed atomic (so two parallel increments of an initially 0 location might result in it holding 1), but there are 'LOCK'd' variants of them: LOCK INC [x] atomically performs a read, a write of the incremented value, *and* a flush of the local write buffer. Compare-and-swap instructions (CMPXCHG) are atomic in the same way, and memory fences (MFENCE) simply flush the local write buffer.

The x86-TSO model makes this behaviour precise in two equivalent ways: an abstract machine with an operational semantics, illustrated in Fig. 1, and an axiomatisation of legal executions, in the style of [Spa92, App. K] (the model covers the normal case of aligned accesses to write-back cacheable memory; it does not cover other memory types, self-modifying code, and so on). For the relationship between the model and the vendor documentation, and with empirical testing, we refer to our previous work [SSO$^+$10; OSS09; SSZN$^+$09].

3. ClightTSO

ClightTSO is a C-like language: imperative, with pointers and pointer arithmetic, and with storage that is dynamically allocated and freed, but not subject to garbage collection (GC)[1]. We choose this level of abstraction for several reasons. First, it is what is typically used for concurrent systems programming, e.g. in OS kernels (where garbage collection may be infeasible), and many concurrent algorithms are expressed in C-

[1]Currently this is stack-allocated storage for function local variables, but our development is structured so that adding explicit `malloc` and `free` should be straightforward.

like pseudocode. Second, it is an attractive starting point for research in relaxed-memory programming language semantics and compilation because C source-level shared-memory accesses will often map 1:1 onto target accesses, without the complexity and cost of accesses required for GC. Last but not least, the work of Leroy et al. on CompCert gives us a verifying compiler for sequential programs, so by using that as a starting point we can focus on the issues involved in relaxed-memory concurrency.

Syntactically, ClightTSO is a straightforward extension of the CompCert Clight language [BL09], adding thread creation and some atomic read-modify-write primitives that are directly implementable by x86 LOCK'd instructions. An excerpt of the abstract syntax is given in Fig. 2, where one can see that programs consist of a list of global variable declarations, a list of function declarations, and the name of a `main` function. Function bodies are taken from a fairly rich language of statements and expressions.

Semantically, though, the addition of relaxed-memory concurrency has profound consequences, as we now discuss.

3.1. TSO

Most obviously, the ClightTSO load and store operations must have TSO semantics to make them directly implementable above x86, so we cannot model memory as (say) a function from locations to values. Instead, we use a derivative of the TSO machine in Fig. 1 (the abstract machine style is more intuitive and technically more convenient here than the axiomatic model). We return in §4.5 and §5.1 to exactly what this is.

3.2. Pointer equality

C implementations are typically not memory-safe: one can use pointer arithmetic to corrupt arbitrary state (including that introduced by compilation). But in order to specify an implementable language, C standards rule out many programs from consideration, giving them undefined behaviour. For example, the draft C1X standard states "*If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime*" [C1X, 6.2.4p2]. In Clight the memory state records what is allocated, with equality testing of pointers giving the undefined value (`Vundef`) if they do not refer to currently allocated blocks. However, in a relaxed-memory setting any appeal to global time should be treated with great caution, and the concept of "currently allocated" is no longer simple: different threads might have different views not only of the values in memory but also of what is allocated. For example, in x86-TSO one thread might free, re-allocate and use some memory while another thread compares against a pointer to it, with the writes of the first thread remaining within its buffer until after the comparison. One could make pointer comparison effectful, querying the x86-TSO abstract machine to see whether a pointer is valid w.r.t. a particular thread whenever it is used, but this would lead to a complex and unwieldy semantics. Moreover, comparing potentially dangling pointers for equality is useful in practice, e.g. in algorithms to free cyclic data structures. Accordingly, for ClightTSO we take pointer comparison to always be defined.

3.3. Block reuse

In turn, this means that the ClightTSO semantics must permit re-use of pointers (again contrasting with Clight, in which allocations are always fresh), otherwise it would not be sound w.r.t. the behaviour of reasonable implementations. For example, in the program below `h` must be allowed to return `0` or `1`, as an implementation might or might not reuse the stack frame of `f` for `g`.

```
int* f() { int a; return &a; }
```

```
int* g() { int a; return &a; }
int h()  { return (f() == g()); }
```

3.4. Memory errors and buffering of allocations and frees

A read or write of a pointer that is dangling w.r.t. that thread must still be a semantic error, so that a correct compiler is not obliged to preserve the behaviour of such programs. Now, implementations of memory allocation and free do not necessarily involve a memory fence or other buffer flush: at the assembly language level, stack allocation and free can be just register operations, while heap `malloc` and `free` might often be w.r.t. some thread-local storage. To test whether pointers are valid, therefore, we treat allocations and frees analogously to writes, adding them to the buffers of the TSO machine. This is a convenient common abstraction of stack and heap allocation (for the former, it essentially models the stack pointer).

An allocation must immediately return a block address to the calling thread, but allocations should not clash when they are unbuffered (when they hit the main memory of the TSO machine), so they must return blocks that are fresh taking into account pending allocations and frees from all threads. It is technically convenient if frees and writes also fail immediately, when they are added to the TSO machine buffer, so we also take all possible sequences of the pending allocations and frees into account when enqueuing them. Otherwise one would have latent failures, e.g. if two threads free a block and those frees are both held in buffers.

3.5. Finite memory

A final novelty of ClightTSO, not directly related to concurrency, is that we support finite memory, in which allocation can fail and in which pointer values in the running machine-code implementation can be numerically equal to their values in the semantics. The latter is convenient for our correctness proofs, simplifying the simulations. It also means that pointer arithmetic works properly (mod 2^{32}) and may be helpful in the future for a semantic understanding of out-of-memory errors. The memory usage of a compiled program and its source may be radically different, as the compiler may be able to promote local variables to registers but will need extra storage for stack frames and temporaries. But (analogous to verifying rather than verified compilation), it would be reasonably straightforward to make the compiler emit and check, for each function, bounds on those. One could then reason about real space usage in terms of a source semantics annotated with these bounds.

Without such a space-usage semantics, our correctness statement will be weaker than one might like, in that it does not constrain the behaviour of the compiler at all after the target language has run out of memory. This provides an easy way to write a stupid but nominally correct ‘cheating’ compiler: a compiler could simply (and silently) generate an impossibly large allocation at the start of a compiled program; the target semantics would immediately fail, and so the rest of the behaviour of the compiler would be unconstrained by the correctness statement. In contrast, the unbounded-memory semantics and correctness statement of CompCert prohibits this, but suffers from the dual problem that a verified program compiled with a nominally correct compiler can nonetheless run out of memory and crash when actually executed.

3.6. Small-step semantics

ClightTSO is a concurrent language, in which execution of an expression or a statement may involve multiple memory reads and hence multiple potential interaction points with other threads. We therefore need a small-step operational semantics for both expressions and statements. Conceptually this is routine, but it requires signifi-

cant re-engineering (described in §5.2) of definitions and proofs w.r.t. CompCert, where Clight had a big-step semantics for expressions.

We use a frame-stack style, with thread states that can be an executing expression paired with an expression-holed continuation or an executing statement paired with a statement-holed continuation:

$$\begin{array}{lll} \mathit{expr\_cont},\ \kappa_{\mathrm{e}} ::= & [\mathit{op}_1^{\tau}{}_{\_}] \cdot \kappa_{\mathrm{e}} \quad | \quad [{\_}^{\tau}{=}e_2] \cdot \kappa_{\mathrm{s}} \quad | \quad [v^{\tau}{=}{\_}] \cdot \kappa_{\mathrm{s}} \quad | \quad \ldots \\ \mathit{stmt\_cont},\ \kappa_{\mathrm{s}} ::= & \texttt{stop} \quad | \quad [{\_}\ \texttt{;}\ s_2] \cdot \kappa_{\mathrm{s}} \quad | \quad \ldots \end{array}$$

$$\begin{array}{lll} \mathit{state} & ::= & e \cdot \kappa_{\mathrm{e}} \,|_{\rho} \\ & | & s \cdot \kappa_{\mathrm{s}} \,|_{\rho} \\ & | & \ldots \end{array}$$

Here ρ is a thread-local environment mapping identifiers to their locations in allocated blocks. The semantics is also parameterised by an unchanging global environment of global variables and functions, and additional machinery is needed to deal with l-values, loops, and function calls, which we return to in §5.2. We also fix a left-to-right evaluation order.

In retrospect, we suspect that it would have been simpler to use what we call a *trace-step* semantics for ClightTSO expressions. The small-step operational semantics described above defines defines a small-step transition relation over states involving expression continuations, by case analysis on the structure of those continuations, and with labels that represent internal events or single memory actions. Instead, a trace-step semantics defines a transition relation over states simply involving expressions, and inductively on the structure of expressions, but with lists of labels; given such, one can easily define a small-step semantics by a general construction. The fact that ClightTSO expressions are terminating (as in Clight) makes this particularly convenient. We used such a semantics as an auxiliary definition for the correctness of one of our phases, instruction selection (from Cminor to CminorSel), for which its inductive-on-expressions structure made it easy to re-use many original CompCert proofs. With hindsight, we would do so for all the front-end languages (before RTL).

3.7. Examples

We give a flavour of the language with some very small examples of ClightTSO source programs.

3.7.1. SB. The x86 visible-store-buffer behaviour can now be seen at the ClightTSO level, e.g. if the following threads are created in parallel then both could print `0` in the same execution.

```
                        int x=0; int y=0;
void *thread0(void *tid)       |void *thread1(void *tid)
{ x=1;                         |{ y=1;
  printf("T0: %d\n", y);       |  printf("T1: %d\n", x);
  return(0); }                 |  return(0); }
```

3.7.2. Spinlock using CAS. More usefully, an efficient spinlock can be implemented directly in ClightTSO using `CAS`, where `CAS(p,v_new,v_old)` either atomically replaces the value at pointer `p` with `v_new`, and evaluates to true, if the previous value was `v_old`, or otherwise evaluates to false; in either case it flushes the local store buffer. Any integer variable can be used to represent the state of the spinlock, with lock and unlock as follows:

```
void lock(int *mutex)          | void unlock(int *mutex)
{ while (CAS(mutex, 1, 0))     | { *mutex = 0; }
    while (*mutex) ; }         |
```

The generated assembler mimics the optimised implementation of Linux spinlocks mentioned in Section 1. As shown by Owens [Owe10], the memory update performed by unlock does not need to be synchronising on x86-TSO.

3.7.3. A publication idiom. The memory model supports the common publication idiom below:

```
                double channel; int flag = 0;
// sender                      | // receiver
channel = 5.2;                 | while (flag == 0);
flag = 1;                      | printf ("%f\n", channel);
```

Since the store buffers are FIFO, when the receiver thread sees the update to `flag`, the contents of the `channel` variable must have been propagated into main memory, and as such must be visible to all other threads (that do not themselves have a pending write to `channel`). For contrast, in C++0x [Bec10; BOS+11] (which also targets non-TSO machines), `flag` must be accessed with sequentially consistent atomics, implemented with costly x86 LOCK'd instructions or fences, or with release/acquire atomics, implemented with normal stores and loads but with a much more involved semantics.

4. VERIFYING COMPILER STRATEGY

Having discussed our x86 target language in §2, and the design and rationale of our ClightTSO source language in §3, we now consider the semantics and proof structure required to make a verifying compiler for a concurrent relaxed-memory language feasible.

4.1. Correctness statement

The first question is the form of the correctness theorems that we would like the compiler to generate. We confine our attention to the behaviour of whole programs, leaving a compositional understanding of compiler correctness for relaxed-memory concurrency (e.g. as in the work of Benton and Hur for sequential programs [BH09]) as a problem for future work. The semantics of ClightTSO and x86-TSO programs will be labelled transition systems (LTS) with internal τ transitions and with visible events for call and return of external functions (e.g. OS I/O primitives), program exit, and semantic failure:

$$event,\ ev ::= \texttt{call}\ id\ vs \mid \texttt{return}\ typ\ v \mid \texttt{exit}\ n \mid \texttt{fail}$$

We split external I/O into call and return transitions so that blocking OS calls can be correctly modelled.

Now, how should the source and target LTS be related? As usual for implementations of concurrent languages, we cannot expect them to be equivalent in any sense, as the implementation may resolve some of the source-language nondeterminism (c.f. [Sew97] for earlier discussion of the correctness of concurrent language implementations). For example, in our implementation, stack frames will be deterministically stack-allocated and the pointers in the block-reuse example above will always be equal. Hence, the most we should expect is that if the compiled program has some observable behaviour then that behaviour is admitted by the source semantics — an inclusion of observable behaviour.

This must be refined further: compiled behaviour that arises from an erroneous source program need not be admitted in the source semantics (e.g. if a program mu-

tates a return address on its stack, or tries to apply a non-function). The compiled program should only diverge, indicated by an infinite trace of τ labels, if the source program can. Moreover, without a quantitative semantics, we have to assume that the target language can run out of memory at any time. We capture all this with the following definition of LTS *trace*.

Traces tr are either infinite sequences of non-`fail` visible events or finite sequences of non-`fail` visible events ending with one of the following three markers: end (designating successful termination), inftau (designating an infinite execution that eventually stops performing any visible events), or oom (designating an execution that ends because it runs out of memory). The traces of a program p are given as follows:

$$\begin{aligned}\mathsf{traces}(p, args) \stackrel{\text{def}}{=}\ & \{\ell \cdot \mathsf{end} \mid \exists s \in \mathsf{init}(p, args).\ \exists s'.\ s \stackrel{\ell}{\Rightarrow} s' \not\rightarrow\} \\ \cup\ & \{\ell \cdot \mathsf{inftau} \mid \exists s \in \mathsf{init}(p, args).\ \exists s'.\ s \stackrel{\ell}{\Rightarrow} s' \stackrel{\tau}{\rightarrow}^{\omega}\} \\ \cup\ & \{\ell \cdot tr \mid \exists s \in \mathsf{init}(p, args).\ \exists s'.\ s \stackrel{\ell}{\Rightarrow} s' \xrightarrow{\texttt{fail}}\} \\ \cup\ & \{\ell \cdot \mathsf{oom} \mid \exists s \in \mathsf{init}(p, args).\ \exists s'.\ s \stackrel{\ell}{\Rightarrow} s'\} \\ \cup\ & \{l \mid \exists s \in \mathsf{init}(p, args).\ s \stackrel{l}{\Rightarrow} \text{ and } l \text{ is infinite}\}\end{aligned}$$

Here $\mathsf{init}(p, args)$ denotes the initial states for a program p when called with command-line arguments $args$; for a finite sequence ℓ of non-`fail` visible events, we define $s \stackrel{\ell}{\Rightarrow} s'$ to hold whenever s can do the sequence ℓ of events, possibly interleaved with a finite number of τ-events, and end in state s'; and for a finite or infinite sequence l of non-`fail` visible events, we define $s \stackrel{l}{\Rightarrow}$ to hold whenever s can do the sequence l of events, possibly interleaved with τ-events.

We treat failed computations as having arbitrary behaviour after their failure point, whereas we allow the program to run out of memory at any point during its execution. This perhaps counter-intuitive semantics of oom is needed to express a correctness statement guaranteeing nothing about computations that run out of memory.

Our top-level correctness statement for a compiler compile from ClightTSO to x86-TSO, modelled as a partial function, will then be a trace inclusion for programs for which compilation succeeds, of the form

$$\forall p, args.\ \mathsf{defined}(\mathsf{compile}(p)) \implies \mathsf{traces}_{\text{x86-TSO}}(\mathsf{compile}(p), args) \subseteq \mathsf{traces}_{\text{ClightTSO}}(p, args).$$

4.2. The CompCert 1.5 proof strategy

ClightTSO is an extension of sequential Clight, and its compiler has to deal with everything that a Clight compiler does, except for any optimisations that become unsound in the concurrent setting. We therefore arrange our semantic definitions and proof structure to re-use as much as possible of the CompCert development for sequential Clight, isolating the parts where relaxed-memory concurrency plays a key role.

Our starting point was CompCert 1.5, comprising around 55K lines of Coq subdivided into 13 compiler phases, each of which builds a semantic preservation proof between semantically defined intermediate languages. The overall strategy is prove trace inclusions by establishing simulation results — more particularly, to build some kind of "downward" simulation for each phase, showing that transitions of a source program for the phase can be matched by transitions of the compiled target program; these can be composed together and combined with determinacy for the target language (there PowerPC or ARM assembly) to give an upward simulation for a complete compilation, showing that any behaviour of a compiled program is allowed by the source program

semantics.[2] Downward simulations are generally easier to establish than upward simulations because compiler phases tend to introduce intermediate states; a downward simulation proof does not have to characterise and relate these.

As we shall see, this strategy cannot be used directly for compilation of concurrent ClightTSO to x86, but much can be adapted.

4.3. Decomposing the proof by compiler phases

Our compiler is divided into similar (but not identical) phases to CompCert. For each phase, we define the semantics of a whole program to be an LTS as above, and inclusion of the above notion of traces also serves as the correctness criterion for each of our phases. The individual correctness results can be composed simply by transitivity of set inclusion.

4.4. Labellisation and threadwise proof

In our concurrent setting the languages are not deterministic, so the CompCert approach to building upward simulations is not applicable. However, for most of the phases we can re-use the CompCert proof, more-or-less adapted, to give downward simulation results for the behaviour of a single thread in isolation — and we can make our semantics deterministic for such. We therefore 'labellise' the semantics for each level (source, target, and each intermediate language). Instead of defining transitions

$$(s, m_{\mathrm{SC}}) \longrightarrow (s', m'_{\mathrm{SC}})$$

over configurations that combine a single-threaded program state s and an SC memory m_{SC} (as most sequential language semantic definitions, including CompCert, do), we define the semantics of a single thread (split apart from the memory) as a transition system:

$$s \xrightarrow{te} s'$$

(together with extra structure for thread creation) where a thread event te is either an external event, as above, an interaction with memory me, an internal τ action, the start or exit of the thread, or an out-of-memory error `oom`:

$$\mathit{thread\_event},\ te ::= \ \mathtt{ext}\ ev \mid \mathtt{mem}\ me \mid \tau \mid \mathtt{start}\ \mathit{opt\_tid}\ p\ vs \mid \mathtt{exit} \mid \mathtt{oom}$$

The whole-system semantics of each level is a parallel composition roughly of the form

$$s_1 \mid \ldots \mid s_n \mid m_{\mathrm{TSO}}$$

of the thread states s_i and a TSO machine m_{TSO}. The threads interact with the TSO machine by synchronising on various events: reads or writes of a pointer p with a value v of a specified $\mathit{memory\_chunk}$ size, allocations and frees of a memory block at a pointer p, various error cases, and thread creation. These transitions are in the style of the 'early' transition system for value-passing CCS [Mil89]: a thread doing a memory read will have a transition for each possible value of the right type. For example, here is the

[2] Terminology: in CompCert "forward simulation" and "backward simulation" refer to the direction of the simulation with respect to the compiler phases, thinking of compilation as "forwards". This clashes with another standard usage in which "forward" and "backward" refer to the direction of transitions. In this paper we need to discuss both, so we use "downwards" (and conversely "upwards") to refer to the direction of compilation, reserving "forwards" and "backwards" for the direction of transitions. (Notwithstanding this, the CompCertTSO sources retain the CompCert usage.)

ClightTSO rule for dereferencing a pointer:

$$\frac{\begin{array}{l}\texttt{access\_mode}\ ty' = \texttt{By\_value}\ c \\ typ = \texttt{type\_of\_chunk}\ c \\ \texttt{Val.has\_type}\ v\ typ\end{array}}{p \cdot [*\_^{ty'}] \cdot \kappa_{\mathrm{e}}\,|_\rho \xrightarrow{\texttt{mem (read}\ p\ c\ v\texttt{)}} v \cdot \kappa_{\mathrm{e}}\,|_\rho}\ \textsc{LoadByValue}$$

The conclusion has a start state with a pointer value p in an expression continuation $[*\_^{ty'}] \cdot \kappa_{\mathrm{e}}$ headed by a dereference at a ClightTSO type ty'. The first premise finds the access mode of that type: here it must be accessed by value and has a chunk c (specifying int/float, size, and signedness). The second premise collapses this onto an internal type typ (just int/float, because internal values do not record their size or signedness). The third premise allows an arbitrary value v that of type typ. Then the conclusion has a transition labelled with a memory read, at pointer p, of that value v, as a chunk c, to a state with v in the remaining continuation. (There is a further subtlety here. One might think that the rule should also check that v represents a value of type ty', not just that it has internal type typ. That check could be added here, but in fact we have it in the TSO machine. The premises do suffice to ensure a receptiveness property.)

External events of the threads (and of the TSO machine) are exposed directly as the whole-system behaviour.

This conceptually simple change to a labellised semantics separates concerns: compiler phases that do not substantially affect the memory accesses of the program can be proved correct per-thread, as described in §5.5 (and those results lifted to the whole system by a general result below), leaving only the two remaining main phases and three fence optimisation phases that require proofs that really involve the TSO machine.

4.5. The TSO machine

Our TSO machine is based on the x86-TSO abstract machine, with a main memory and per-thread buffers, but with several differences. The TSO machine must handle memory allocations and frees (which are buffered), and various memory errors; the main memory records allocation as in CompCert. We use the TSO semantics for software threads, not hardware threads, which is sound provided that the scheduler flushes the buffer during task switching. We use the same TSO machine for all the intermediate languages, and we uniformly lift threadwise LTSs to the parallel composition with the TSO machine.

4.6. Establishing whole-system trace inclusions from threadwise downward simulations

For the phases that do not substantially change memory accesses, we establish whole-system trace inclusions from threadwise downward simulations in three steps. First, we observe that a downward simulation from a receptive language to a determinate language implies the existence of upward simulation and use this to obtain threadwise upward simulation. Then we lift the threadwise upward simulation to a whole-system upward simulation. Finally, we establish trace inclusion from the whole-system upward simulation.

We say that two labels are of the same kind, written $te \asymp te'$ if they only differ in input values. In our case, $te \asymp te'$ if (i) te and te' are reads from the same memory location (but not necessarily with the same value), or (ii) te and te' are external returns, or (iii) $te = te'$.

Definition 4.1. A thread LTS is *receptive* if $s \xrightarrow{te} t$ and $te' \asymp te$ implies $\exists t'.\ s \xrightarrow{te'} t'$.

Definition 4.2. A thread LTS is *determinate* if $s \xrightarrow{te} t$ and $s \xrightarrow{te'} t'$ implies $te \asymp te'$ and, moreover, if $te = te'$, then $t = t'$.

Definition 4.3. A relation R between the states of two thread LTSs S and T is a *threadwise downward simulation* if there is a well-founded order $<$ on the states of S such that if given any $s, s' \in S$, $t \in T$ and label te, whenever $s \xrightarrow{te} s'$ and $s \; R \; t$, then either

(1) $te = \texttt{fail}$, or
(2) $\exists t'.\ t \xrightarrow{\tau}{}^{*} \xrightarrow{te} \xrightarrow{\tau}{}^{*} t' \wedge s' \; R \; t'$, or
(3) $te = \tau \wedge s' \; R \; t \wedge s' < s$.

Definition 4.4. A relation R is a *threadwise upward simulation* if there is a well-founded order $<$ on T such that whenever $t \xrightarrow{te} t'$ and $s \; R \; t$, then either

(1) $\exists s'.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{te} s' \wedge s' \; R \; t'$, or
(2) $\exists s'.\ s \xrightarrow{\texttt{fail}} s'$, or
(3) $te = \tau \wedge s \; R \; t' \wedge t' < t$.

Moreover, if $t \not\rightarrow$ (t is stuck) and $s \; R \; t$, then $s \not\rightarrow$ or $\exists s'.\ s \xrightarrow{\texttt{fail}} s'$.

Note the subtle asymmetry in handling errors: if a source state does an error or gets stuck, both the upward simulation and downward simulation hold. In contrast, the target states' errors must be reflected in the source to make the upward simulation hold. This is necessary to allow compilers to eliminate errors but not to introduce them.

THEOREM 4.5. *If R is a threadwise downward simulation from S to T, S is receptive, and T is determinate, then there is a threadwise upward simulation that contains R.* [Coq proof]

Eliding details of initialisation and assumptions on global environments, we have:

Definition 4.6. A relation $R : \mathsf{States}(S) \times \mathsf{States}(T)$, equipped with a well-founded order $<$ on $\mathsf{States}(T)$, is a *measured upward simulation* if, whenever $s \; R \; t$ and $t \xrightarrow{ev} t'$, then either

(1) $\exists s'.\ s \xrightarrow{\tau}{}^{*} s' \xrightarrow{\texttt{fail}}$ (s can reach a semantic error), or
(2) $\exists s'.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{ev} s' \wedge s' \; R \; t'$ (s can do a matching step), or
(3) $ev = \tau \wedge t' < t \wedge s \; R \; t'$ (t stuttered, with a decreasing measure).

THEOREM 4.7. *A threadwise upward simulation can be lifted to a whole-system measured upward simulation, for the composition of the threads with the TSO machine.* [Coq proof]

THEOREM 4.8. *A whole-system upward simulation implies trace inclusion.* [Coq proof]

To establish correctness of compiler phases that remove dead variable loads and concretise undefined values, we have also proved variants of Theorems 4.5 and 4.7 for suitably modified Definitions 4.3 and 4.4. Here, we only describe the concretising simulations.

4.6.1. Concretising Simulations. The simulation statement for phases that concretise undefined values is interesting because we capture the concretisation in the simulation statement, whereas CompCert's approach is to define an auxiliary relation on states

and memories that expresses the less-defined property for all values in the states and memory. One example of concretisation of values is the compilation of function entry in the reloading phase: in LTLin, all registers are set to `Vundef` upon function entry, but in Linear, they keep their original value.

Such a refinement cannot break the overall upward simulation: if the LTLin code used the `Vundef` value in any interesting way it would get stuck and the simulation would be trivially satisfied. It is important to note that even though the compiler concretises only values in registers, the values in registers can be written to memory and spread through the entire system state. The overall simulation relation has to account for this and allow arbitrary parts of the state to be more concrete.

When constructing the whole-system upward simulation, there are two cases of interaction with memory: if the target transition system gets a value from memory, then the source system must be able to accept a less concrete value, because the memory may contain a less concrete value. In contrast, if the target system writes to memory, the source system may want to write a less concrete value. To make concretisation in input and output explicit, we introduce two relations on events, written $\leq_{\mathrm{in}}$ and $<_{\mathrm{out}}$. We prove the downward-to-upward simulation theorem abstractly, requiring only that the $\leq_{\mathrm{in}}$ and $<_{\mathrm{out}}$ relations satisfy the following properties: (i) uniqueness of the same-kind relation for more output-concrete values, i.e., $l_1 \asymp l_2$ and $l <_{\mathrm{out}} l_2$ implies $l_1 = l_2$, (ii) τ is not more output-concrete than any other action, i.e., for all l it is not true that $l <_{\mathrm{out}} \tau$, (iii) the less-concrete-input relation is reflexive, i.e., for all l, $l \leq_{\mathrm{in}} l$.

Our concretising threadwise upward simulation handles the output and input cases separately. We require that any output action transition can be simulated by *some* less concrete output action, and any input action transition can be simulated by *all* less concrete input actions:

Definition 4.9. A relation R is a *concretising threadwise upward simulation* if there is a well-founded order $<_{\mathrm{T}}$ on T such that whenever $t \xrightarrow{te} t'$ and $s\ R\ t$, then either

(1) $\exists s'\ te'.\ te' <_{\mathrm{out}} te \wedge s \xrightarrow{\tau}{}^{*} \xrightarrow{te'} s' \wedge s'\ R\ t'$, or
(2) $\forall te''.\ te'' \leq_{\mathrm{in}} te \rightarrow \exists s'.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{te} s' \wedge s'\ R\ t'$, or
(3) $\exists s'.\ s \xrightarrow{\mathtt{fail}} s'$, or $te = \tau \wedge s\ R\ t' \wedge t' <_{\mathrm{T}} t$.

Moreover, if $t \not\rightarrow$ (t is stuck) and $s\ R\ t$, then $s \not\rightarrow$ or $\exists s'.\ s \xrightarrow{\mathtt{fail}} s'$.

The concretising downward simulation unsurprisingly requires that output actions are simulated by more concrete output actions. The input action simulation is more intricate — to prove the upward simulation from the downward simulation we need to ensure that the source LTS can accept any less concrete value for any input action:

Definition 4.10. A relation R between the states of two thread LTSs S and T is a *concretising threadwise downward simulation* if there is a well-founded order $<_{\mathrm{S}}$ on the states of S such that if given any $s, s' \in S$, $t \in T$ and label te, whenever $s \xrightarrow{te} s'$ and $s\ R\ t$, then either

(1) $te = \mathtt{fail}$, or
(2) $\exists t'\ te'.\ te <_{\mathrm{out}} te' \wedge t \xrightarrow{\tau}{}^{*} \xrightarrow{te'} \xrightarrow{\tau}{}^{*} t' \wedge s'\ R\ t'$, or
(3) $\exists t'.\ t \xrightarrow{\tau}{}^{*} \xrightarrow{te} \xrightarrow{\tau}{}^{*} t' \wedge \forall te''.\ te'' \leq_{\mathrm{in}} te \rightarrow \exists s''. s \xrightarrow{te''} s'' \wedge s''\ R\ t'$, or
(4) $te = \tau \wedge s'\ R\ t \wedge s' <_{\mathrm{S}} s$.

Using these definitions, we establish concretising versions of Theorems 4.5 and 4.7.

For our concrete case of thread events, $te \leq_{\mathrm{in}} te'$ iff $te = te'$ or te and te' read the same chunk at the same location and the value of te is less concrete than the value of

te'. We define $te <_{\text{out}} te'$ to hold iff te is a write of a less concrete value from the same chunk at the same location as te' (in reality, one also has to define $\leq_{\text{in}}$ and $<_{\text{out}}$ for read-modify-write handling).

4.7. Establishing whole-system trace inclusions for the two phases that substantially change memory accesses

In ClightTSO (as in Clight) local variables are all in allocated blocks, but an early phase of the compiler identifies the variables whose addresses are not taken (by any use of the `&` operator) and keeps them in thread-local environments, changing loads and stores into (τ-action) environment accesses; moreover, individual stack allocations on function entry are merged into one large allocation of the entire stack frame. Conversely, a later phase does activation record layout, and thread-local state manipulation (τ actions) is compiled into memory accesses to the thread-local part of activation records. In both cases, the thread has different views of memory in source and target, and these views involve the TSO-machine buffering of loads, stores, allocations and frees. We return to this, which is the heart of our proof, in §5.3 and §5.4.

4.8. Establishing whole-system trace inclusions for the three phases that change fences

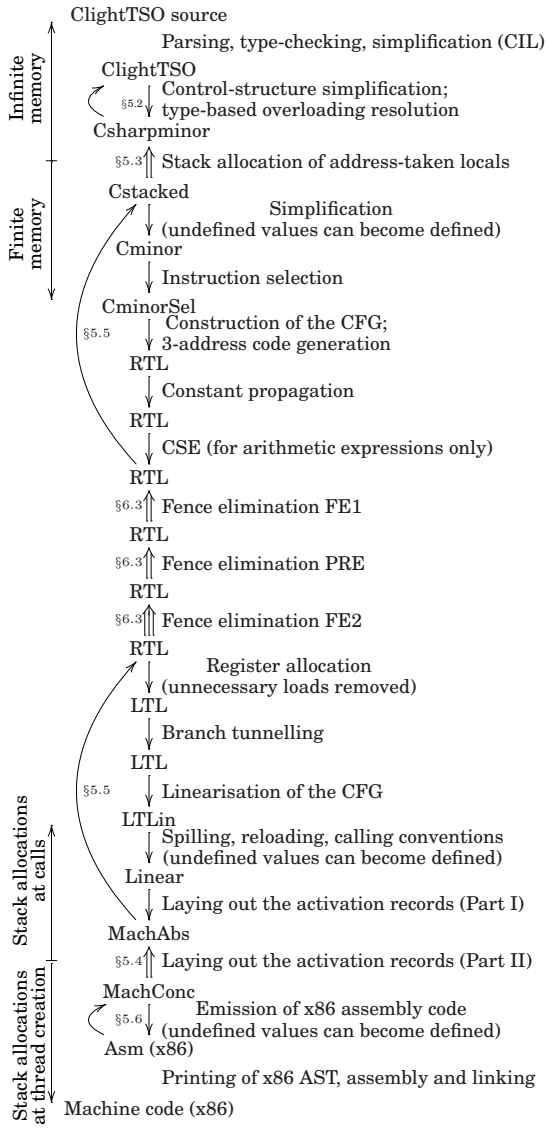
Our compiler contains one phase that inserts memory fences at appropriate program points and two phases that remove redundant memory fences: one where the removed fences have a trivial effect as the buffer is empty when the fences are executed, and one where the effect of the fence is never observed by the program. The correctness of the first two of these transformations is straightforward and shown using a whole-system upward simulation. The correctness of the third transformation is much subtler and requires a new form of whole-system upwards simulation, which we call a weak-tau simulation. We return to this in §6.

4.9. Finite memory revisited

To be faithful to a real machine semantics, our x86 semantics uses finite memory and performs memory allocations only when threads are initialized (the stack of the thread is allocated). In Clight, however, small memory allocations happen whenever a variable is declared; as a result, the memory should be unbounded because the compiler can promote local variables to registers and thus a Clight program can have a footprint that would not fit in the x86 memory. In our intermediate languages, we switch from infinite to finite memory in the Csharpminor to Cstacked phase (§5.3), where we move local variables whose address is not taken to local environments, and perform one allocation (for the remaining local variables) per function call. Since our pointer type needs to accommodate both the finite and infinite nature of addresses, our pointers are composed of two parts: an unbounded block identifier and machine integer offset within the block. The lower-level language semantics uses only the finite memory in block 0—the memory refuses to allocate any other block. The higher level languages can allocate in any block. Note that one memory block can contain more than one memory object. A later phase (MachAbs to MachConc, §5.4) compiles away the allocations per function call, pre-allocating a thread's stack when it is created.

4.10. The final phase: targetting x86

We target x86 because x86-TSO gives us a relatively simple and well-understood relaxed memory model for a common multiprocessor. CompCert 1.5 targets sequential PowerPC and ARM assembly language, but these have much more intricate concurrent behaviour [SSA$^+$11; AMSS10]). We therefore implemented an x86 backend, described in §5.6, adopting parts of the new x86 backend of CompCert 1.8 but with a different instruction semantics.



Our proof structure is indicated by single arrows for threadwise downward simulations (lifted to upward trace inclusions, shown with upwards single arrows, by Theorems 4.5, 4.7 and 4.8); straight double arrows for direct proofs of whole-system upward simulations; and a straight triple arrow for a direct proof of a whole-system upward weak-tau simulation.

ClightTSO and Csharpminor perform a stack allocation for each individual variable in the program and assume an infinite memory, whereas the languages below have only finite memory. From Cstacked to MachAbs a stack allocation occurs for each non-empty stack frame (that is, almost every function call), whereas in MachConc and Asm only when a thread is created.

Fig. 3. CompCertTSO phases

5. COMPCERTTSO

Following the strategy above, we have built a working verified compiler from ClightTSO to x86 assembly language with x86-TSO semantics. This shows (a) how we can reason about concurrent TSO behaviour, in the phases where that plays a key role, and (b) how our overall strategy enables relatively straightforward adaptation of the existing sequential proof, in the phases where concurrent memory accesses do not have a big impact.

The structure of our compiler, and of its proof, is shown in Fig. 3. The subdivision into phases between intermediate languages follows CompCert 1.5 as far as possible, with our major changes being:

— The source and target languages are ClightTSO and concurrent x86 assembly, not Clight and PowerPC or ARM assembly.
— The semantics is expressed with a TSO machine, which is common to all phases.
— We need a stack of memory-model-aware abstractions for the intermediate languages. While named after those of CompCert, their semantics are all adapted to labellised TSO semantics.
— The simulation from ClightTSO to the first intermediate language, Csharpminor, is a new proof above our small-step semantics.
— The CompCert phase that does stack allocation of some local variables (those whose address is taken by `&`), from Csharpminor to Cminor, is divided into two via a new intermediate language Cstacked. Cstacked has the same syntax as Csharpminor (and compilation to it is the identity on terms) but a memory semantics more like Cminor. The proof of the Csharpminor-to-Cstacked phase is a new direct whole-system upward simulation argument, dealing with the very different patterns of memory accesses in the two languages and how they interact with the TSO machine.
— The proofs of the previous middle phases of the compiler, from RTL to MachAbs with various optimisations, are relatively straightforward adaptations of the CompCert proofs to our per-thread labellised semantics and then lifted by the general results of the previous section.
— The fence elimination phases are new.
— Our Mach-to-Asm phase generates x86 rather than PowerPC or ARM assembly.

The rest of this section discusses these in more detail except for the fence elimination optimisations, which are deferred to Section 6. To give a flavour of the actual Coq development we switch presentation style, quoting small excerpts of the Coq source rather than hand-typesetting. Our main result is as follows.

THEOREM 5.1 (COMPILER CORRECTNESS).

```
forall fe1 fi2 fe2 p p',
  transf_c_program false fe1 fi2 fe2 p = OK p' ->
  forall args trace,
    valid_args args ->
    prog_traces Asm.x86_sem p' args trace ->
    prog_traces Csem.Clight.cl_sem p args trace.
```

[Coq proof]

Here `transf_c_program` is the compiler, `p` ranges over ClightTSO programs, `p'` ranges over x86 programs, `args` ranges over command-line arguments, and `trace` ranges over traces. The first four arguments of `transf_c_program` control whether fence insertion and fence elimination are performed, as described in §6.

Proof outline: First, we construct threadwise downward simulations from ClightTSO to Csharpminor, between each of the non-fence-elimination phases from Cminor to MachAbs, and from MachConc to Asm. Then, we turn these threadwise downward simulations to threadwise upward simulations by Theorem 4.5 (and by the analogous theorems for the concretising threadwise downward simulation and for the lock-step threadwise downward simulation with unnecessary load removal). Then, by Theorem 4.7, we turn the threadwise upward simulations into whole-system measured upward simulations. In §5.3 and §5.4, we also establish measured upward simulations from Cstacked to Csharpminor and from MachAbs to MachConc. In §6, we also establish measured upward simulations for the first two fence elimination phases and an upward weak-tau simulation for the third fence elimination phase. By Theorem 4.8 (and by the analogous theorem for weak-tau simulations), we deduce that the traces of the output program of each phase are included in those of its input program. Finally, by transitivity of trace inclusion, we get the end-to-end trace inclusion.

5.1. TSO machine design and interaction with threads

To separate the sequential language semantics from the memory model, we split the whole-system semantics in two parts: the thread transition systems indexed by thread identifiers, and the TSO transition system with a state consisting of the main memory (essentially an array of values) and buffers, represented as thread-id-indexed lists of buffered events. The buffered events can be writes, allocations, or frees:

```
Inductive buffer_item :=
  | BufferedWrite (p: pointer) (c: memory_chunk) (v: val)
  | BufferedAlloc (p: pointer) (i: int) (k: mobject_kind)
  | BufferedFree  (p: pointer) (k: mobject_kind).
```

Note that all the transition systems have different labels: the whole system labels are $event$s (§4.1), the threads' labels are $thread_event$s (§4.4) and the TSO machine labels are tso_events:

```
Inductive tso_event :=
  | TSOmem       (tid: thread_id) (m: mem_event)
  | TSOreadfail  (tid: thread_id) (p: pointer) (c: memory_chunk)
  | TSOfreefail  (tid: thread_id) (p: pointer) (k: mobject_kind)
  | TSOoutofmem  (tid: thread_id) (i: int) (k: mobject_kind)
  | TSOstart     (tid: thread_id) (newtid: thread_id)
  | TSOexit      (tid: thread_id)
  | TSOtau.
```

where the memory events mem_event are:

```
Inductive mem_event :=
  | MEfence
  | MEwrite (p: pointer) (chunk: memory_chunk) (v: val)
  | MEread  (p: pointer) (chunk: memory_chunk) (v: val)
  | MErmw   (p: pointer) (chunk: memory_chunk) (v: val) (instr: rmw_instr)
  | MEalloc (p: pointer) (size: int) (k: mobject_kind)
  | MEfree  (p: pointer) (k: mobject_kind).
```

The TSO machine differs from our original *x86-TSO* [SSO+10; OSS09] semantics described in Section 2 by adding error handling, allocation and free, thread creation, and exit, and by replacing machine lock and unlock transitions by explicit read-modify-write transitions.

Ideally, the TSO transition system would synchronise with the thread transition systems on memory events and thread management events (producing a whole-system τ transition), and all the remaining events of the threads and the TSO machine would be exposed as whole-system transitions. Unfortunately, there are several cases where we need a more fine-grained approach because of error-handling; for example, if a thread issues a read, the TSO machine can either successfully read a value (using the `TSOmem` t (`MEread` ...) event), or it can fail because the memory is not allocated (with the `TSOreadfail` event). We should note that we handle out-of-memory events separately because we aim to separate programmer errors, such as memory safety violations, from a possibly inefficient allocator that produces excessively fragmented memory.

For full details of the TSO machine transition system, see Figures 4 and 5. The TSO machine handles the successful cases of memory operations using rules TSO-READ, TSO-WRITE, TSO-ALLOC, TSO-FREE, TSO-FENCE and TSO-RMW. The read rule obtains the value from its current view of memory, i.e., the main memory with the reading thread's buffer applied. The fence and read-modify-write rules require the buffer of the thread to be flushed. For the other memory rules, the TSO machine appends the memory operation to the thread's buffer. When inserting to memory buffers, we always make sure that all possible interleavings of applying buffers to memory would succeed. In particular, it is important to guarantee that all allocations in buffers are fresh after any unbuffering. It might seem that it would be sufficient to check freshness when inserting the allocation. However, when inserting a free event into a buffer, we might free memory that has a pending allocation if we unbuffer the free event before the allocation event. To avoid these corner cases, we simply require that no insertion to buffers can introduce errors when unbuffering, and fail eagerly if there is a potential error.

The TSO machine handles write, read and read-modify-write errors using the TSO-READ-FAIL rule. For simplicity, the rule can only be applied with an empty buffer, but it is easy to establish (and we have a Coq proof) that this is weakly bisimilar to the more permissive alternative, where writes fail if insertion into a buffer would cause an error after some unbuffering, and reads fail if the memory being read is not allocated in the TSO machine's memory with the thread's buffer applied. Similarly, TSO-FREE-FAIL can only fail with an empty buffer, but to have the bisimilarity we further insist that writes in other buffers can be successfully performed after the deallocation. Finally, there are unsurprising steps for applying the head of a buffer (TSO-UNBUFFER), adding and removing threads (TSO-START, TSO-EXIT) and out-of-memory handling (TSO-OUT-OF-MEMORY).

The whole-system transition system mostly synchronises corresponding transitions of threads and the TSO machine, but there are several exceptions to this scheme. The rule handling thread start creates a new thread and initialises the thread with a function identified by the name in the spawning thread's start event. If there is no function of the required name there is a start error transition rule. The rule for thread stuckness fires an error transition if there is a thread that cannot make any progress. This can only happen if there is a run-time error, such as multiplication of two pointers. We should note that there are two sets of rules for thread start and external action because their argument passing is different in the back end of the compiler (MachConc, Asm) and the front/middle end of the compiler (Clight to MachAbs). Languages between Clight and MachAbs pass arguments to an external (or thread start) function in the external/thread-start event and no memory is involved. In contrast, the MachConc and Asm languages use the approach mandated by the calling conventions: the arguments are passed on stack in memory, the external and thread start thread-events take the addresses of their arguments' locations, and the whole-system transition is

```
Inductive tso_step : tso_state -> tso_event -> tso_state -> Prop :=

  (* MEMORY OPERATIONS *)
  | tso_step_write : (* Memory write  (goes into buffer) *)
    forall t ts ts' p c v
      (EQts': ts' = buffer_insert ts t (BufferedWrite p c v))
      (SAFE: unbuffer_safe ts'),
    tso_step ts (TSOmem t (MEwrite p c v)) ts'

  | tso_step_read :   (* Memory read *)
    forall ts t m' p c v
      (AB: apply_buffer (ts.(tso_buffers) t) ts.(tso_mem) = Some m')
      (LD: load_ptr c m' p = Some v),
    tso_step ts (TSOmem t (MEread p c v)) ts

  | tso_step_read_fail: (* Memory read failure *)
    forall ts t p c
      (Bemp: ts.(tso_buffers) t = nil)
      (LD: load_ptr c ts.(tso_mem) p = None),
    tso_step ts (TSOreadfail t p c) ts

  | tso_step_alloc :  (* Memory allocation  (goes into buffer) *)
    forall t ts ts' p i k
      (EQts': ts' = buffer_insert ts t (BufferedAlloc p i k))
      (UNS: unbuffer_safe ts'),
    tso_step ts (TSOmem t (MEalloc p i k)) ts'

  | tso_step_free :  (* Memory deallocation (goes into buffer) *)
    forall t ts ts' p k
      (EQts': ts' = buffer_insert ts t (BufferedFree p k))
      (UNS: unbuffer_safe ts'),
    tso_step ts (TSOmem t (MEfree p k)) ts'

  | tso_step_free_fail : (* Memory deallocation fail *)
    forall t ts p k
      (Bemp: ts.(tso_buffers) t = nil)
      (FAIL: match free_ptr p k (tso_mem ts) with
           | None => True
           | Some m' => exists tid', exists p, exists c, exists v, exists b,
                           tso_buffers ts tid' = BufferedWrite p c v :: b
                        /\ store_ptr c m' p v = None
           end),
    tso_step ts (TSOfreefail t p k) ts

  | tso_step_outofmem :
     forall t ts n k
       (OOM: forall p,
               ~ unbuffer_safe (buffer_insert ts t (BufferedAlloc p n k))),
    tso_step ts (TSOoutofmem t n k) ts
```

Fig. 4. TSO machine transition system: Part 1

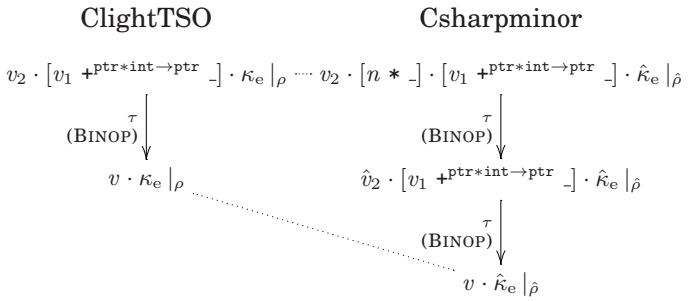
responsible for reading out the arguments from the TSO memory before emitting a whole-system external event (or spawning a thread).

5.2. Small-stepping (ClightTSO to Csharpminor)

ClightTSO is compiled into Csharpminor, a high-level intermediate representation that has a simpler form of expressions and statements. Most notably, the translation unifies various looping constructs found in the source, compiles away casts, translates union and structs into primitive indexed memory accesses, and makes variable l-value and r-value distinctions explicit. High-level type information found in ClightTSO is

```
(* UNBUFFERING *)
| tso_step_unbuffer : (* Apply buffer item *)
  forall t ts bufs' bi b m'
    (EQbufs: ts.(tso_buffers) t = bi :: b)
    (EQbufs': bufs' = tupdate t b ts.(tso_buffers))
    (AB: apply_buffer_item bi ts.(tso_mem) = Some m'),
  tso_step ts (TSOtau) (mktsostate bufs' m')

(* ATOMIC INSTRUCTIONS *)
| tso_step_mfence : (* Mfence (note that the buffer must be flushed) *)
  forall ts t
    (Bemp: ts.(tso_buffers) t = nil),
  tso_step ts (TSOmem t MEfence) ts

| tso_step_rmw : (* Read-modify-write (note that the buffer must be flushed) *)
  forall ts ts' t p c v instr m'
    (Bemp: ts.(tso_buffers) t = nil)
    (LD: load_ptr c ts.(tso_mem) p = Some v)
    (STO: store_ptr c ts.(tso_mem) p (rmw_instr_semantics instr v) = Some m')
    (EQts': mktsostate ts.(tso_buffers) m' = ts'),
  tso_step ts (TSOmem t (MErmw p c v instr)) ts'

(* THREAD MANAGEMENT *)
| tso_step_start : (* Thread start *)
  forall ts ts' t bufs' newtid
    (EQbufs': bufs' = tupdate newtid nil ts.(tso_buffers))
    (EQts': mktsostate bufs' ts.(tso_mem) = ts'),
  tso_step ts (TSOstart t newtid) ts'

| tso_step_finish : (* Thread finish *)
  forall ts t
    (Bemp: ts.(tso_buffers)  t = nil),
  tso_step ts (TSOexit t) ts
```

Fig. 5. TSO machine transition system: Part 2



Here `int` = `int(I32,Signed)` and `ptr` = `pointer(int)`. The type annotation in the multiplication (*) context is omitted.

Fig. 6. Part of the simulation relating ClightTSO and Csharpminor evaluation for addition of an `int` and a pointer

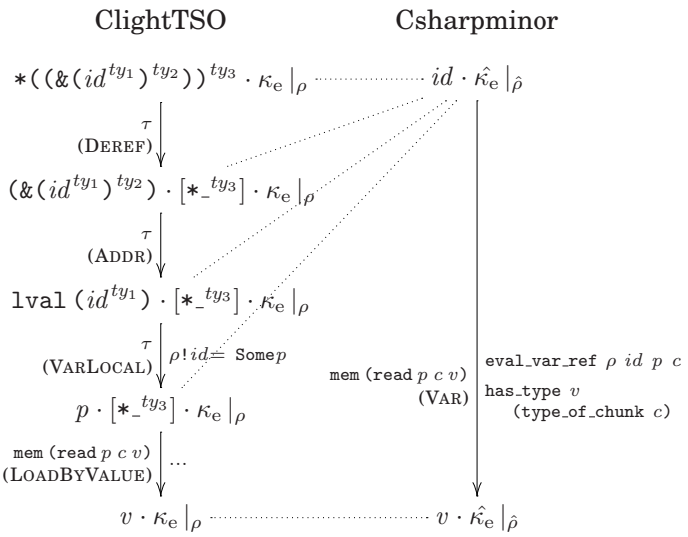


Fig. 7. ClightTSO compilation can sometimes eliminate source-level transitions

compiled to a lower-level byte-aware memory representation. Accounting for these differences in the simulation is complicated by the relatively large size of the two languages: ClightTSO's definition has 94 rules, while Csharpminor has 62.

Because expression evaluation is defined by a small-step semantics, adapting the downward simulation proofs directly from CompCert (which uses a big-step expression evaluation semantics) was not feasible, and much of the proof, along with the simulation change, had to be written from scratch as a result. Since the two languages are relatively close, however, the revised simulation could sometimes simply map ClightTSO transitions directly to the corresponding Csharpminor ones; evaluation of constants, unary operations, and certain components of function call and return are such examples.

However, as we mentioned earlier, compilation often results in a ClightTSO term becoming translated to a sequence of lower-level simpler Csharpminor terms. To illustrate, the diagram shown in Fig. 6 shows the evaluation of a binary addition of an integer and a pointer. For ClightTSO, the multiplication of the integer operand by the representation size of the pointer type is performed implicitly, subsumed within the intrinsic definition of addition. In Csharpminor, an explicit binary multiplication operation is introduced. Notice that the continuations in the subsequent matching states are structurally quite different from each other as a result; the simulation relation must explicitly account for these differences.

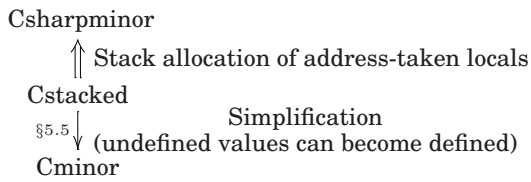
Perhaps a more surprising consequence of using a small-step semantics is that the simulation relation may sometimes be required to match multiple ClightTSO transitions to a single Csharpminor one. For example, compilation from ClightTSO to Csharpminor eliminates various states defined in ClightTSO to deal with addressing and dereferencing. Consider the evaluation of an identifier that appears in an r-value context. In ClightTSO, the identifier is first translated into a pointer, and a separate step returns either the contents of the pointer (in case it references a scalar type) or the pointer itself (in case of e.g., arrays or structs). Compilation to Csharpminor removes this intermediate step, generating the appropriate access instruction directly, since the pointer type is statically known. This simplification generalizes to sequences

of address-of and dereferencing operations. We depict the sequence of steps necessary to compute a variable's address, and then dereference it (if it is a scalar) in Fig. 7. The relation `eval_var_ref` states that variable `id`, in the context of local environment ρ, evaluates to pointer p that references an object with memory representation c. The value v read must have a type consistent with c as defined by relation `has_type`. Notice that ClightTSO requires four steps to perform this operation while compilation to Csharpminor requires only one. To account for such differences, the simulation relation forces Csharpminor transitions to stutter, and we incorporate a measure on ClightTSO expressions and continuations that allows matching of several intermediate ClightTSO states to a single Csharpminor one. Indeed, such a measure, suitably adapted, must be defined for most other compiler phases.

Besides memory read and write operations, the ClightTSO semantics also generates events for function argument and local variable allocation as part of the function calling sequence. The small-step semantics requires these operations be performed in stages. After all argument expressions and the function pointer have been evaluated, memory is allocated for each formal parameter, as well all local variables, in turn. Each distinct allocation is represented as a separate labelled transition. After allocation, the values of the actuals are written to the formals. On function exit, allocated storage is freed individually. The corresponding Csharpminor transitions are similar, albeit with a change in the underlying type representation used to guide memory allocation and writes.

5.3. Changing memory accesses (1) (Csharpminor to Cstacked)

5.3.1. Languages and Compilation. The Csharpminor to Cstacked phase bridges the semantic gap to the next intermediate language, Cminor, by introducing a new semantics for the Csharpminor syntax. That is, the program transformation from Csharpminor to Cstacked is an identity function. However, the Cstacked memory semantics closely follows that of Cminor, which differs radically from Csharpminor.



To understand the motivation for introducing Cstacked, we summarise the main features of the following compilation phase (Cstacked to Cminor):

(1) Local variable reads and writes are turned into explicit memory accesses or local state reads and updates. Note that in Csharpminor, as in C, it is legal to take the address of a local variable and even to pass it to another thread, so long as it is not accessed outside its lifetime. Variables whose address is never taken, however, are guaranteed to be thread-local, and the compiler lifts such variables from memory to local state. The remaining variables are kept in memory.
(2) Individual local variable allocations are replaced with single stack-frame allocation.
(3) Switch-case statements are compiled to switch-table statements.

Without the intermediate Cstacked phase, the first two steps would change memory semantics: Step 1 would replace memory accesses to local variables with local state manipulation that does not touch memory, and Step 2 would replace the individual variable (de)allocations with a single stack-frame (de)allocation in Cminor.

To separate concerns, the Cstacked semantics only captures the memory effects of the transformation, i.e., its transitions simulate the compilation steps 1 and 2. Cstacked and Csharpminor only differ in handling local variables. The change is most evident in the types of local environments, which are part of the local state of threads. In Csharpminor, a local environment is a map from names to pointers and type information that essentially describes the size of a local variable in memory:

$$\mathit{var\_kind},\ \mathit{vk} ::= \texttt{scalar}\ \mathit{memory\_chunk} \mid \texttt{array}\ \mathit{size}$$
$$\mathit{cshm\_env},\ \mathit{cshe} ::= \texttt{nil} \mid (\ \mathit{id} : (\ p\ ,\ \mathit{vk}))::\mathit{cshe}$$

In Cstacked, a local environment consists of a stack frame pointer and a map that assigns to each name a value or an offset within the stack-frame:

$$\mathit{st\_kind},\ \mathit{sk} ::= \texttt{local}\ v \mid \texttt{stack\_scalar}\ \mathit{memory\_chunk}\ \mathit{ofs} \mid \texttt{stack\_array}\ \mathit{size}\ \mathit{ofs}$$
$$\mathit{cst\_items},\ \mathit{csti} ::= \texttt{nil} \mid (\ \mathit{id} : \mathit{sk})::\mathit{csti}$$
$$\mathit{cst\_env},\ \mathit{cste} ::= (\ p\ ,\ \mathit{csti})$$

Note that Cstacked can keep values of local variables in the local environment (when the corresponding $\mathit{st\_kind}$ is `local`). This contrasts with Csharpminor, which stores the values of all local variables in memory.

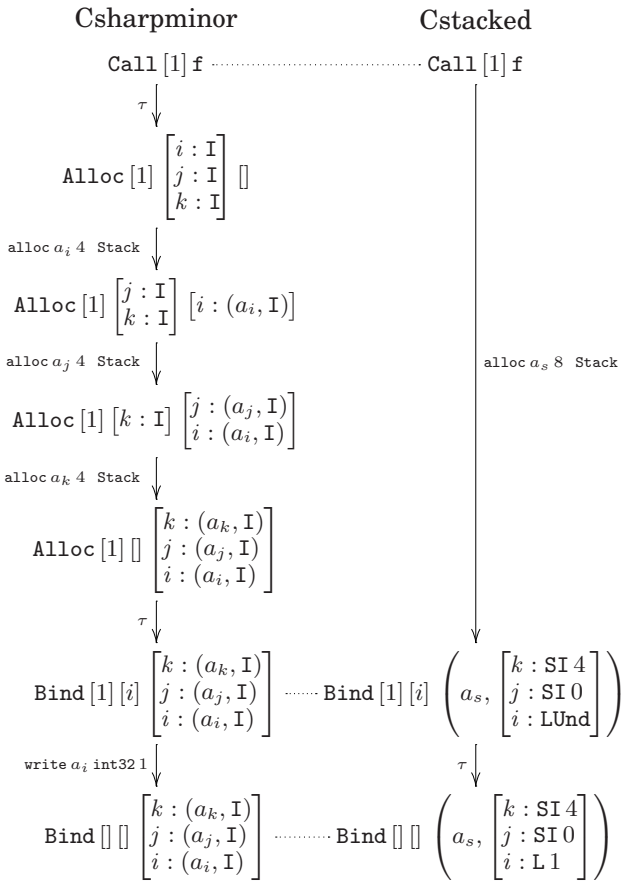
The difference in the environment drives all the other changes from Csharpminor to Cstacked: we adjust the rules for assignment, the write of a function's return value, local variable reads, function entry, and function exit to handle local in-state variables and on-stack variables separately. The most significant change is in function entry, where we scan the function body for the `&` operator and compute the size of its stack frame together with offsets for on-stack local variables.

We illustrate the radical difference between the memory semantics of Csharpminor and Cstacked on the environment construction and parameter binding in function entry. Consider the following function:

```
int f(int i) {int j, k; g(i, &j, &k); return j+k;}
```

Fig. 8 shows the environment construction and argument binding transitions following an invocation of `f` with parameter 1. The states have the following meaning: the state `Call` $l\ f$ follows the evaluation of actual parameters l in the invocation of f; `Alloc` $l\ v\ e$ is an intermediate state for allocation of local variables v, where e is an accumulator for the environment and l is the list of values to be bound to the function's formal parameters; `Bind` $l\ p\ e$ is a state for binding parameter names p to values l in environment e. The `Alloc` to `Bind` transition retrieves the parameter names from the state's continuation, which we omit in this example for brevity. Note that the states do not refer to memory directly. Instead, the transitions expose the memory interaction in the labels. In Csharpminor, the semantics of function entry allocates three different 4-byte blocks, one for parameter `i`, and two for variables `j` and `k`. In Cstacked (and in all languages between Cminor and MachAbstract), the function entry semantics allocates a single 8-byte stack frame for variables `j` and `k`. No memory is reserved for variable `i` because `i`'s value is kept in the thread-local local environment. The binding transitions are also different: Csharpminor writes the value 1 of parameter `i` to memory, but Cstacked simply stores the value in the environment. Indeed, note the difference in the environment entry for i in the last `Bind` states at the bottom of the figure: the Csharpminor entry only contains a pointer to memory, whereas the Cstacked entry contains the value of the variable.

5.3.2. Simulating Cstacked in Csharpminor. Remember that the Csharpminor-Cstacked phase switches from infinite memory to finite memory. This is necessary to be able to simulate the creation of the Cstacked local environments by fresh memory allocation



Here `I` stands for `scalar int32`, `LUnd` for `local Vundef`, `SI` *ofs* for `stack_scalar int32` *ofs*, and `L 1` for `local (Vint 1)`.

Fig. 8. Function entry transitions in Csharpminor and Cstacked

in Csharpminor so that the memory cannot be allocated even by future Cstacked allocations. We call the finite space used by Cstacked the *machine space*. The remaining (infinite) part of the Csharpminor memory space in other blocks is called *scratch space*. Our representation of pointers is of the form (b, ofs) where b is an integer block identifier and $ofs \in \{0, \ldots, 2^{32} - 1\}$ is an offset. In our semantics, the machine space pointers have block $b = 0$, the pointers with non-zero b are scratch space pointers. We simulate Cstacked transitions so that we preserve equality of pointer values in the states and the values in the (machine) memory:

— We simulate Cstacked stack frame allocation by allocations of individual variables at the same (machine) memory location as they have in Cstacked. Moreover, we allocate space for Cstacked local environments in globally fresh blocks in the scratch memory.
— Cstacked memory reads/writes are simulated by the same reads/writes in Csharpminor.
— Cstacked local environment accesses (which are τ events in Cstacked) are simulated by memory accesses to the corresponding Csharpminor scratch memory.

— We simulate Cstacked stack frame deallocation by freeing the individual variables, including the ones in non-machine memory, in Csharpminor.

The simulation relation on the states of the parallel composition of threads and the TSO machine consists of three main components: a thread state relation, a TSO buffer relation and a memory relation.

5.3.3. Relating thread states. The main source of difficulty is relating the local environments of Cstacked and Csharpminor, because the values of the local environments in Cstacked correspond to the memory contents of Csharpminor. Therefore, the thread state simulation must relate a Cstacked thread state with a Csharpminor thread state *and* memory.

In our TSO semantics, a thread's view of memory may differ from the real contents of the memory and from other threads' views of memory because of possibly pending writes, allocations and frees in store buffers of this and other threads. We consider local environments related for thread t if the values in the local environments in the Cstacked state are the same as the ones in the memory of Csharpminor's TSO machine with t's buffer applied. Moreover, we consider stack environments related if for each Cstacked environment item of the stack kind with offset ofs, the corresponding Csharpminor item's pointer equals the sum of Cstacked stack frame pointer and ofs. Since Cstacked and Csharpminor only differ in their environments, the thread state simulation relation is a natural lifting of the environment relation.

All thread transitions preserve such a relation because they can only affect the thread's buffer. However, the simulation of applying other threads' buffers to the main memory (unbuffering) requires a stronger relation. In particular, the state relation does not prevent unbuffering in one thread from interfering with another thread's state relation. To get non-interference for unbuffering, we keep track of memory partitioning among threads (this is also necessary to make sure that threads do not free each others' stack frames) by augmenting the state relation with the partitions they own in memory.

5.3.4. Relating buffers. The buffer relation requires that a Cstacked (stack-frame) allocation corresponds to individual disjoint Csharpminor allocations (of individual variables) that must be in the stack-frame; Cstacked writes correspond to the same writes in Csharpminor buffer; and frees in a Cstacked buffer correspond to frees of sub-ranges in Csharpminor. To relate frees, we must know the sizes of objects in memory because a free label does not contain a size; hence, we parametrise the buffer relation by the thread's partition. It is worth noting that the Csharpminor buffer may contain extra memory labels for the local environment manipulation, which are τ labels in Cstacked and thus do not appear in the Cstacked buffer. We only require the operations in the labels to be valid in the thread's partition.

Fig. 9 illustrates the buffer relation. Assuming that the TSO machine inserts labels to the top of the buffer and applies the labels to memory from the bottom, the buffer contents might be generated by the function `f` from the beginning of this section, where the allocations correspond to the transitions from Fig. 8, the dotted part of the buffer is generated by the function `g`, the frees correspond to local variable deallocations at function exit, and the write label is issued by writing the return value to the caller's stack frame. The grey labels are the memory manipulation removed by the compiler, or, more precisely, they are the labels introduced by the upward simulation (note that they act on scratch memory).

In the simulation proof, the buffer relation says how to simulate Cstacked buffer application in Csharpminor while preserving the simulation relation. For example, if

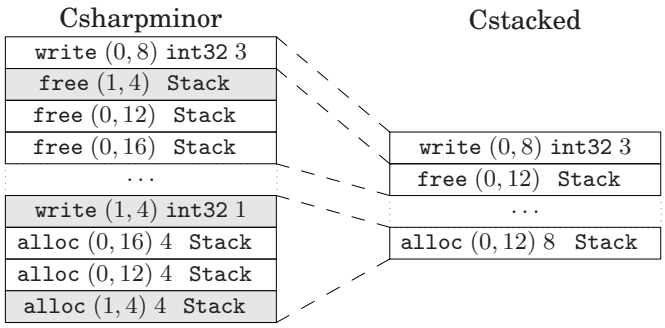


Fig. 9. Buffer relation

we are to simulate Cstacked buffer application of the `alloc` label, we apply the three corresponding allocations followed by the write from the Csharpminor buffer.

5.3.5. Relating TSO states. The whole-system simulation relation states that there are Cstacked and Csharpminor partitionings, i.e., maps from thread ids to partitions such that

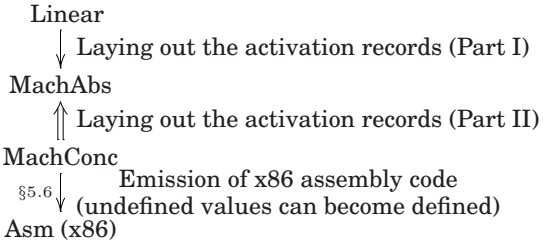
— The Csharpminor (resp. Cstacked) partitioning corresponds to the ranges allocated in the Csharpminor (resp. Cstacked) TSO machine's memory. Moreover, the partitionings must be pairwise disjoint and for each thread, the Csharpminor machine partitions must contain sub-ranges of Cstacked partitions. This is necessary to guarantee that any Cstacked allocation can be successfully simulated in Csharpminor[3].
— The values in the machine memory are the same in Cstacked and in Csharpminor. We need this property to establish that reads of the same address give the same value in Cstacked and in Csharpminor.
— Each thread's Cstacked and Csharpminor buffers are related.
— For each thread t, the states of t in Csharpminor and Cstacked are related in the partitions and memory updated by t's buffers.

The relation also imposes several consistency invariants: to guarantee that Cstacked writes do not overwrite Csharpminor scratch memory, we require that scratch pointers only appear as pointers in Csharpminor environments. With these ingredients, the relation on the TSO states is a whole-system upward simulation relation.

5.4. Changing memory accesses (2) (MachAbs to MachConc)

The overall structure of the simulation proof from MachAbs to MachConc is similar to the Csharpminor-Cstacked correctness proof. MachAbs and MachConc are again two different semantics for the same programs.

[3]A simulation of successful allocation is an interesting (and lengthy) exercise because one must show that in Csharpminor, no possible partial application of other threads' buffers conflicts with the simulated allocations. The partial buffer applications create states that do not directly correspond to any Cstacked state (e.g., partially allocated environments), forcing us to invent a new simulation relation for this purpose.



Both MachAbs's and MachConc's thread states include the processor state, i.e., the state of general purpose registers, the current function, the stack pointer and the instruction pointer. The semantics differ in storage of function frames, which contain (non-escaping) local variables, function arguments, callee-saved register contents and return addresses. The MachAbs semantics stores the frames in its state, and instructions that manipulate the function frames (`getstack`, `setstack`, `getparam`) do not touch memory, i.e., they perform a τ transition, which accesses only the thread-local function frames. In contrast, MachConc stores the function frames in (global) memory; so the three aforementioned instructions generate read or write events for communicating with the TSO machine. It is worth remembering that not all function-local variables are thread-local since a program can take the address of a function-local variable, send it to another thread which can then access it. If a C program takes an address of a function-local variable, the variable is put into memory, called the stack frame memory, by the Csharpminor-Cstacked phase; otherwise, the variable is kept in the thread-local environment. The stack frame management does not change until the MachAbs phase, where stack frames are still allocated in memory upon function entry and deallocated on function exit. In MachConc, stack frames live inside function frames. The function frames are included in the thread's stack space that is allocated upon starting the thread. To avoid confusion, we will refer to the stack frame memory as MachAbs frames and the function frame memory as MachConc frames.

We split the MachAbs-MachConc simulation proof in two parts. First, we prove a form of threadwise upward simulation that keeps track of thread-local parts of memory and is independent of the TSO semantics. Then we show that if we have the threadwise upward simulation, then there is a whole-system upward simulation. This second part of the proof does not refer directly to the thread semantics—it only uses the abstract notion of threadwise simulation. Before we give an overview of the proof, we describe our intermediate threadwise simulation abstraction. We illustrate the concept of threadwise simulation relation on an example program:

```
int f(int x) {
  int i, *p;
  i = x;  p = &i;
  return *p;
}

int main() {
  int r = f(1);
  return r;
}
```

Observe that the variable `i` is part of `f`'s MachAbs frame while `p` is not (because the program does not take the address of `p`). For the purposes of our explanation here, we assume that the MachConc frame of `f` includes both `p` and `i`. The MachConc frame

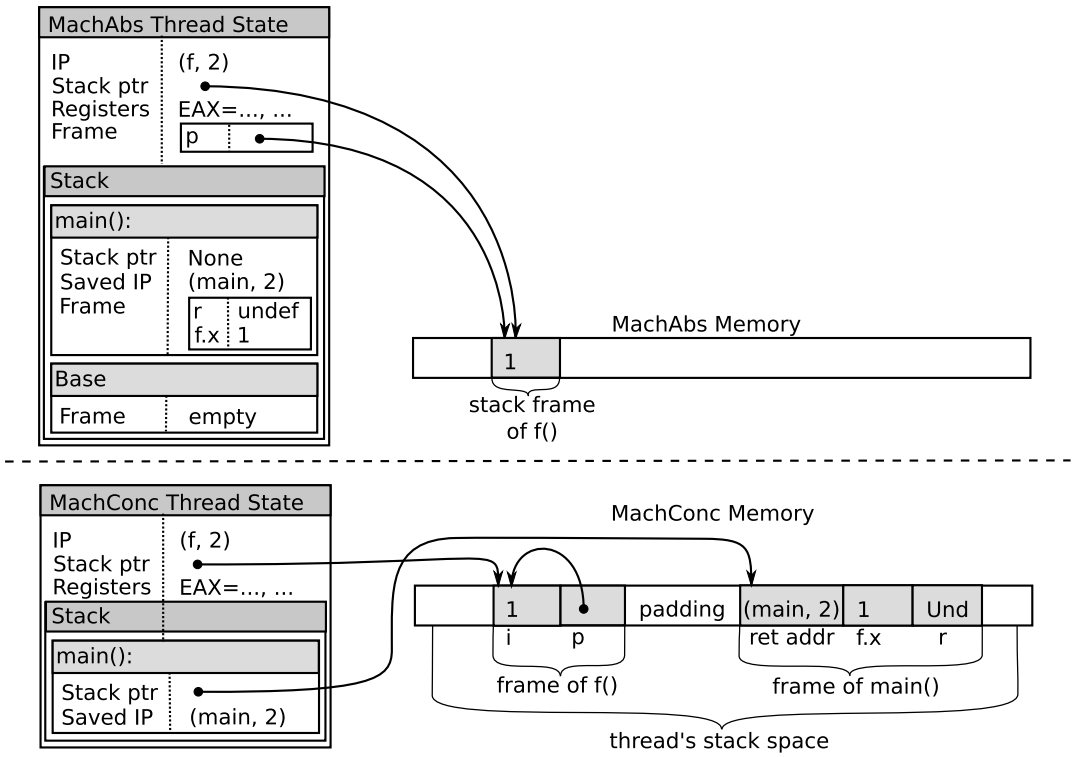


Fig. 10. MachConc and MachAbs thread states

of `main` contains `r`, a place-holder for the parameter `x` of `f`, and a place-holder for the return address from `f`. To illustrate the difference between MachAbs and MachConc, Figure 10 shows thread states of the MachAbs semantics and the MachConc semantics just before returning from function `f`. Note that the memory of MachAbs only allocates space for the MachAbs frame of `f`. Since `main` does not need to store anything on the stack, `main`'s MachAbs frame is empty and the corresponding stack frame pointer is `None`. In contrast, MachConc stores all local variables, return addresses and function arguments in memory. Unlike in MachAbs, the memory is allocated only when a thread starts and remains allocated until the thread exits.

5.4.1. Threadwise simulation definition. The threadwise simulation serves as an interface between the threadwise and whole-system correctness lemmas. We take care to make the definition parametric in the source and target transition systems so that we can completely separate the whole-system argument about the TSO machine from the semantics of MachAbs and MachConc.

In addition to relating the usual source and target states, our simulation relation keeps track of memory ownership and local memory content: we decorate the source and target states with lists of owned memory regions and we also associate local memory with the target state so that the simulation relation can describe relationship between the target's function (MachConc) frames in memory and the source's local state. As a result, the simulation relation is of the form

```
Variable rel : TgtS -> list arange  ->  mem ->
               SrcS -> list arange  ->
```

```
Prop.
```

where `TgtS` is the type of target transition system states, `SrcS` is the type of source states, `list arange` is a list of memory ranges and `mem` is memory. In our example in Figure 10, the source and target states are illustrated on the left. The list of source ranges corresponds to MachAbs frames, i.e., the list contains a single range – the stack frame of `f`. The list of target ranges is a list containing a single element – the thread's entire stack space. The target memory is the MachConc memory. We omit the source memory from the threadwise simulation because the relationship between the MachAbs memory and non-local MachConc memory will be specified by the whole-system simulation relation.

We require that the threadwise simulation relation must preserve stuckness, must only depend on local memory and must simulate events correctly. More precisely, the simulation relation preserves stuckness if for any related states s and t, if t is stuck then s is stuck. By local memory we mean the following: memory chunk c at location p is local for target list of ranges tp and source list of ranges sp if the chunk is inside some range in tp, but does not overlap with any range from sp. In our MachAbs-MachConc case, a chunk is local if it is in the part of the extra memory that was allocated by MachConc to hold the MachConc frames. A simulation relation rel is only dependent on local memory if for all states s, t, lists of ranges sp, tp and memories m, m' that have equal value in all their memory chunks, $rel(t, tp, m, s, sp)$ implies $irel(t, tp, m', s, sp)$. The local memory of MachConc is MachConc's stack space without the MachAbs frames. In Figure 10, the chunk containing variable `p` is local, but the chunk containing `i` is not.

The event simulation essentially says that operations on local memory in the target can be simulated by τ events in the source. Simulation of allocation is subtle: we allow the target semantics to allocate extra memory to store its local state. In our MachConc-MachAbs simulation, MachAbs and MachConc do not perform memory allocation at the same time: MachConc semantics allocates the entire stack space (our semantics allocates 8MB for thread stacks) at thread start and frees the space at thread exit, but MachAbs allocates its stack frames upon function entry and deallocates on function exit.

We illustrate the precise simulation definition on the example of read simulation that is a part of the event simulation:

```
Definition local_simulation :=
  forall ss ts ts' tm sp tp l,
    tgt_step ts l ts' ->
    rel ts tp tm ss sp ->
    match l with
      | ...
      | TEmem (MEread p c v)  => read_simulation ss ts ts' tm sp tp p c v
      | ...
    end.
```

That is if $ts \xrightarrow{l} ts'$ in the target semantics and ts, tp, tm are related with ss, sp, where tp are the memory ranges owned by the target state ts, sp are the memory ranges owned by the source state ss, tm is the local memory associated with the state ts, and l is a read event of value v from chunk c at location p, then we require that

— the state ss can reach an error, or
— the chunk c at location p is local for tp, sp, the load of the chunk from memory tm must succeed, and if the value of the chunk in memory matches the value v from

the event then the source semantics can do the transition $ss \xrightarrow{\tau} ss'$, where ss', sp is related to ts', tp, tm, or ss can stutter with decreasing measure, or

— the source semantics can perform a read transition to a related state, where the read must read the same value or the undef value.

The precise Coq statement of the definition follows.

```
Definition read_simulation ss ts ts' tm sp tp p c v :=
  (* Either can do an error *)
  stuck_or_error _ ss \/
  (* either it is a Machconcr-local access such that
       we can do matching tau in source *)
  (chunk_inside_range_list p c tp /\
   range_not_in (range_of_chunk p c) sp /\
   load_ptr c tm p <> None /\
   (load_ptr c tm p = Some v ->
    (exists ss', src_taustep ss TEtau ss' /\
                 rel ts' tp tm ss' sp) \/
    (rel ts' tp tm ss sp /\ ord ts' ts) \/
    stuck_or_error _ ss)) \/
  (* or we can do a read of any less defined value *)
  (forall v' (LD : Val.lessdef v' v),
    exists ss', src_taustep ss (TEmem (MEread p c v')) ss' /\
                rel ts' tp tm ss' sp).
```

5.4.2. Threadwise simulation for MachAbs to MachConc. The threadwise simulation relation essentially requires that the MachAbs state matches MachConc frames in memory and that the MachConc and MachAbs registers match, with the exception of the stack pointer, which points to the stack frame in MachAbs, but in MachConc points to the frame. In reality, the simulation relation must also keep track of several other technical invariants. Most notably, we require that all MachAbs frame ranges are inside the stack space of MachConc, the stack space is allocated in MachConc's memory and all MachConc frames are properly aligned.

Although the simulation proof does not explicitly mention separation between the individual function frames, most of the proof work concentrates on establishing separation of memory regions to guarantee non-interference of memory operations with frames. For example, for a simulation of a local variable write in MachConc, not only we need to show the correspondence of the updated memory with the updated frame in MachAbs, we also must establish that all the other MachAbs frames still correspond to the MachConc memory, i.e., that the memory used by the other frames does not change. We do this by showing that the updated frame is disjoint from all the other frames. A similar proof obligation comes even from simulation of local reads, because the simulation of reads requires us to show that local reads do not interfere with MachAbs stack frames.

Another interesting example of the difference between the languages is the handling of function entry – while the MachAbs semantics allocates the MachAbs frame on function entry, the MachConc semantics simply decrements its stack pointer. If the decremented stack pointer exceeds the allocated stack range, the MachConc semantics issues an out-of-memory label and the simulation is trivially satisfied. Otherwise, we simulate function entry by the stack space allocation in MachAbs. Since the newly allocated memory is still inside the MachConc stack space, we keep the invariant requiring that each range allocated in MachAbs is a sub-range of some range allocated in MachConc.

We also show the stuckness simulation and non-interference with non-local memory requirements of the threadwise simulation by case analysis on the transition relation for stuckness and by induction on the derivation of the simulation relation for the non-interference.

5.4.3. Whole-system simulation from threadwise simulation. The overall structure of the whole-system simulation relation is similar to the Csharpminor-to-Cstacked relation: for both the target and source there must be disjoint partitioning of memory between threads such that the values in the allocated source memory are the same as the values in the target memory at the same location. For each thread we have (i) each range in the thread's source partition is a subrange of some range in the thread's target partition, (ii) the buffers of the threads are related in a similar way to the Csharpminor-Cstacked buffer relation, and (iii) the source and target thread states are related in their partitions and target memory after applying the thread's buffer.

The trickiest part is to prove that applying the buffer in one thread does not affect the state relation (iii) for all the other threads. This is tricky because our threadwise non-interference with non-local memory only applies to memory *after* completely applying the thread's buffer. To get the non-interference before applying the buffers we make a subtle use of the fact that no unbuffering can fail.

5.5. The 'easy' phases, including optimisations

We have enabled all the CompCert 1.5 optimisations that are sound under the TSO semantics except tail call optimisation. These are: constant propagation and partial evaluation, a restricted version of CSE (common subexpression elimination) that eliminates only common arithmetic expressions, but does not eliminate common memory loads, redundant load removal (as part of register allocation), and branch tunneling. Tail call optimisation is sound but not very useful in our setting as in the x86 ABI all function arguments are stack-allocated, so one only rarely has empty stack frames. The only CompCert 1.5 optimisation that we do not perform because it is unsound under the TSO memory model is CSE for memory reads, as demonstrated by the following example (adapted from [Pug00]):

```
          int x;
x = 0;  | void f (int *p) {int a = x, b = *p, c = x;
x = 1;  |   printf("%d%d%d", a, b, c);}
        | f(&x);
```

CSE would replace the assignment `c = x` with `c = a`, allowing the second thread to print 010, a behaviour that is not allowed by the TSO semantics.

Labellising CompCert's definitions of RTL, LTL, LTLin, Linear, MachAbs, and MachConc and establishing that they are determinate and receptive (so that they can be composed with the TSO machine) was straightforward because the CompCert 1.5 definitions of these languages were already fully small-step. Porting CompCert's downward simulation proofs to threadwise downward simulation proofs and lifting them to measured whole-system upward simulations using Theorems 4.5, 4.7 and 4.8 was equally straightforward. (In the early days of the project, porting one phase took approximately two days, but by the end 3 hours were sufficient to port constant propagation and lift it to a measured whole-system upward simulation.) Elimination of redundant loads required a small adaptation of the downward-to-upward simulation infrastructure. Moreover, the Cstacked-Cminor and spilling/reloading phases may change some of the undefined values in the source semantics to particular values in the target semantics requiring us to prove another slightly more general version of Theorems 4.5 and 4.7.

The CompCert instruction selection phase, from Cminor to CminorSel, uses various "smart constructors" to choose appropriate operations and addressing modes for the target machine; its correctness relies on many lemmas showing the correctness of these with respect to expression evaluation. To make it easy to port these lemmas, for this phase we introduced a "trace step" semantics, as outlined in §3.6. The inductive-on-expressions structure of these helped significantly, though some plumbing was required to compose the result with the adjacent phases.

5.6. The x86 backend

We adapted the x86 backend from CompCert 1.8 (CompCert 1.5 supported PowerPC and ARM only), with several notable differences in the semantics and proofs. Our x86 semantics is based on a well-tested HOL4 formalisation of part of the x86 instruction set [SSZN+09, Section 3]. The structure of our instruction AST is closer to that of general x86 instructions, with their various combinations of immediate, register and addressing-mode arguments, than the AST used in CompCert 1.8, which defines a flat AST supporting just the combinations used by the compiler. This does entail some additional complexity in the proof, but allows a more generally reusable and extensible x86 semantics. For instance, binary operations over integers are represented as

```
Xbinop : int_binop_name -> int_dest_src -> instruction
```

where `int_dest_src` accounts for all possible combinations of operands:

```
Inductive int_dest_src :=
| Xri (dst: ireg) (src: imm)   (**r r32, imm32/imm8(sign-extended) *)
| Xrr (dst: ireg) (src: ireg)  (**r r32, r32 *)
| Xrm (dst: ireg) (src: xmem)  (**r r32, m32 *)
| Xmr (dst: xmem) (src: ireg). (**r m32, r32 *)
```

Immediate operands to an arithmetic instruction or an indexed memory access, denoted by `imm`, can be either integer literals, or a symbolic reference (the address of a symbol – symbolic references are resolved later by the linker):

```
Inductive imm :=
| Cint (i: int)
| Csymbol (s: ident) (offset: int).
```

while the rich set of indexed-memory addressing modes is accounted by `xmem`:

```
Inductive xmem :=
| Xm (idx: option (word2 * ireg)) (base: option ireg) (displ: imm).
```

We faithfully model flag updates for integer arithmetic and comparison instructions, and the semantics of the conditional branches is defined in terms of the flag status (in contrast to CompCert 1.8 where it is axiomatised). This required proving several theorems about 32-bit integer arithmetic, relating the flag state to the logic result of comparison instruction; their proof is tiresome in Coq (while the HOL4 model-checker `blast` can prove them automatically by exhaustive exploration of the state space).

For floating point, we target the SSE2 instruction set, and floating point arithmetic and comparison semantics is axiomatised, as in CompCert 1.8.

We had to add a number of pseudo-instructions standing for short instructions sequences. Ideally these should have been represented as real instructions in the AST, but unfortunately their semantics cannot be specified in the current CompCert setting. These are:

— `Xxor_self r` standing for `XOR r, r`. This is a pseudo-instruction to work around the fact that `Val.xor x x` is not always `Vzero` (in particular, when `x = Vundef`);

— `Xset cc r` for setting `r <- cc`. This corresponds to

```
SETcc   CL
MOVZBL  r, CL
```

because the semantics of `SETcc` cannot be represented faithfully in isolation. `SETcc` writes only the least significant byte of `ECX` leaving the rest unchanged. If, however, `ECX` held `Vundef`, then the resulting value of `ECX` could not be represented as a CompCert value;

— several floating-point instructions: `Xmovftr` and `Xmovftm` for truncating floating point moves, dropping precision (these are pseudo-instructions because our semantic values do not include single-point floating point values); `Xmovstr` and `Xmovrst` for moving values between the FP stack and the XMM registers; `Xnegf` for negating floating point numbers; `Xfctiu` for converting a floating point number to an unsigned integer; and `Xiuctf` for converting an unsigned integer to a floating point number. These are pseudo-instructions because the floating point semantics is axiomatized.

OS-specific behaviours, like thread creation and thread exit, are axiomatised.

Many x86 assembler instructions can involve both a read and a write to memory, and their semantics must define two separate interactions with the TSO machine. This is done by extending the state of the local computations with *partially executed instructions*, that keep track of the pending write (if any) of the continuation.

At the compilation level, the main difference with CompCert 1.8 is that we replaced individual stackframe allocations with one-off stack space allocation at the start of the thread and direct stack pointer arithmetic. We detect stack overflow by checking that the stack pointer register stays inside the thread's stack space. If not, the semantics issues an explicit `oom` event.

Using the more realistic single stack space gives us the added benefit of direct access to function arguments and the return address. This contrasts with CompCert that accesses arguments through an indirect link to parent stackframe and models the return address with a virtual return-address register (similarly to PowerPC's real link register).

Saving the return address in the stack enables a more realistic modelling of x86, but several parts of the x86 semantics remain less realistic than we would wish. The most notable abstraction in the semantics is modelling register and memory contents by the high-level `value` datatype (as in CompCert), which is a discriminated union of pointers, integers, floats and undefined value, instead of the more appropriate bit-vector representation. Unfortunately, this deficiency is not easy to remove, mostly because code pointers, such as the instruction pointer register or return addresses on the stack, critically use the block-offset components of pointers for function id and instruction index within the function respectively.

6. FENCE OPTIMISATIONS

The previous two sections focussed on the correctness statement and proofs in the TSO setting of what were largely standard sequential-compiler phases, restricting some sequential optimisations (CSE in particular) to make them sound in that setting. In this section we turn to some concurrency-specific optimisations, removing redundant fence instructions. We detect and optimise away the following cases of redundant MFENCE instructions:

— a fence is redundant if it always follows a previous fence or locked instruction in program order, with no memory store instructions in between (FE1);

— a fence is redundant if it always precedes a later fence or locked instruction in program order, with no memory read instructions in between (FE2).

We also perform partial redundancy elimination (PRE) [MR79] to improve on the second optimisation: we selectively insert memory fences in the program to make fences that are redundant along *some* execution paths to be redundant along all paths, which allows FE2 to eliminate them. The combined effect of PRE and FE2 is quite powerful and can even hoist a fence instruction out of a loop, as we shall see later in this section.

The correctness of FE1 is intuitive: since no memory writes have been performed by the same thread since executing an atomic instruction, the thread's buffer must be empty and so the fence instruction is effectively a no-op and can be optimised away.

The correctness of FE2 is more subtle. To see informally why it is correct, first consider the simpler transformation that swaps a MFENCE instruction past an adjacent store instructions (that is, MFENCE;store $\rightsquigarrow$ store;MFENCE). To a first approximation, we can think of FE2 as successively applying this transformation to the earlier fence (and also commuting it over local non-memory operations) until it reaches the later fence; then we have two successive fences and we can remove one. Intuitively, the end-to-end behaviours of the transformed program, store;MFENCE, are a subset of the end-to-end behaviours of the original program, MFENCE;store: the transformed program leaves the buffer empty, whereas in the original program there can be up to one outstanding write in the buffer. Notice that there is an intermediate state in the transformed program that is not present in the original program: if initially the buffer is non-empty, then after executing the store instruction in store;MFENCE we end up in a state where the buffer contains the store and some other elements. It is, however, impossible to reach the same state in the original MFENCE;store program because the store always goes into an empty buffer. What saves soundness is that this intermediate state is not observable. Since threads can access only their own buffers, the only way to distinguish an empty buffer from a non-empty buffer must involve the thread performing a read instruction from that intermediate state.

Indeed, if there are any intervening reads between the two fences, the transformation is unsound, as illustrated by the following variant of SB+mfences:

Thread 0	Thread 1
MOV [x]←1	MOV [y]←1
MFENCE (*)	MFENCE
MOV EAX←[y]	MOV EBX←[x]
MFENCE	

If the MFENCE labelled with (*) is removed, then it is easy to find an x86-TSO execution that terminates in a state where EAX and EBX are both 0, which was impossible in the unoptimised program.

This 'swapping' argument works for finite executions, but does not account for infinite executions, as it is possible that the later fence is never executed — if, for example, the program is stuck in an infinite loop between the two fences. The essential difficulty of the proof is that FE2 introduces non-observable non-determinism. It is well-known that reasoning about such transformations cannot, in general, be done solely by a standard forward simulation (e.g., [LV95]), but it also requires a backward simulation [LV95] or, equivalently, prophecy variables [AL91]. We tried using backward simulation to carry out the proof, but found the backward reasoning painfully difficult. Instead, we came up with a new kind of forward simulation, which we call a *weak-tau simulation*, that incorporates a simple version of a boolean prophecy variable that is much easier to use and suffices to verify FE2. The details are in §6.3.

$$
\begin{array}{lll@{\qquad}lll}
T_1(\texttt{nop},\mathcal{E}) & = & \mathcal{E} & T_2(\texttt{nop},\mathcal{E}) & = & \mathcal{E} \\
T_1(\texttt{op}(op,\vec{r},r),\mathcal{E}) & = & \mathcal{E} & T_2(\texttt{op}(op,\vec{r},r),\mathcal{E}) & = & \mathcal{E} \\
T_1(\texttt{load}(\kappa,\textbf{\textit{addr}},\vec{r},r),\mathcal{E}) & = & \mathcal{E} & T_2(\texttt{load}(\kappa,\textbf{\textit{addr}},\vec{r},r),\mathcal{E}) & = & \top \\
T_1(\texttt{store}(\kappa,\textbf{\textit{addr}},\vec{r},\textbf{\textit{src}}),\mathcal{E}) & = & \top & T_2(\texttt{store}(\kappa,\textbf{\textit{addr}},\vec{r},\textbf{\textit{src}}),\mathcal{E}) & = & \mathcal{E} \\
T_1(\texttt{call}(sig,ros,\textbf{\textit{args}},\textbf{\textit{res}}),\mathcal{E}) & = & \top & T_2(\texttt{call}(sig,ros,\textbf{\textit{args}},\textbf{\textit{res}}),\mathcal{E}) & = & \top \\
T_1(\texttt{cond}(cond,\textbf{\textit{args}}),\mathcal{E}) & = & \mathcal{E} & T_2(\texttt{cond}(cond,\textbf{\textit{args}}),\mathcal{E}) & = & \mathcal{E} \\
T_1(\texttt{return}(\textbf{\textit{optarg}}),\mathcal{E}) & = & \top & T_2(\texttt{return}(\textbf{\textit{optarg}}),\mathcal{E}) & = & \top \\
T_1(\texttt{threadcreate}(\textbf{\textit{optarg}}),\mathcal{E}) & = & \top & T_2(\texttt{threadcreate}(\textbf{\textit{optarg}}),\mathcal{E}) & = & \top \\
T_1(\texttt{atomic}(\textbf{\textit{aop}},\vec{r},r),\mathcal{E}) & = & \bot & T_2(\texttt{atomic}(\textbf{\textit{aop}},\vec{r},r),\mathcal{E}) & = & \bot \\
T_1(\texttt{fence},\mathcal{E}) & = & \bot & T_2(\texttt{fence},\mathcal{E}) & = & \bot
\end{array}
$$

Fig. 11. Transfer functions for FE1 and FE2

We can observe that neither optimisation subsumes the other: in the program below on the left the (*) barrier is removed by FE2 but not by FE1, while in the program on the right the (†) barrier is removed by FE1 but not by FE2.

MOV [x]←1	MFENCE
MFENCE (*)	MOV EAX←[x]
MOV [x]←2	MFENCE (†)
MFENCE	MOV EBX←[y]

6.1. Implementation

The fence instructions eligible to be optimised away are easily computed by two intra-procedural dataflow analyses over the boolean domain, $\{\bot, \top\}$, performed on RTL programs. Among the intermediate languages of CompCertTSO, RTL is the most convenient to perform these optimisations, and it is the intermediate language where most of the existing optimisations are performed: namely, constant propagation, CSE, and register allocation.

The first is a *forward* dataflow problem that associates to each program point the value $\bot$ if along *all* execution paths there is an atomic instruction *before* the current program point with no intervening writes, and $\top$ otherwise. The problem can be formulated as the solution of the standard forward dataflow equation:

$$\mathcal{FE}_1(n) = \begin{cases} \top & \text{if predecessors}(n) = \emptyset \\ \bigsqcup_{p \in \text{predecessors}(n)} T_1(\textbf{\textit{instr}}(p), \mathcal{FE}_1(p)) & \text{otherwise} \end{cases}$$

where p and n are program points (i.e., nodes of the control-flow-graph), the join operation is logical disjunction (returning $\top$ if at least one of the arguments is $\top$), and the transfer function T_1 is defined in Fig. 11.

The second is a *backward* dataflow problem that associates to each program point the value $\bot$ if along *all* execution paths there is an atomic instruction *after* the current program point with no intervening reads, and $\top$ otherwise. This problem is solved by the standard backward dataflow equation:

$$\mathcal{FE}_2(n) = \begin{cases} \top & \text{if successors}(n) = \emptyset \\ \bigsqcup_{s \in \text{successors}(n)} T_2(\textbf{\textit{instr}}(s), \mathcal{FE}_2(s)) & \text{otherwise} \end{cases}$$

where the join operation is again logical disjunction and the transfer function T_2 is defined in Fig. 11.

To solve the dataflow equations we reuse the generic implementation of Kildall's algorithm provided by the CompCert compiler. Armed with the results of the dataflow

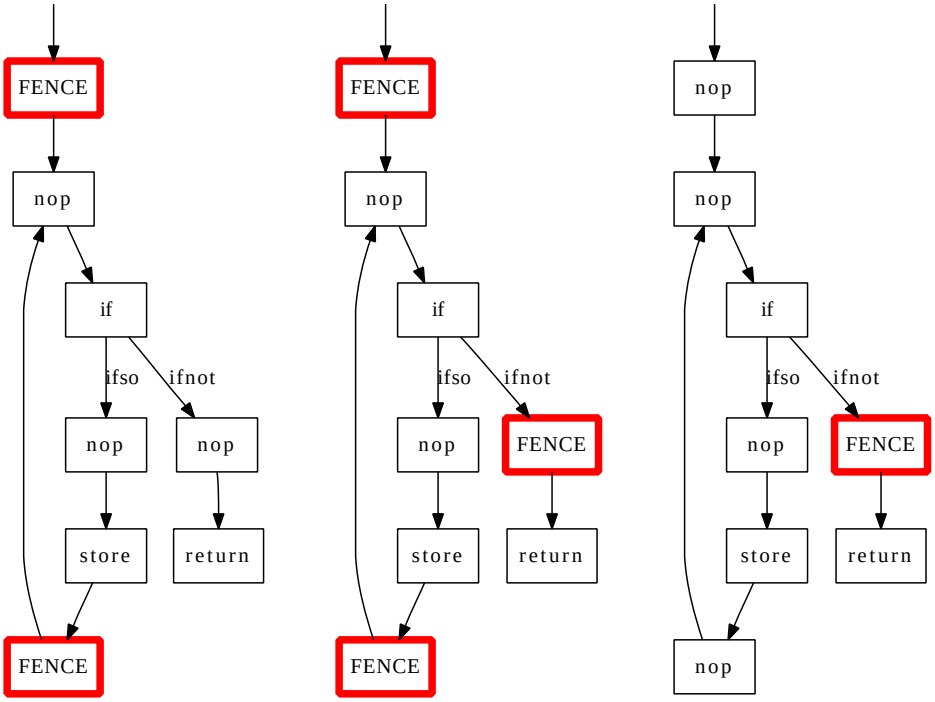


Fig. 12. Unoptimised RTL, RTL after PRE, and RTL after PRE and FE2

analysis, a pass over the RTL source replaces the fence nodes whose associated value in the corresponding analysis is $\bot$ with `nop` (no-operation) nodes, which are removed by a later pass of the compiler.

6.2. Partial Redundancy Elimination

In practice, it is common for MFENCE instructions to be redundant on some but not all paths through a program. To help with these cases, we perform a partial redundancy elimination phase (PRE) that inserts `fence` instructions so that partially redundant fences become fully redundant. For instance, the RTL program on the left of Fig. 12 (from Fraser's lockfree-lib) cannot be optimised by FE2: PRE inserts a memory fence in the *ifnot* branch, which in turn enables FE2 to rewrite the program so that all execution paths go through at most one `fence` instruction.

The implementation of PRE runs two static analyses to identify the program points where `fence` nodes should be introduced. First, the RTL generation phase introduces a nop as the first node on each branch after a conditional; these nop nodes will be used as placeholders to insert (or not) the redundant barriers. We then run two static analyses:

— the first, called A, is a backward analysis returning $\top$ if along *some* path after the current program point there is an atomic instruction with no intervening reads;
— the second, called B, is a forward analysis returning $\bot$ if along *all* paths to the current program point there is a `fence` with no later reads or atomic instructions.

The transformation inserts fences *after conditional nodes* on branches whenever:

$$
\begin{array}{lll|lll}
T_A(\texttt{nop},\mathcal{E}) & = & \mathcal{E} & T_B(\texttt{nop},\mathcal{E}) & = & \mathcal{E} \\
T_A(\texttt{op}(op,\vec{r},r),\mathcal{E}) & = & \mathcal{E} & T_B(\texttt{op}(op,\vec{r},r),\mathcal{E}) & = & \mathcal{E} \\
T_A(\texttt{load}(\kappa,\boldsymbol{addr},\vec{r},r),\mathcal{E}) & = & \bot & T_B(\texttt{load}(\kappa,\boldsymbol{addr},\vec{r},r),\mathcal{E}) & = & \top \\
T_A(\texttt{store}(\kappa,\boldsymbol{addr},\vec{r},\boldsymbol{src}),\mathcal{E}) & = & \mathcal{E} & T_B(\texttt{store}(\kappa,\boldsymbol{addr},\vec{r},\boldsymbol{src}),\mathcal{E}) & = & \mathcal{E} \\
T_A(\texttt{call}(sig,ros,\boldsymbol{args},\boldsymbol{res}),\mathcal{E}) & = & \bot & T_B(\texttt{call}(sig,ros,\boldsymbol{args},\boldsymbol{res}),\mathcal{E}) & = & \top \\
T_A(\texttt{cond}(cond,\boldsymbol{args}),\mathcal{E}) & = & \mathcal{E} & T_B(\texttt{cond}(cond,\boldsymbol{args}),\mathcal{E}) & = & \mathcal{E} \\
T_A(\texttt{return}(\boldsymbol{optarg}),\mathcal{E}) & = & \bot & T_B(\texttt{return}(\boldsymbol{optarg}),\mathcal{E}) & = & \top \\
T_A(\texttt{threadcreate}(\boldsymbol{optarg}),\mathcal{E}) & = & \bot & T_B(\texttt{threadcreate}(\boldsymbol{optarg}),\mathcal{E}) & = & \top \\
T_A(\texttt{atomic}(\boldsymbol{aop},\vec{r},r),\mathcal{E}) & = & \top & T_B(\texttt{atomic}(\boldsymbol{aop},\vec{r},r),\mathcal{E}) & = & \bot \\
T_A(\texttt{fence},\mathcal{E}) & = & \top & T_B(\texttt{fence},\mathcal{E}) & = & \bot
\end{array}
$$

Fig. 13. Transfer functions for analyses A and B of PRE

— analysis B returns $\bot$ (i.e., there exists a previous fence that will be eliminated if we were to insert a fence at both branches of the conditional nodes); and
— analysis A returns $\bot$ (i.e., the previous `fence` will not be removed by FE2); and
— analysis A returns $\top$ *on the other branch* (the other branch of the conditional already makes the previous `fence` partially redundant).

If all three conditions hold for a `nop` node following a branch instruction, then that node is replaced by a `fence` node. A word to justify the *some path* (instead of *for all paths*) condition in analysis A: as long as there is a fence on some path, then at all branch points PRE would insert a fence on all other paths, essentially converting the program to one having fences on all paths.

The transfer functions T_A and T_B are detailed in Fig. 13. Note that T_B defines the same transfer function as T_2, but here it is used in a forward, rather than backward, dataflow problem.

6.3. Proofs of the optimisations

We give brief outlines of the formal Coq proofs of correctness for the three fence elimination optimisations.

6.3.1. Fence Elimination 1. We verify this optimisation by a whole-system measured upward simulation.

Take $>$ to be empty relation (which is trivially well-founded) and $s\ R\ t$ the relation requiring that (*i*) the control-flow-graph of t is the optimised version of the CFG of s, (*ii*) s and t have identical program counters, local states, buffers and memory, and (*iii*), for each thread i, if the analysis for i's program counter returned $\bot$, then i's buffer is empty.

It is straightforward to show that each target step is matched exactly by the corresponding step of the source program. In the case of a `nop` instruction, this could arise either because of a `nop` in the source or because of a removed `fence`. In the latter case, the analysis will have returned $\bot$ and so, according to $\sim$, the thread's buffer is empty and so the `fence` proceeds (i.e., it does not block).

6.3.2. Fence Elimination 2. We verify this optimisation by exhibiting a weak-tau simulation. Eliding assumptions on initial states, weak-tau simulations are defined as follows:

Definition 6.1. A pair of relations $R, R' : \mathsf{States}(S) \times \mathsf{States}(T)$, equipped with a relation $<$ on $\mathsf{States}(T)$, is a *weak-tau upward simulation* if:

(1) $R \subseteq R'$ (i.e., for all s, t, $s\ R\ t$ implies $s\ R'\ t$); and

(2) Whenever $s\ R\ t$ and $t \xrightarrow{ev} t'$, then either
  (a) $\exists s'.\ s \xrightarrow{\tau}{}^{*}\ s' \xrightarrow{\texttt{fail}}$ (s can reach a semantic error), or
  (b) $\exists s'.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{ev} s' \wedge s'\ R\ t'$ (s can do a matching step), or
  (c) $ev = \tau \wedge t' < t \wedge s\ R\ t'$ (t stuttered); and
(3) Whenever $s\ R'\ t$ and $t \xrightarrow{\tau} t'$ and $t' < t$,
  (a) $\exists s'.\ s \xrightarrow{\tau}{}^{*}\ s' \xrightarrow{\texttt{fail}}$ (s can reach a semantic error), or
  (b) $\exists s'.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{\tau} s' \wedge s'\ R'\ t'$ (s can do a matching step).

Similar to measured upward simulations, weak-tau simulations imply trace inclusion. To prove this in the case where the target trace contains an infinite τ sequence, we do a case split on whether the trace contains an infinite sequence of states in the $<$ relation. If it does, then we can use the relation R' to construct an infinite sequence of source τ transitions. Otherwise, the relation R can stutter only for finite sequences of τ steps each time and will thus produce an infinite sequence.

THEOREM 6.2. *A weak-tau upward simulation implies trace inclusion.* [Coq proof]

To verify the optimisation, we will use the following auxiliary definitions:

— Define $s \equiv_i t$ to hold whenever thread i of s and t have identical program counters, local states and buffers.
— Define $s \rightsquigarrow_i s'$ if thread i of s can execute a sequence of `nop`, `op`, `store` and `fence` instructions and end in the state s'.
— Define $t' < t$ to hold whenever $t \xrightarrow{\tau} t'$ by a thread executing a `nop`, an `op`, or a `store` instruction.

Take $s\ R\ t$ the relation requiring that (*i*) t's CFG is the optimised version of s's CFG, (*ii*) s and t have identical memories, (*iii*), for each thread i, either $s \equiv_i t$ or the analysis for i's program counter returned $\bot$ (meaning that there is a later fence in the CFG with no reads in between) and there exists a state s_0 such that $s \rightsquigarrow_i s_0$ and $s_0 \equiv_i t$.

Take $s\ R'\ t$ to be the relation requiring that: (*i*) the CFG of t is the optimised version of the CFG of s, and (*ii*), for each thread i, there exists s_0 such that $s \rightsquigarrow_i s_0$ and $s_0 \equiv_i t$.

We will now show that R, R', and $<$ form a weak-tau simulation. First, observe that condition (1) follows immediately from the definition; that is, $R \subseteq R'$.

To prove condition (2), we match every step of the target with the corresponding step of the source whenever the analysis at the current program point of the thread doing the step returns $\top$. It is possible to do so, because by the simulation relation ($s\ R\ t$), we have $s \equiv_i t$.

Now, consider the case when the target thread i does a step and the analysis at the current program point returns $\bot$. According to the simulation relation (R), we have $s \rightsquigarrow_i s_0 \equiv_i t$. Because of the transfer function, T_2, that step cannot be a `load` or a `call`/`return`/`threadcreate`. We are left with the following cases:

— `nop` (either in the source program or because of a removed fence), `op`, or `store`. In these cases, we stutter in the source, i.e. do $s\ R\ t'$. This is possible because we can perform the corresponding transition from s_0 (i.e., there exists an s' such that $s \rightsquigarrow_i s_0 \rightsquigarrow_i s' \equiv_i t'$).
— `fence`, `atomic`: This is matched by doing the sequence of transitions from s to s_0 followed by flushing the local store buffer and finally executing the corresponding `fence` or `atomic` instruction from s_0.
— Thread i unbuffering: If i's buffer is non-empty in s, then unbuffering one element from s preserves the simulation relation. Otherwise, if i's buffer is empty, then there exists an s' such that $s \rightsquigarrow_i s' \rightsquigarrow_i s_0$ and i's buffer in s' has exactly one element.

	br	*br*+FE1	*aw*	*aw*+FE2	*aw*+PRE+FE2
Dekker	3	2	5	4	4
Bakery	10	2	4	3	3
Treiber's stack	5	2	3	1	1
Fraser's skiplist	32	18	19	12	11
TL2	166	95	101	68	68
Genome	133	79	62	41	41
Labyrinth	231	98	63	42	42
SSCA	1264	490	420	367	367

Fig. 14. Static number of fences present after fence optimisations

Then the transition from $t \xrightarrow{\tau} t'$ is simulated by first doing $s \xrightarrow{\tau}{}^{*} \xrightarrow{\tau} s'$ followed by an unbuffering from s', which preserves the simulation relation.

To prove condition (3), we simulate a target thread transition by doing the sequence of transitions from s to s_0 followed by executing the corresponding instruction from s_0.

6.3.3. Partial Redundancy Elimination. Even though this optimisation was the most complex to implement, its proof was actually the easiest. What this optimisation does is to replace some `nop` instructions by `fence` instructions depending on some non-trivial analysis. However, as far as correctness is concerned, it is always safe to insert a `fence` instruction irrespective of whatever analysis was used to used to decide to perform the insertion. Informally, this is because inserting a memory fence just restricts the set of behaviours of the program; it never adds any new behaviour.

In the formal proof, we take the simulation relation to be equality except on the programs themselves, where we require the target program to be the 'optimised' version of the source program. Executing the inserted `fence` instruction in the target is simulated by executing the corresponding `nop` in the source.

7. RUNNING CompCertTSO

Despite making little attempt at optimising the generated code, results on simple sequential and concurrent benchmarks (mostly drawn from [Com09]) show that our generated code runs at about 75% of the performance of `gcc -O1`. As a more representative example, we have also successfully compiled Fraser's lock-free skiplist algorithm [Fra03]; we are roughly 70% of the performance of `gcc -O1` on this benchmark. Porting required only three changes, all to in-line assembly macros, two of which were replacing macros for `CAS` and `MFENCE` by the ClightTSO constructs.

7.1. Fence optimisation

For a crude investigation of the effect of the fence optimisations, we instructed the RTL generation phase of CompCertTSO to systematically introduce an MFENCE instruction before each memory read (strategy *br*), or after each memory write (strategy *aw*), and looked at how many were removed by the fence optimisation phases. In Figure 14 we consider several well-known concurrent algorithms, including Dekker and Bakery mutual exclusion algorithms, Treiber's stack [Tre86], the TL2 lock-based STM [DSS06], the already mentioned Fraser's lockfree implementation of skiplists, and several benchmarks from the STAMP benchmark [CMCKO08]; for each the table reports the total numbers of fences in the generated assembler files, following the *br* and *aw* strategies, possibly enabling the FE1, PRE and FE2 optimisations.

A basic observation is that FE2 removes on average about 30% of the MFENCE instructions, while PRE does not further reduce the static number of fences, but rather reduces the dynamic number of fences executed, e.g. by hoisting fences out of loops as

in Figure 12. When it comes to execution times, then the gain is much more limited than the number of fences removed. For example, we observe a 3% speedup when PRE and FE2 are used on the skiplist code (running `skiplist 2 50 100` on a 2-core x86 machine): the hand-optimised (barrier free) version by Fraser is about 45% faster than the code generated by the *aw* strategy. For Lamport's bakery algorithm we generate optimal code for `lock`, as barriers are used to restore SC on accesses to the `choosing` array.

Looking at the fences we do not remove in more detail, the Treiber stack is instructive, as the only barrier left corresponds to an update to a newly allocated object, and our analyses cannot guess that this newly allocated object is still local; a precise escape analysis would be required. In general, about the half of the remaining MFENCE instructions precede a function call or return; we believe that performing an interprocedural analysis would remove most of these barriers. Our focus here is on *verified* optimisations rather than performance alone, and the machine-checked correctness proof of such sophisticated optimisations is a substantial challenge for future work.

8. DISCUSSION

We reflect briefly on the impact of the tool chain and proof style that we employed to ease development of our compiler.

The main tool was Coq. Here we found the proof style advocated by SSReflect [GM07] to be helpful in ensuring proof robustness, but to retain backward compatibility with CompCert, we employed it selectively. Occasionally, we used specialised tactics to automate some of the more tedious proofs, such as the threadwise determinacy and receptiveness of all the languages.

To give the reader a flavour for the effort involved in the development, we list the number of lines of proof and specifications (definitions and statements of lemmas and theorems) for the various parts of our compiler, as reported by `coqwc`. Blank lines and comments are not counted.

	Specs	Proof
Library code	8998	11058
ClightTSO definition (§3)	2424	267
x86 definition (§5.6)	1012	70
TSO machine (§5.1)	992	992
ClightTSO to Csharpminor (§5.2)	1990	2526
Csharpminor to Cstacked (§5.3)	3421	8163
10 intermediate language definitions (§5.5)	6466	1107
Intermediate 'easy' phases (§5.5)	11166	9742
Fence Elimination (§6)	940	1099
MachAbs to MachConc (§5.4)	2724	6525
MachConc to Asm (§5.6)	1729	2833
TOTAL	41862	44382

As described in §4, we structured our development to re-use as much of CompCert 1.5 as we could, but much is new. The total of 86K lines for CompCertTSO compares with around 55K lines for CompCert 1.5 (31K lines of specifications and 23K lines of proofs). The project has taken approximately 45–50 man-months.

The semantics of ClightTSO is given as an inductively defined relation, as usual and following Clight. To make it easier to check the integrity of the definition, we also implemented a functional characterisation of the threadwise single-step transition relation and proved that the two definitions are equivalent. By extracting the functional version into an OCaml program serving as an interpreter, we were able to

test the semantics on sample ClightTSO programs. This revealed a number of subtle errors in our original definitions. It would also be worth testing our x86 semantics against processor behaviour, as we did for a HOL4 x86 semantics in previous work with Myreen [SSZN+09]. As mentioned in §5.6, and as in CompCert, there is a significant semantic gap between our assembly semantics and that of actual x86 machine code (with CompCert values rather than bit vectors); this "final phase" verification is an open question for future work.

A mechanised theorem is only useful if its statement can be understood, and for CompCertTSO the overall correctness theorem involves the ClightTSO and x86 semantics. We defined ClightTSO using Ott [SZNO+10], a tool that generates Coq and LaTeX definitions from a single source; it also helped in enforcing naming conventions. The ClightTSO grammar and semantic rules, and the terms in examples, are all parsed and automatically typeset.

9. RELATED WORK

Research on verified compilation of sequential languages has a long history. Notable recent work includes CompCert, which we have already discussed in detail; Chlipala's compiler from a small impure functional language to an idealised assembly language, focussing on Coq proof automation [Chl10]; Myreen's JIT compiler from a bytecode to x86 [Myr10]; and Benton and Hur's compilation [BH09] from a simply typed functional language to a low-level SECD machine. This last differs from most other work in giving a compositional understanding of compiler correctness rather than just a relationship between the whole-program behaviours of source and target.

Verified compilation of concurrent languages has received much less attention. Perhaps the most notable example is the work of Lochbihler [Loc10] extending Jinja (a compiler from sequential Java to JVM, verified in Isabelle/HOL) to concurrency. As here, shifting to a small-step semantics required non-trivial proof effort, but the Jinja memory accesses in source and target are very closely related, so issues of relaxed-memory behaviour, memory layout, finite memory, and so on seem to have played no role. To the best of our knowledge, there is no prior work addressing verified compilation for a relaxed-memory concurrent language.

An alternative approach to extending CompCert with concurrency has been suggested by Hobor et al. [HAZN08]. They define a concurrent version of Cminor equipped with a concurrent separation logic. The idea is to do verifying compilation for programs that have been proved correct in such a logic, and their *oracle semantics* for concurrent Cminor (factored rather differently to ours) is intended to make that possible without extensive refactoring of the CompCert proofs. That is in some sense complementary to our work: we focus on intrinsically racy concurrent algorithms, whereas programs proved correct in that logic are known to be race free (as most application code is expected to be). However, we conjecture that an oracle semantics could be defined directly above the labellised semantics that we use. More recently, Stewart and Appel [SA11] report on an almost-complete mechanised soundness proof of a separation logic for Cminor, introducing machinery to factor out the world structure required for the logic from the language operational semantics.

The problem of inserting memory barriers so that a program admits only SC executions has been an important research topic since Sasha and Snir's seminal paper on delay set analysis [SS88]. Most formal studies of this problem [SS88; Alg10; BMS10] have been in terms of hypothetical program executions and, unlike our work, have not been integrated in a working compiler. There is also some more practical algorithm and compiler work. Lee and Padua [LP01] describe an algorithm based on dominators for inserting memory fences, while Sura et al. [SFW+05] focus on the more practical aspects, e.g., on how to approximate delay sets by performing cheaper whole-program

analyses coupled with an escape analysis. These perform much more sophisticated analyses than the ones we implemented, but none comes with a mechanised soundness proof. Another line of research [BAM07; HR07; KVY10] uses model checking techniques to insert fences to ensure SC. While these techniques may insert fewer fence instructions for small intricate concurrent libraries, they often guarantee soundness only for some clients of those libraries, and are too expensive to perform in a general-purpose compiler.

ClightTSO is not intended as a proposal for a complete language: its load and store operations are loosely analogous to the C++0x *atomics* [Bec10; BOS+11] and Java *volatiles* [MPA05], and it has no distinguished class of memory operations which are supposed to be thread-local or otherwise race-free (and hence which a compiler is licenced to optimise between synchronisation points). It is closer to the pseudocode or C-with-macros that is commonly used for concurrent shared-memory algorithms, and the ClightTSO operations can be implemented efficiently, with simple x86 loads and stores. Volatiles and C++0x SC atomics need heavier implementations, though C++0x also has cheaper *low-level atomics* with weaker semantics that are cheaper to implement. Java and C++0x also have more complex semantics, albeit not specific to TSO processors (essentially x86 and Sparc).

10. CONCLUSION

The shift to commodity multicore processors has recently made relaxed-memory concurrent computation pervasive, but semantics and verification in this setting is a long-standing problem. As Lamport wrote in 1979 [Lam79]:

> *For some applications, achieving sequential consistency may not be worth the price of slowing down the processors. In this case, one must be aware that conventional methods for designing multiprocess algorithms cannot be relied upon to produce correctly executing programs. Protocols for synchronizing the processors must be designed at the lowest level of the machine instruction code, and verifying their correctness becomes a monumental task.*

This paper is a step towards putting them on a rigorous foundation, both for programming and verification. While it remains a very challenging task, it is no longer monumental: the advances in semantics and reasoning techniques that we can bring to bear make it entirely feasible.

Acknowledgements. We thank Xavier Leroy for enlightening discussions and for making CompCert available.

REFERENCES

S. V. Adve and M. D. Hill. Weak ordering — a new definition. In *Proc. ISCA*, 1990.

Martn Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, pages 253–284, 1991.

Jade Alglave. *A shared memory poetics*. PhD thesis, Université Paris 7, 2010.

J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Proc. CAV*, 2010.

H.-J. Boehm and S.V. Adve. Foundations of the C++ concurrency memory model. In *Proc. PLDI*, 2008.

Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 12–21. ACM, 2007.

P. Becker, editor. *Programming Languages — C++. Final Committee Draft.* 2010. ISO/IEC JTC1 SC22 WG21 N3092.

N. Benton and C.K Hur. Biorthogonality, step-indexing and compiler correctness. In *Proc. ICFP*, 2009.

Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

Sebastian Burckhardt, Madanlal Musuvathi, and Vasu Singh. Verifying local transformations on relaxed memory models. In *CC*, 2010.

H.-J. Boehm. Threads cannot be implemented as a library. In *Proc. PLDI*, pages 261–268, 2005.

M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proc. POPL*, 2011.

Programming languages – C (committee draft, WG14 N1494, ISO/IEC 9899:201x). `http://www.open-std.org/jtc1/sc22/wg14/www/docs/PostColorado.htm`.

A. Chlipala. A verified compiler for an impure functional language. In *Proc. POPL*, 2010.

P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *Proc. ESOP*, 2007.

Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

The Compcert verified compiler, v. 1.5. `http://compcert.inria.fr/release/compcert-1.5.tgz`, August 2009.

The Coq proof assistant. `http://coq.inria.fr/`.

David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC*, 2006.

Keir Fraser. *Practical Lock Freedom*. PhD thesis, 2003. Also available as Tech. Report UCAM-CL-TR-639.

G. Gonthier and A. Mahboubi. A small scale reflection extension for the coq system. Technical report, 2007.

A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proc. ESOP*, 2008.

Thuan Quang Huynh and Abhik Roychoudhury. Memory model sensitive bytecode verification. *Form. Methods Syst. Des.*, 31:281–305, December 2007.

Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. In *FMCAD*, 2010.

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, 1979.

D. Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. 1999.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

1999. Linux Kernel mailing list, thread "spin_unlock optimization(i386)", 119 messages, Nov. 20–Dec. 7th, `http://www.gossamer-threads.com/lists/engine?post=105365;list=linux`. Accessed 2009/11/18.

A. Lochbihler. Verifying a compiler for Java threads. In *Proc. ESOP'10*, 2010.

Jaejin Lee and David A. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Trans. Comput.*, 50:824–833, August 2001.

Nancy Lynch and Frits Vaandrager. Forward and backward simulations I: untimed systems. *Inf. Comput.*, 121:214–233, September 1995.

R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

J. Manson, W. Pugh, and S.V. Adve. The Java memory model. In *Proc. POPL*, 2005.

E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22:96–103, February 1979.

M. O. Myreen. Verified just-in-time compiler on x86. In *Proc. POPL*, 2010.

S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *Proc. TPHOLs*, 2009.

S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *Proc. ECOOP*, 2010.

W. Pugh. The Java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6), 2000.

J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, 2008.

Gordon Stewart and Andrew W. Appel. Local actions for a curry-style operational semantics. In *Proc. PLPV*, pages 31–42, 2011.

P. Sewell. On implementations and semantics of a concurrent programming language. In *Proc. CONCUR*, July 1997.

Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler techniques for high performance sequentially consistent Java programs. In *PPoPP*, pages 2–13, New York, NY, USA, 2005. ACM.

The SPARC architecture manual, v. 9. `http://dev elopers.sun.com/solaris/articles/sparcv9.pdf`.

The SPARC Architecture Manual, V. 8. SPARC International, Inc., 1992. Revision SAV080SI9308. `http://www.sparc.org/standards/V8.pdf`.

Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10:282–312, 1988.

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *Proc. PLDI*, 2011.

P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *C. ACM*, 53(7):89–97, 2010.

S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. POPL*, 2009.

Jaroslav Ševčík , Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. In *Proc. POPL*, 2011.

P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.

R. K. Treiber. Systems programming: Coping with parallelism. Technical report, 1986.

E. Torlak, M. Vaziri, and J. Dolby. MemSAT: checking axiomatic specifications of memory models. In *PLDI*, 2010.

V. Vafeiadis and F. Zappa Nardelli. Verifying fence elimination optimisations. In *Proc. SAS 2011*.