

Relaxed-Memory Concurrency and Verified Compilation

Jaroslav Ševčík¹ Viktor Vafeiadis¹ Francesco Zappa Nardelli² Suresh Jagannathan³ Peter Sewell¹

¹University of Cambridge

²INRIA

³Purdue University

(work done while on sabbatical at Cambridge)

Abstract

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.

We define a concurrent relaxed-memory semantics for *ClightTSO*, an extension of CompCert’s Clight in which the processor’s memory model is exposed for high-performance code. We discuss a strategy for verifying compilation from ClightTSO to x86, which we validate with correctness proofs (building on CompCert) for the most interesting compiler phases.

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

Keywords Relaxed Memory Models, Verifying Compilation, Semantics

1. Introduction

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 [SSZ⁺09]. 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.

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) will 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. Another alternative is to forbid programs containing races and give SC semantics to the rest [AG96], relying on synchronisation from the implementations of lock and unlock. Precisely defining a non-SC 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, vA08, TVD10], and the ongoing work on C++0x [Bec10].

However, when it comes to concurrent systems code and concurrent data structure libraries, for example as used in an OS kernel and in `java.util.concurrent` [Lea99], it seems that 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]. This is 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 such 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, Ler09]. In this paper, we consider verifying compilation in the setting of concurrent programs with a realistic relaxed memory model.

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 or Sparc 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 §6, in a full language one would expect to augment these with thread-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 most of our phases (10 of 15) the memory accesses of source and target are in exact 1:1 correspondence. Moreover, for two 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.

Thirdly, we present evidence that our approach is effective (§5). We have implemented a compiler from ClightTSO to x86 multiprocessors, taking CompCert as a starting point, and have proved correctness (in Coq [Coq]) for key phases thereof. 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]. Finally, we reflect on the formalisation process and on the tools we used (§6), discuss related work, and conclude. 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.

2. Background: x86-TSO

We begin by recalling the relaxed-memory behaviour of our target language, x86 multiprocessor assembly programs, as captured

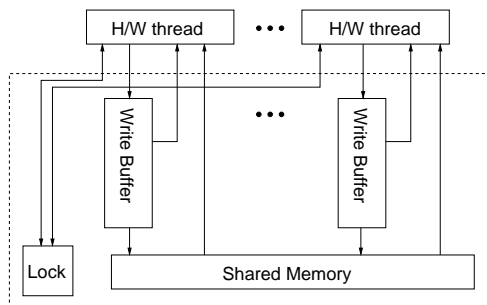


Figure 1. x86-TSO block diagram

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 $[x] \leftarrow 1$	MOV $[y] \leftarrow 1$
MOV EAX $\leftarrow [y]$	MOV EBX $\leftarrow [x]$
Allowed Final State: Thread 0:EAX=0 \wedge Thread 1:EBX=0	

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 $\text{INC } [x]$. 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: $\text{LOCK INC } [x]$ 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, for example in an OS kernel (where garbage collection may be infeasible), and many concurrent algorithms are expressed in C-like pseudocode. Second, it is an attractive starting point for research in relaxed-memory PL 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, 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

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

```

type, ty ::= void | int (intsize, signedness)
| float (floatsize) | pointer (ty) | array (ty, len)
| function ( $\tau^*$ , ty) | struct (id,  $\phi$ ) | union (id,  $\phi$ )
| comp_pointer(id) | (ty)
fieldlist,  $\phi$  ::= nil | (id, ty):: $\phi$ 
unary_operation,  $op_1$  ::= ! | ~ | -
binary_operation,  $op_2$  ::= + | - | * | / | % | & | | | ^ | << | >>
| == | != | < | > | <= | >=
expr, e ::= aty
expr_descr, a ::= n | f | id | *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, as ::= CAS | ATOMIC_INC | MFENCE
statement, s ::= skip |  $e_1 = e_2$  | opt_lhs e' (es) |  $s_1 ; s_2$ 
| 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 as (es)
labeled_statements, ls ::= default :s | case n:s ; ls
fn_defn_internal ::= ty id (arglist) {varlist s}
program ::= dcls fn_defns main = id

```

Figure 2. ClightTSO abstract syntax (excerpts)

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:

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).

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.

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()); }

```

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 are simply 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 they 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 enqueueing them. Otherwise one would have latent failures, e.g. if two threads free a block and those frees are both held in buffers.

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 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, for each function, annotations giving bounds on those. One could then reason about real space usage in terms of such an annotated source semantics.

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 significant re-engineering (described in §5.1) 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:

$$\begin{aligned}
\text{expr_cont}, \kappa_e &::= [op_1^T] \cdot \kappa_e \mid [-^T = e_2] \cdot \kappa_s \mid [v^T = _] \cdot \kappa_s \mid \dots \\
\text{stmt_cont}, \kappa_s &::= \text{stop} \mid [-; s_2] \cdot \kappa_s \mid \dots \\
\text{state} &::= \begin{array}{l} e \cdot \kappa_e \mid \rho \\ \mid \\ s \cdot \kappa_s \mid \rho \\ \mid \\ \dots \end{array}
\end{aligned}$$

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 §5.1. We also fix a left-to-right evaluation order.

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

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

```

Spinlock using CAS More usefully, an efficient spinlock can be implemented directly in ClightTSO using CAS. Any integer variable can be used to represent the state of the spinlock, with lock and unlock as follows:

```

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

```

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.

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

```

double channel; int flag = 0;
// sender { channel = 5.2; flag = 1; }
// receiver { while (flag == 0); 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] (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.

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. We

take the observable behaviour of both ClightTSO and x86-TSO programs to be labelled transition systems (LTS) with visible actions for call and return of external functions (e.g. OS I/O primitives), program exit, semantic failure, and an out-of-memory error, together with internal τ actions:

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

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 (in some sense) equivalent, as the implementation may resolve some of the source-language nondeterminism (c.f. [Sew97]). 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 — some kind of *backward simulation*² result.

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 mutates a return address on its stack, or tries to apply a non-function). We have to distinguish between such semantic errors (modelled with `fail`) and out-of-memory allocation errors (modelled by `oom`) so that we can correctly blame the source program in the former case. Moreover, the compiled program should only diverge, indicated by an infinite trace of τ labels, if the source program can. We express all this with the following notion of backward simulation between a source LTS S and a compiled target LTS T .

DEFINITION 1. A family of relations $R^i : \text{States}(S) \times \text{States}(T)$, indexed by elements i of a well-founded order $>$, is a measured backward simulation if, whenever $s R^i t$ and $t \xrightarrow{ev} t'$ for $ev \neq \text{oom}$, then either

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

Given a measured backward simulation relating S and T , one can easily see that if S has no semantic failures then any (finite or infinite) completed trace of T , that does not include an out-of-memory error, is a trace of S .

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.

CompCert 1.5 is 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 to build some kind of forward simulation for each phase; these can be composed together and combined with determinacy for the target language (PowerPC or ARM assembly) to give a backward simulation for a complete compilation. Forward simulations are generally easier to establish than backward simulations because compiler phases tend to introduce intermediate states;

²Terminology: in CompCert “forward” and “backward” refer to the direction of the simulation with respect to the compiler phases, not to the direction of transitions. Thinking of compilation as “forwards”, a forward simulation means that source behaviours can be matched by the target.

a forward 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.

Decomposing the proof by compiler phases Our compiler is divided into similar (but not identical) phases to CompCert. The above notion of backward simulation also serves as the correctness criterion for each of our phases:

THEOREM 1. *The composition of two measured backward simulations is a measured backward simulation.* [Coq proof]

Labellisation and threadwise proof In our concurrent setting the languages are not deterministic, so the CompCert approach to building backward simulations is not applicable. However, for most of the phases we can re-use the CompCert proof, more-or-less adapted, to give forward 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

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

over configurations that combine a single-threaded program state s and an SC memory m_{SC} (as most sequential language semantics, 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, or the start or exit of the thread:

thread_event, $te ::=$
`ext ev | mem me | τ | start opt_tid p vs | exit`

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

$$s_1 \mid \dots \mid s_n \mid m_{\text{TSO}}$$

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 *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 ClightTSO rule for dereferencing a pointer:

$$\frac{\begin{array}{l} \text{access_mode } ty' = \text{By_value } c \\ \text{typ} = \text{type_of_chunk } c \\ \text{Val.has_type } v \text{ } ty \end{array}}{p \cdot [*_{ty'}] \cdot \kappa_e \mid_{\rho} \xrightarrow{\text{mem (read } p \text{ } c \text{ } v)} v \cdot \kappa_e \mid_{\rho}} \quad \text{LOADBYVALUE}$$

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

This conceptually simple change separates concerns: compiler phases that do not substantially affect the memory accesses of the program can be proved correct per-thread, as described in §5.4 (and those results lifted to the whole system by a general result below), leaving only the two remaining phases that require proofs that really involve the TSO machine.

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.

Lifting threadwise forward simulations to whole-system backward simulations We convert forward threadwise simulations to whole-system backward simulations in two steps. First, we observe that a forward simulation from a determinate language to a receptive language implies the existence of backward 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) l and l' are reads from the same memory location (but not necessarily with the same value), or (ii) l and l' are external returns, or (iii) $l = l'$.

DEFINITION 2. *A thread LTS is receptive if $s \xrightarrow{te} t$ and $te' \asymp te$ implies $\exists t'. s \xrightarrow{te'} t'$.*

DEFINITION 3. *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. *A relation R between the states of two thread LTSs S and T is a threadwise forward 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 $te = \text{fail}$, or $\exists t'. t \xrightarrow{\tau^*} \xrightarrow{te} \xrightarrow{\tau^*} t' \wedge s' R t'$, or $te = \tau \wedge s' R t \wedge s' < s$.*

DEFINITION 5. *A relation R is a threadwise backward simulation if there is a well-founded order $<$ on T such that whenever $t \xrightarrow{te} t'$ and $s R t$, then either $\exists s'. s \xrightarrow{\tau^*} \xrightarrow{te} s' \wedge s' R t'$, or $\exists s'. s \xrightarrow{\text{fail}} s'$, or $te = \tau \wedge s R t' \wedge t' < t$. Moreover, if $t \not\rightarrow$ (t is stuck) and $s R t$, then $s \not\rightarrow$ or $\exists s'. s \xrightarrow{\text{fail}} s'$.*

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

THEOREM 2. *If R is a threadwise forward simulation from S to T , S is receptive, and T is determinate, then there is a threadwise backward simulation that contains R .* [Coq proof]

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

THEOREM 3. *A threadwise backward simulation can be lifted to a whole-system measured backward simulation, for the composition of the threads with the TSO machine.* [Coq proof]

To establish correctness of compiler phases that remove dead variable loads and concretise undefined values, we have also proved variants of Theorems 2 and 3 for suitably modified Definitions 4 and 5.

The two non-threadwise proofs In ClightTSO (as in Clight) local variables are all in allocated blocks, but an early phase of the compiler identifies the variables whose address is 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 heart of our proof, in §5.2 and §5.3.

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 (§5.2), 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 phase, §5.3) compiles away the allocations per function call pre-allocating a thread’s stack when it is created.

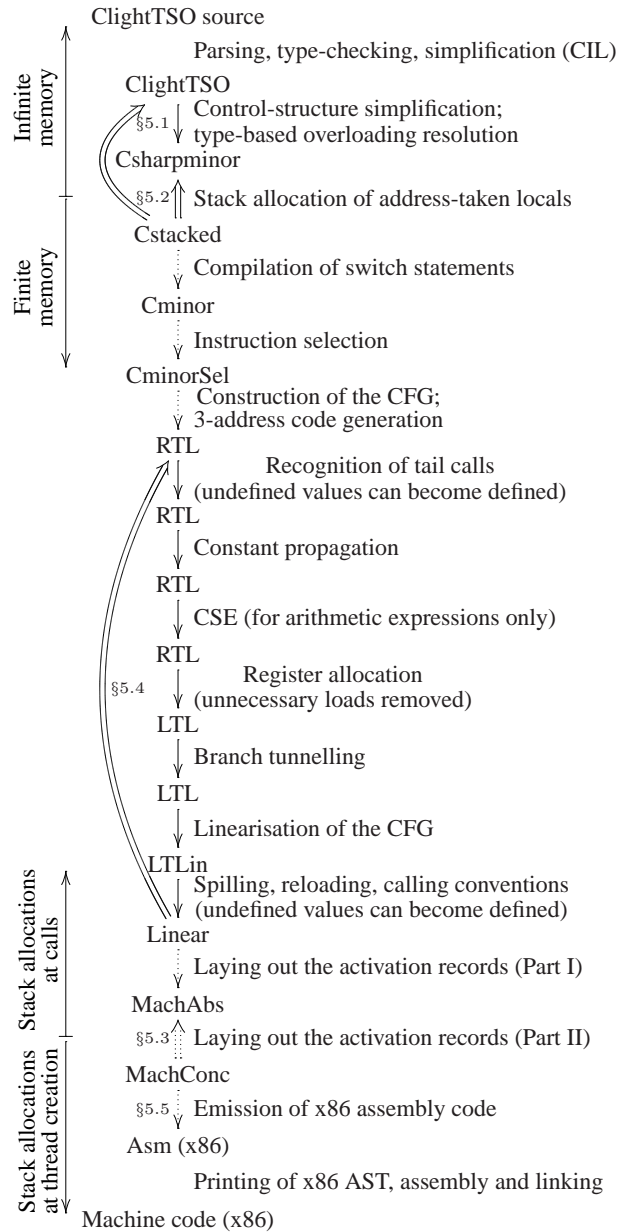
The final phase: targeting x86 We target x86 because x86-TSO gives us a relatively simple and well-understood relaxed memory model for a common multiprocessor. CompCert targets sequential PowerPC and ARM assembly language, but these have much more intricate concurrent behaviour which is still not fully understood (though c.f. [AMSS10]). We therefore need an x86 backend, described in §5.5, but as our focus is on concurrent behaviour rather than sequential computation we make no attempt at optimising the generated code for the x86 instruction set, instead trying to re-use as much as possible.

5. CompCertTSO

Following the strategy above, we have built a working compiler from ClightTSO to x86 assembly language with x86-TSO semantics, and have proved correctness of the most interesting phases. 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. Our development, all mechanised in Coq, is available online³.

The structure of our compiler, and of its proof, is shown in Fig. 3. The subdivision into phases between intermediate languages follows CompCert 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 labelled TSO semantics.



Our proof structure is indicated by single arrows for threadwise forward simulations; and straight double arrows for direct proofs of whole-system backward simulations. Arrows are solid if the proof is completed and dotted otherwise. The completed composite whole-system backward simulations (lifted with Theorems 2 and 3, and composed with Theorem 1) are shown with curved double arrows.

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.

Figure 3. CompCertTSO phases

³ www.cl.cam.ac.uk/users/pes20/CompCertTSO

- 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 backward 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 middle phases of the compiler, from RTL to Linear with various optimisations, are relatively straightforward adaptations of the CompCert proofs to our per-thread labelled semantics and then lifted by the general results of the previous section.
- Our Mach-to-Asm phase generates x86 rather than PowerPC or ARM assembly.

The rest of this section discusses these in more detail. Our main results are as follows.

THEOREM 4. *Given a ClightTSO program, p , and its compilation to a Cstacked program, p' , there is a measured backward simulation between the LTSes of p and p' .* [Coq proof]

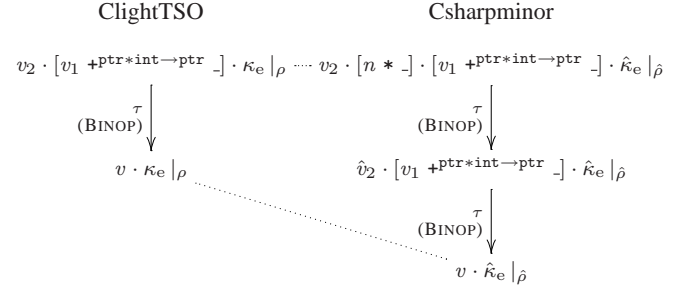
THEOREM 5. *If an RTL program, p , has been successfully compiled to a Linear program, p' , by the following phases: tail call recognition, constant propagation, restricted CSE, register allocation, branch tunnelling, linearisation and reloading, then there is a measured backward simulation between the LTSes of p and p' .* [Coq proof]

Proof outline: First, we construct threadwise forward simulations from ClightTSO to Csharpminor, and between each of the seven phases from RTL to Linear. (More precisely, for the tail call recognition and reloading phases we establish a threadwise forward simulation with undefs, and for the register allocation phase a lock-step threadwise forward simulation with unnecessary load removal.) Then, we turn these threadwise forward simulations to threadwise backward simulations by Theorem 2 (and by the analogous theorems for the threadwise forward simulation with undefs and for the lock-step threadwise forward simulation with unnecessary load removal). Then, by Theorem 3, we turn the threadwise backward simulations into whole-system measured backward simulations. In §5.2, we also establish a measured backward simulation from Cstacked to Csharpminor. Finally, by composing these measured backward simulations according to Theorem 1, we get the overall measured backward simulations.

At the time of writing, the remaining proofs required for a complete verifying compiler include our MachAbs-to-MachConc phase, which involves memory manipulations very similar to those of Csharpminor-to-Cstacked, and the forward simulations for the rest of x86 code generation. All of these have been sketched out (in Coq) in detail. In addition, there is compilation from Cstacked to RTL, which should be straightforward adaptation from CompCert. We are therefore confident that the main intellectual challenges have all been addressed.

5.1 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



where $\text{int} = \text{int} (\text{I32}, \text{Signed})$ and $\text{ptr} = \text{pointer} (\text{int})$. The type annotation in the $*$ context is omitted.

Figure 4. Part of the simulation relating ClightTSO and Csharpminor evaluation for addition of an int and a pointer.

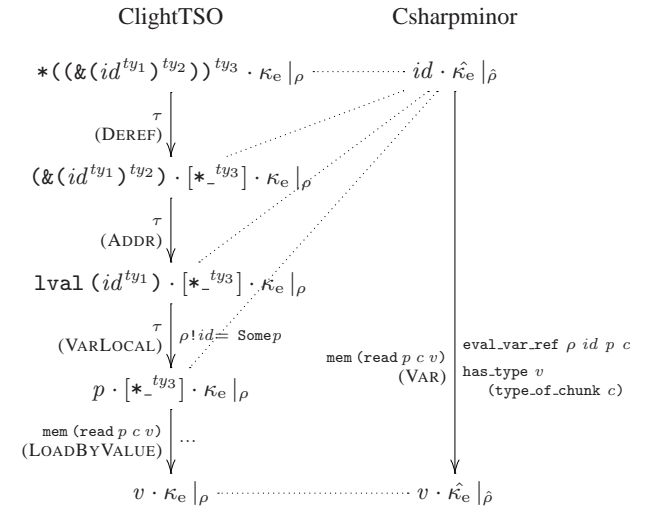


Figure 5. ClightTSO compilation can sometimes eliminate source-level transitions.

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 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 90 rules, while Csharpminor has 56.

Because expression evaluation is defined by a small-step semantics, adapting the forward 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. 4 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. 5. 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, incorporating 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 as 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.

5.2 Changing memory accesses (1) (Csharpminor to Cstacked)

Languages and Compilation The Csharpminor to Cstacked phase bridges the semantic gap to the next intermediate language, Cminor, by introducing a new semantics of 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 com-

piler 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 ones without fall through semantics.

Without the intermediate Cstacked phase, the first two steps change memory semantics: step 1 replaces memory accesses to local variables with local state manipulation that does not touch memory, and step 2 replaces 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:

```
var_kind, vk ::= scalar memory_chunk | array size
cshm_env, cshe ::= nil | (id : (p, vk)) : 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:

```
st_kind, sk ::= local v | stack_scalar memory_chunk ofs
| stack_array size ofs
cst_items, csti ::= nil | (id : sk) : csti
cst_env, cste ::= (p, csti)
```

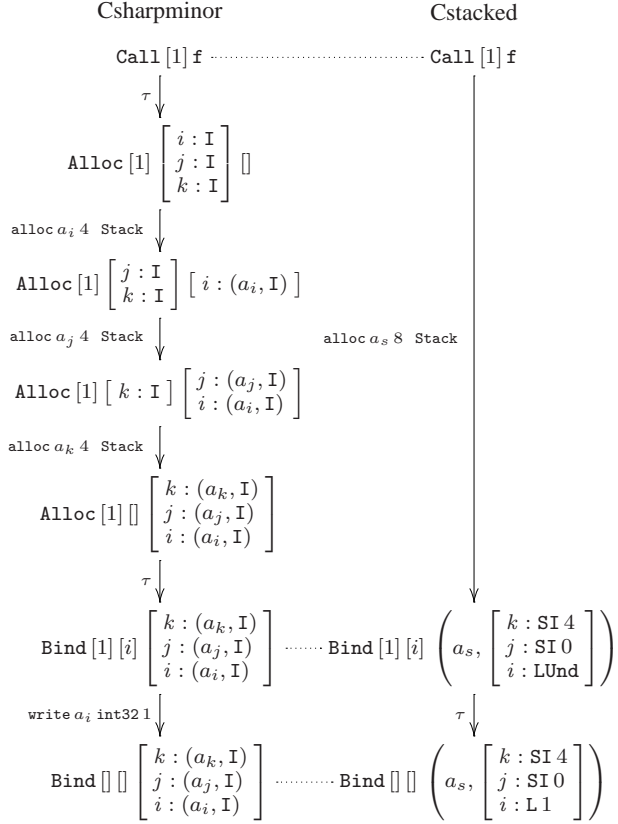
Note that Cstacked can keep values of local variables in the local environment (when the corresponding `st_kind` is `local`). This contrasts with Csharpminor, which stores the values of all local variables in memory.

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. 6 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



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

Figure 6. Function entry transitions in Csharpminor and Cstacked.

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.

Simulating Cstacked in Csharpminor Remember that the Csharpminor-Cstacked phase switches from infinite memory to finite memory. This is necessary to allow the backward simulation to allocate fresh space for the Cstacked local environments in Csharpminor so that the space 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 block is called *scratch space*. Our representation of pointers is of the form (b, ofs) where b is an integer block identifier and $ofs \in \{0, \dots, 2^{32} - 1\}$ 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.

Relating 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 Cshm. Therefore, the thread state simulation must relate Cstacked thread state with 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 Cshm'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 Cshm item's pointer equals the sum of Cstacked stack frame pointer and ofs . Since Cstacked and Cshm only differ in 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.

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; frees in 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 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. 7 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. 6, 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,

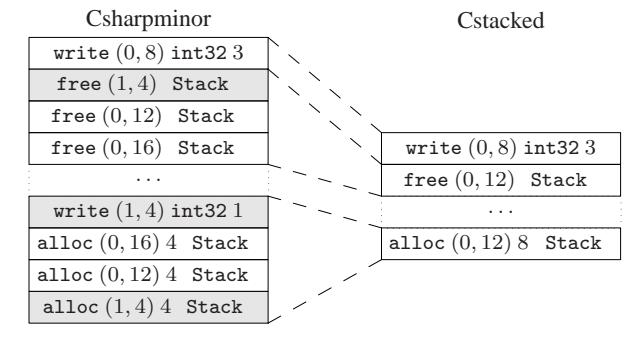


Figure 7. Buffer relation.

or, more precisely, they are the labels introduced by the backward 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 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.

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 Csharpminor⁴.
- 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 scratch pointers only appear as pointers in Csharpminor environments. With these ingredients, the relation on the TSO states is a whole-system backward simulation relation.

5.3 Changing memory accesses (2) (MachAbs to MachConc)

In many respects, the simulation from MachAbs to MachConc is similar to the Csharpminor-Cstacked simulation. MachAbs and MachConc are again two different semantics for the same programs.

In MachAbs, the thread-local state consists of the current function being executed, the program counter, the stack pointer, the register file, the current stack frame, and a sequence of the stack frames for the function’s callers. Instructions that manipulate lo-

⁴ 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.

cal stack variables (`getstack`, `setstack`, `getparam`) perform a 7-step, which accesses only the thread-local stack frames. In contrast, MachConc allocates the stack frames in (global) memory; so the three aforementioned instructions generate read or write events for communicating with the TSO machine.

The proof is by whole-system measured backward simulation, which keeps track of which regions of MachConc’s memory are local for a given thread (corresponding to the thread-local parts of the stack frames) and which regions correspond to the possibly shared parts of each thread’s stack frames. For the thread-local parts, MachConc’s memory is related to the corresponding thread’s frames only after the thread’s buffered updates have been applied, whereas for the shared parts, MachConc’s memory is immediately related to MachAbs’s memory (i.e., before any buffers have been applied). In addition, the MachConc and the MachAbs buffers for each thread are related in an element-wise manner if we ignore the thread-local writes from the MachConc buffers (as they do not correspond to any memory writes in MachAbs). Of course, deciding whether a buffered write is thread-local depends in its position inside the buffer, since preceding buffered allocations and frees can affect whether an address is local or shared. The full simulation relation also relates the states of each thread in the two semantics and contains several administrative properties, such as that the various thread-local and shared stack allocation ranges are pairwise disjoint, and that stack frames must be aligned to 16 byte boundaries.

A distinguishing aspect of this simulation is that we compile away stack-allocations and frees. In MachAbs, each non-empty stack frame is allocated with at a fresh address at function entry, and deallocated at function return. Concretely, however, no memory allocation takes place; the stack pointer is simply incremented or decremented accordingly. Therefore, in MachConc, each thread is allocated a stack when created, and the stack pointer is simply decremented at function entry and incremented at function return (x86 stacks grow downwards). If decrementing the stack pointer exceeds the allocated stack range, the semantics raises an ‘Out of Memory’ error. In concrete x86 executions, this would correspond to a segmentation fault due to stack overflow. Compiling away stack-allocations and frees makes the simulation relation slightly more intricate as the relations between buffers and between memories over the appropriate ranges are of equality on values; the MachAbs values are less defined than the corresponding MachConc ones. This is because a newly allocated stack frame in MachAbs will initially contain `Vundef` everywhere, whereas in MachConc the corresponding block, after decrementing the stack pointer, will contain whatever values happened to be there.

5.4 The ‘easy’ phases, including optimisations

We have enabled all the CompCert 1.5 optimisations that are sound under the TSO semantics. These are: constant propagation & partial evaluation, a restricted version of CSE (common subexpression elimination) that eliminates only common arithmetic expression, but does not eliminate common memory loads, redundant load removal (as part of register allocation), branch tunneling, and tail call optimisation. The only CompCert 1.5 optimisation we do not perform is CSE for memory reads, because this is unsound under the TSO memory model as demonstrated by the following example (adapted from [Pug00]):

```

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

```

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.

Labelling CompCert’s definitions of RTL, LTL, LTLin, Linear, MachAbs, and MachConc and establishing that they are deterministic 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 forward simulation proofs to threadwise forward simulation proofs and lifting them to measured whole-system backward simulations using Theorems 2 and 3 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 backward simulation.) Elimination of redundant loads required a small adaptation of the forward-to-backward simulation infrastructure. Moreover, the tail call optimisation and the 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 2 and 3.

5.5 The x86 backend

Our x86 backend closely follows the structure of the PowerPC backend. We make no attempt at optimising the generated code for the x86 architecture. In particular we do not target any of the rich addressing modes supported by the x86 arithmetic instructions: we want to reuse as much as possible of the structure of the CompCert intermediate languages, including Mach, and Mach operations manipulate only registers. In the Mach-to-Asm compilation phase, we cannot map an operation that manipulates registers to an assembler instruction that accesses a memory location, because below the Mach language it is not known which parts of the memory are thread-local and which are not.

The only parts affected for x86 are the offset computation during the layout of the stack frames, register allocation, and assembler generation. The stack-frame layout has been updated to reflect the x86 MacOSX calling conventions; in addition the stack-pointer of the parent stack-frame is saved just below the function arguments.

The CompCert spilling and reloading phase, despite being well adapted to architectures with many registers like PowerPC, is naive for x86, and requires that some registers are reserved as temporaries for spilling and reloading. A side effect of the CompCert algorithm is that, if all temporaries are stored into stack slots, then after spilling and reloading the generated assembler is 2-address code. We use this property to convert RTL 3-address code to the 2-address code required by the x86 assembler. It would not be difficult to modify the RTLGen phase to directly generate 2-address code, but to make that change worthwhile, the register allocation, and spilling and reloading algorithms should be updated to perform well on an architecture with few registers, and this is not in the scope of this paper. Registers EBX, ECX, and ESI are used as temporaries, and some register shuffling is inserted when generating the assembler to work around idiosyncrasies of x86 assembler (e.g. IDIV and size conversion instructions). For floating point, we use the SSE2 registers and we target the SSE2 instruction set; to respect the calling conventions, the top of the floating point stack is considered as a special register.

5.6 Running CompCertTSO

Despite not making any attempt at optimising the generated code (in particular, RTL registers are systematically reloaded and spilled by the x86 backend), results on simple benchmarks (both sequential and parallel) show that our generated code runs between slightly faster to two times slower than code generated by `gcc -O1`. As a more representative example, we have also successfully compiled Fraser’s lock-free skiplist algorithm [Fra03]; we are roughly 40% slower than `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.

6. 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) for some of the important phases of our compiler.

Phases	Specs	Proofs
TSO machine & memory	2079	2746
Simulations (§4)	1075	1810
ClightTSO definition	2010	186
ClightTSO-Csharpminor (§5.1)	1452	2379
Csharpminor-Cstacked (§5.2)	3481	8208
RTL-Linear (§5.4)	7141	5098
MachAbs-MachConc (§5.3)	910	1934

Of those the RTL-Linear phases are adaptations of existing code (7 phases in total); the rest is largely new. For comparison, CompCert 1.5 has roughly 31K lines of specifications and 23K lines of proofs for all the phases. The project has taken approximately 32 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].

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.

7. 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 [BH09] compilation 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 labelled semantics that we use.

ClightTSO is not intended as a proposal for a complete language: its load and store operations are loosely analogous to the C++0x *atomics* and Java *volatiles*, and it has no distinguished class of memory operations which are supposed to be thread-local (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 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).

8. 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.

Acknowledgements We thank Xavier Leroy for enlightening discussions about this work. We acknowledge funding from EP-SRC grants EP/F036345 and EP/H005633 and ANR grant ANR-06-SETI-010-02.

References

- [AG96] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.
- [AMSS10] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Proc. CAV*, 2010.
- [Bec10] P. Becker, editor. *Working Draft, Standard for Programming Language C++*. March 2010. N3090=10-0080.
- [BH09] N. Benton and C.K Hur. Biorthogonality, step-indexing and compiler correctness. In *Proc. ICFP*, 2009.
- [Boe05] H.-J. Boehm. Threads cannot be implemented as a library. In *Proc. PLDI*, pages 261–268, 2005.
- [CIX] Programming languages – C (committee draft, WG14 N1494, ISO/IEC 9899:201x). <http://www.open-std.org/jtc1/sc22/wg14/www/docs/PostColorado.htm>.
- [Chl10] A. Chlipala. A verified compiler for an impure functional language. In *Proc. POPL*, 2010.
- [CKS07] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *Proc. ESOP*, 2007.
- [Com09] The CompCert verified compiler, v. 1.5. <http://compcert.inria.fr/release/compcert-1.5.tgz>, August 2009.
- [Coq] The Coq proof assistant. <http://coq.inria.fr/>.
- [Fra03] Keir Fraser. *Practical Lock Freedom*. PhD thesis, 2003. Also available as Tech. Report UCAM-CL-TR-639.
- [GM07] G. Gonthier and A. Mahboubi. A small scale reflection extension for the coq system. Technical report, 2007.
- [HAZN08] A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proc. ESOP*, 2008.
- [Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, 1979.
- [Lea99] D. Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
- [Lin99] 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.
- [Loc10] A. Lochbihler. Verifying a compiler for java threads. In *Proc. ESOP’10*, 2010.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.
- [MPA05] J. Manson, W. Pugh, and S.V. Adve. The Java memory model. In *Proc. POPL*, 2005.
- [Myr10] M. O. Myreen. Verified just-in-time compiler on x86. In *Proc. POPL*, 2010.
- [OSS09] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *Proc. TPHOLS*, 2009.
- [Owe10] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *Proc. ECOOP*, 2010.
- [Pug00] W. Pugh. The Java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6), 2000.
- [Sew97] P. Sewell. On implementations and semantics of a concurrent programming language. In *Proc. CONCUR*, July 1997.
- [Spa] The SPARC architecture manual, v. 9. <http://developers.sun.com/solaris/articles/sparcv9.pdf>.
- [Spa92] *The SPARC Architecture Manual, V. 8*. SPARC International, Inc., 1992. Revision SAV080SI9308. <http://www.sparc.org/standards/V8.pdf>.
- [SSO+10] 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.
- [SSZN+09] 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.
- [SZNO+10] 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.
- [TVD10] E. Torlak, M. Vaziri, and J. Dolby. MemSAT: checking axiomatic specifications of memory models. In *PLDI*, 2010.
- [vA08] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, 2008.