Concurrent Permission Machine for modular proofs of optimizing compilers with shared memory concurrency.

Santiago Cuellar

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Andrew Appel

subm. Date

Abstract

Optimizing compilers change a program based on a formal analysis of its code, and modern processors further rearrange the program order. It is hard to reason about such transformations, which makes them a source of bugs, particularly for concurrent shared-memory programs where the order of execution is critical. On the other hand, programmers should reason about their program in the source language, which abstracts such low level details.

We present the Concurrent Permission Machine (CPM), a semantic model for shared-memory concurrent programs, which is: (1) sound for high-order Concurrent Separation Logic, (2) convenient to reason about compiler correctness, and (3) useful for proving reduction theorems on weak memory models. The key feature of the CPM is that it exploits the fact that correct shared-memory programs are permission coherent: threads have (at any given time) noncompeting permission to access memory, and their load/store operations respect those permissions.

Compilers are often written with sequential code in mind, and proving the correctness of those compilers is hard enough without concurrency. Indeed, the machine-checked proof of correctness for the CompCert C compiler was a major advance in the field. Using the CPM to conveniently distinguish sequential execution from concurrent interactions, I show how to reuse the (sequential) CompCert proof, without major changes, to guarantee a stronger concurrent-permission-aware notion of correctness.

Acknowledgements

Acknowledgements...

Dedication.

Contents

List of Tables

List of Figures

Chapter 3

Top to Bottom structure

The goal of this work is to connect the proofs of correctness of a source language, all the way to its correct execution in a real machine. We do this for a rich logic (CSL) to reason about a real language (C, via its high-level intermediate language Clight), through a realistic optimizing compiler (CompCert) and executing on a real weak-cache-consistent multicore processor (X86). Each of these parts has its own intricacies, so we aim to make our work modular.

The programmer shouldn't have to know anything about the compiler or the computer architecture. She wants to use the Dijkstra model of semaphores controlling access to shared data, perhaps using Hoare-style monitors, or using other patterns not limited to a simple mutex. A mutex is a semaphore that is always unlocked by the same thread that locked it; we also support more general synchronization patterns in which one thread acquires a semaphore that some other thread then releases. The programmer also wants to program as if in cooperative concurrency: a thread executes until it performs an explicit synchronization operation (such as semaphore acquire or release), and then some other thread might execute. She should not need to reason about interleavings of instructions. Not only are interleavings difficult to think about, but they are unsound: interleavings connote sequential consistency, and today's machines are not sequentially consistent.

The compiler designer shouldn't have to know anything about the concurrency libraries. She would like to pretend that there's only a single thread, and would like to ignore the nasty problem of weak cache consistency. Similarly, the compiler writer shouldn't need to know the details of the logic used to prove programs correct. He only needs to know a concrete semantics for the compiling languages.

The Concurrent Permission Machine is the connecting thread of this top-to-bottom concurrency result; it provides the semantic interface to modularly compose the three main components of the framework. We model a concurrent program as a list of threads, each with a local-variable (or register-bank) state, each with a permission-map (partial function from address to read/write permission), all sharing a single memory. The machine is equipped with a *schedule* determining the order of thread execution.

The structure of the entire project is as follows:

(top) Any C program, proven correct in Concurrent Separation Logic, runs correctly in the CPM. (Work by Madiot, Mansky, Appel)

(**Compiler**) The CompCert compiler preserves correctness of compiled programs with respect to the CPM semantics. (The contribution of this thesis)

(bottom) Safe executions of the CPM, in the compiled program, run correctly on X86 machines. (Work by Giannarakis and Beringer)

Putting it all together, our main theorem is a top-to-bottom preservation of correctness,

Theorem 3.0.1 (Informal Main Theorem)**.** *A correct C program, compiled by CompCert, runs safely in an X86 machine.*

In the result we present here, we go down as far as assembly language, not machine language; this is because no one has formalized the assembly-to-machine level of CompCert. The preservation of *safety* we prove includes: no execution of undefined instructions or loading from inaccessible memory; no violations of the spinlock well-synchronized property. Owens [29] proved that a spinlock-well-synchronized execution behaves correctly on a TSO (total-store-order) weakly consistent multiprocessor; Giannarakis has generalized Owens's definition of spinlock-well-synchronized to a stronger property that ensures correct execution on other weakly consistent architecture (see section 7.2). Correctness means that any finite prefix of the trace of input-output communications satisfies a specification; the compiler is shown to preserve this traces (up to memory rearrangements by the compiler, section 4.1). In subsection 3.1.1 we formalize how this preservation of traces translates to correctness, but this has not yet been implemented in Coq.

In the rest of this chapter, we will present the details of the main theorem and describe the CPM formally. In chapters 7 and 6, we describe the parts of the top-to-bottom proof that are not developed in this thesis. The entire work has been made public in our tech report [9] and authors who wish to cite the results described in chapters 7 and 6 should cite that report rather than this thesis.

3.1 Main theorem

In this subsection we will describe the main theorem of the top-to-bottom work. We present two top to bottom theorems for reservation of safety and preservation of correctness; the former is mechanized in Coq en the second is formally proven in the following subsection, but has not been mechanized as part of this thesis. We state the theorems in two parts in the first part we state the theorems and the proofs; in the second part, we describe the Coq implementations of the theorem and the technical details of the definitions.

3.1.1 Formal definitions

Safety

The notion of *safety* we preserve means no execution of undefined instructions or loading from inaccessible memory and no violations of the spinlock well-synchronized property.

Theorem 3.0.1 (Main Theorem)**.** *Given a source program P satisfying a CSL specification S, and an x86 assembly language program Q obtained by CompCert compilation of P: Any execution of Q is safe (no undefined behavior), well synchronized, and* correct*—the data transmitted through lock acquire/release events conforms to the resource invariants of S.*

Proof of Main Theorem. The full proof has been mechanized and can be found in the accompanying Coq code. The rest of the thesis expands on the different parts of the proof. □

Correctness

Each part of the top-to-bottom proof (i.e., CSL soundness, compiler correctness and well-synchronization proof) is stronger than preservation of safety and, as we show here, can be composed to show a preservation of correctness. The proof presented below is not mechanized as part of this thesis and is left as future work.

Threads communicate by releasing and acquiring locks that control access to data regions. In terms of Concurrent Separation Logic, we say a program is *correct* if the data in such regions always satisfies an appropriate predicate—the *resource invariant*—when the corresponding locks are released. Therefore, any *observer* of the program—a thread who communicates with it by acquiring and releasing locks—will see only output that satisfies this correctness specification. Because our CSL has the appropriate partial-commutative-monoid ghost variables [19], resource invariants can specify general protocol-correctness properties and not just safety.

We summarize a thread's output by monitoring the data controlled by locks as they are released, and a thread's input via data controlled by locks acquired. The event trace of locks acquired/released, and the contents of memory transferred by those locks, we preserve all the way from the top level (CSL) to the bottom level (assembly) so that we can state correctness properties of the assembly-language execution. We pay particular attention to those global locks defined at the beginning of the program, since their invariants are explicit in the precondition of main (and thus in the specification of the program).s

Theorem 3.1.1 (Correctness preservation)**.** *Let P, S and Q be given as in theorem 3.0.1. For any trace prefix T_Q of an execution of Q, there is an execution of P with trace T_P, which is equal up to some injection ($T_P \overset{j}{\hookrightarrow} T_Q$). In particular, any integer values transferred by releasing locks are equal in both programs.*

Moreover, for every lock l, defined in the precondition of main (in S) with some invariant R, every time l is realeased (in T_P and T_Q), the transferred resources satisfy R.

Proof. We look at how every part of the top-to-bottom work preserves traces (up to injection).

The instruction interleaving proof in section 7.1 constructs a new coarse-grain schedule and a new execution but preserves all synchronization operation in the same order. The transferred resources of each synchronization oparation are also the same up to some reordering of memory.

```
1  Theorem top2bottom_correctness:
2      (* C program is proven to be safe in CSL*)
3      ∀ (main:AST.ident), CSL_correct C_program main →
4
5      (* C program compiles to some assembly program*)
6      CompCert_compiler C_program = Some Asm_program →
7
8      (* Statically checkable properties of ASM program *)
9      ∀ (STATIC: static_validation Asm_program main),
10
11     (* For all schedules, *)
12     ∀ U : schedule,
13
14     (*The asm program can be initialized with a memory and CPM state*)
15     ∃ (m : mem) (cpm : CPM),
16       initial_state Asm_program STATIC cpm m ∧
17
18       (* The assembly language program
19        is correct and well-synchronized *)
20       spinlock_safe U cpm m.
```

Figure 3.1: Coq definition of our main theorem

The CPM simulations definition 5.1.1, given by the the compile correctness specification, preserve synchronization operation in the same order and the transferred resources are equal up to the injection given by compilation.

The CSL proof, that P satisfies S, includes proving that every lock release satisfies the invariant of the lock. For any lock defined in the precondition of main that invariant is exactly the one in S. □

3.1.2 Coq definitions

Figure 3.1 shows the Coq statement of our main theorem 3.0.1. We show how the code implements the theorem and explain the code below:

In what follows we explain how the code in Figure 3.1 implements Theorem 3.0.1.

- "Given a source program P satisfying a CSL specification S ..." (lines 2-3):

CSL_correct C_program: States that the program has been proven correct in CSL for some specification written in concurrent separation logic (starting at some main function main) as described in chapter 6.

- "... and an x86 assembly language program Q obtained by CompCert compilation of P ..." (lines 5-9)
  - CompCert_compiler: States that the CompCert compiler translates C_program into the assembly program Asm_program.
  - static_validation: We validate the translation by statically checking a couple properties of the translated program Asm_program. All of them are known to be preserved by the compiler, but the fact that CompCert preserves them has not yet been proven in Coq. We leave removing these conditions as future work. These properties are:
    - limited_builtins: The program only uses the builtins we currently support: memcopy, mem_alloc, mem_free. These are the only builtins that CompCert inserts during compilation.
    - valid_mem: The initial memory has no dangling pointers.
    - ge_wd: The global environment is allocated in the initial memory.
    - main_ident_correct: The assembly program has an entry function named main.
- "...Any execution of Q is safe (no undefined behavior), well synchronized and correct." (lines 11-20)
  - initial_state: The program can be initialized with the initial memory m and the initial CPM cpm.
  - spinlock_safe: For all executions (and for all schedules, quantified in line 11), the initial state is safe and well synchronized, as defined in section 7.2.

3.2 The concurrent permission machine

The Concurrent Permission Machine is a language-agnostic operational semantics for concurrent programs. We use the CPM as our interface, at the C source level, between *proofs of source program properties* and *the operational semantics of the programming language*; we use it as the operational model that allows *modular reasoning about compiler proofs*; and we use it as our interface, at the x86 assembly language level, between *permission safety* (derived from those proofs) and the *well-synchronized* property.

3.2.1 Overview

The main idea of a Concurrent Permission Machine is a simple one: annotate the small-step operational semantics–of C, of assembly language, or of any intermediate language–with per-thread permissions at every address. A thread is *stuck* if it tries to read address a without read permission, or write without write permission. No two threads ever have conflicting permissions to the same address, so races are impossible. Acquiring a lock *increases* the thread's permissions at some set of addresses; releasing a lock *decreases* permissions. Which permissions are increased or decreased? Intuitively, the semaphore controls access to some shared data; it is those addresses. Of course, these permissions have no physical manifestation in the execution of a machine-language program, so we prove an erasure theorem. But we must wait until the very bottom level (of our proof) before erasing them, so as to be able to prove the kind of race freedom necessary on a weak-cache-consistent multicore processor.

We model a concurrent program as a list of threads, each with a local-variable (or register-bank) state, each with a permission-map (partial function from address to read/write permission), all sharing a single memory. The program is equipped with a *schedule* determining the order of thread execution.

To enable instantiations to C and assembly (or other languages), CPMs are built parametrically on top of *interaction semantics* [5], a common abstraction for languages operating over shared CompCert memories, each with its own code representation, thread-local state, and operational relation for internal steps (To support our proofs, we further developed Beringer et al.'s interaction semantics into MOIST semantics; see chapter 4). For CompCert's languages (i.e., C and Asm), we use $\Psi \vdash_{\text{CompCert}} \langle\sigma, m\rangle \overset{\epsilon}{\mapsto} \langle\sigma', m'\rangle$ to refer to this small-step relation, where Ψ is the program, σ and σ' are thread-local states (a thread-local variable store in case of C, a register bank in case of assembly), m, m' are CompCert memories, and ϵ is a (possibly empty) trace of events that denote the nonatomic memory accesses performed. We use it to state theorems about the absence of races between nonatomic accesses. Thread-local execution proceeds ad infinitum, reaches a *Halted* state, or reaches an external function call. In the latter case—marked at_external—the CPM takes control, for example by executing a concurrency primitive (see below), and scheduling other threads. Internal execution is resumed by the primitive after_external, decorated by the (optional) return value.

The CPM supports external function calls for dynamic creation of threads and locks; thus we have a subset of C11 concurrency. A program can make these external function calls:

```
struct lock;
void spawn (void *func, void *arg); /* spawn a thread */
void makelock(struct lock *p); /* initialize a lock */
void freelock(struct lock *p); /* decommission a lock */
void acquire (struct lock *p); /* acquire a lock */
void release (struct lock *p); /* release a lock */
```

To specify the gain or loss in access rights that a thread experiences when making such external calls, CompCert and CPMs mark locations with *permissions*—

instrumentation values that may take (in increasing order of permissiveness) levels None < Nonempty < Readable < Writable < Freeable [25]. In fact, CompCert's memory model associates permissions to each location that indicate the running thread's current access rights. A thread's small-step relation is *stuck* whenever a memory operation (such as load, store, free) is not supported by the thread's permission at the location in question. Thread-local execution does not alter existing permissions, with the exception of lowering it to None during stack frame deallocation. In contrast, external calls (and specifically, the concurrency primitives) may modify the permissions arbitrarily (though not greater than the max permissions). The CPM regulates permission transfers between different threads, maintaining *permission coherence* (in particular: the absence of conflicting write permissions by different threads), but also preserving safety of individual threads as these are compiled.

Example. Before moving on to the precise definitions of the CPM, let's attempt a first approximation of how the rule for Release should work. We review this example in the next section and the correct rule is shown in fig. 3.3.

The multithreaded CPM is guided by a *schedule* $\mho$; we write $i \cdot \mho$ for a schedule in which the ith thread is next to execute. If there are (so far) k spawned threads, the machine maintains a list $\vec{s}$ of k thread-states (local-variable sets) and a list $\vec{\pi}$ of k permission-maps; all threads share the memory m. To release a lock a, the current thread i must be at_external, ready to call Release with argument a. The machine must make sure that the lock is currently locked ($m(a) = 0$) and unlock it ($m[a \mapsto 1] = m'$).

$$\frac{\begin{array}{c}\text{at\_external } s_i = \text{Some}(\text{Release}, a) \\ m(a) = 0 \qquad m[a \mapsto 1] = m' \\ \text{guess } \delta \qquad \vec{\pi}[i \mapsto \delta/\pi_i] = \vec{\pi}'\end{array}}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}), m\rangle \mapsto \langle \mho, (\vec{s}', \vec{\pi}'), m'\rangle}$$

(SIMPLIFIED RELEASE)

What permissions should the release transfer? Suppose a controls access to the linked list rooted at address p. Release should give up all permissions to that linked list—they are transferred to the lock. Unfortunately, different linked lists can be stored at p, so the data protected by a can be different every time the lock is released. In concurrent separation logic, we describe this with a resource invariant R; in any given memory, only one set of addresses[1] can satisfy R; this determines which addresses to transfer. In the top-to-bottom proof an oracle is constructed from the CSL proof. In the CPM the data is given by the oracle as a guess δ. The CPM updates the permissions of the current thread, to contain the old permissions updated with the permissions transfered δ/π_i.

3.2.2 Formal definition of the CPM

The CPM operates over states of the form $\langle \mho, (\vec{s}, \vec{\pi}, L), m\rangle$, where

$\mho$ is a (cooperative) schedule, a finite sequence of natural numbers indicating which thread to run next. We approximate infinite computations by quantifying over all finite prefixes.

$\vec{s}$ is a list of marked local states of the CompCert language the CPM is instantiated by; $s_i \in \{\text{Start}(f, a), \text{Run}(\sigma_i), \text{Blocked}(\sigma_i), \text{Resume}(v_i, \sigma_i)\}$, where σ_i is the ith thread's local state. Start(f, a) denotes that the thread should start

[1]In general we do not require *precise* resource invariants, but the linked-list predicate happens to be precise.

$$\frac{\begin{array}{c} s_i = \text{Blocked}(\sigma) \quad \text{at\_external}\,\sigma = \text{Some}(\text{Acquire}, a) \quad m|_{\pi_i^2}(a) = 1 \\ m[a \mapsto 0] = m' \quad \text{guess } \delta \quad \delta/\pi_i = \pi' \quad \pi_i \oplus L(a) = \pi' \\ \vec{s}[i \mapsto \text{Resume}(0,\sigma)] = \vec{s}' \quad \vec{\pi}[i \mapsto \pi'] = \vec{\pi}' \\ L[a \mapsto \text{Some } \emptyset] = L' \end{array}}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \overset{\text{Acq}_i\, a\, \delta}{\mapsto} \langle \mho, (\vec{s}', \vec{\pi}', L'), m'\rangle} \tag{\textsc{acquire}}$$

$$\frac{s_i = \text{Blocked}(\sigma) \quad \text{at\_external}\,\sigma = \text{Some}(\text{Acquire}, a) \quad m|_{\pi_i^2}(a) = 0}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \overset{\text{AcF}_i\, a}{\mapsto} \langle \mho, (\vec{s}, \vec{\pi}, L), m\rangle} \tag{\textsc{acqfail}}$$

$$\frac{\begin{array}{c} s_i = \text{Blocked}(\sigma) \quad \text{at\_external}\,\sigma = \text{Some}(\text{Release}, a) \quad m|_{\pi_i^2}(a) = 0 \\ m[a \mapsto 1] = m' \quad \vec{s}[i \mapsto \text{Resume}(0,\sigma)] = \vec{s}' \quad L(a) = \emptyset \\ \text{guess } \delta \quad \text{guess } \delta_{\text{L}} \quad \delta_{\text{L}}/\emptyset \oplus \delta/\pi_i = \pi_i \\ \vec{\pi}[i \mapsto \delta/\pi_i] = \vec{\pi}' \quad L[a \mapsto \text{Some}(\delta_{\text{L}}/\emptyset)] = L' \end{array}}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \overset{\text{Rel}_i\, a\, \delta\, \delta_{\text{L}}}{\mapsto} \langle \mho, (\vec{s}', \vec{\pi}', L'), m'\rangle} \tag{\textsc{release}}$$

$$\frac{\begin{array}{c} s_i = \text{Blocked}(\sigma) \quad \text{at\_external}\,\sigma = \text{Some}(\text{mkLock}, a) \quad \vec{s}[i \mapsto \text{Resume}(0,\sigma)] = \vec{s}' \\ \pi_i[a \mapsto (\text{Nonempty}, \text{Writable})] = \pi_i' \quad \vec{\pi}[i \mapsto \pi_i'] = \vec{\pi}' \\ L(a) = \emptyset \quad L[a \mapsto \text{Some None}] = L' \quad m|_{\pi_i^2}[a \mapsto 0] = m' \end{array}}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \mapsto \langle \mho, (\vec{s}', \vec{\pi}', L'), m'\rangle} \tag{\textsc{make lock}}$$

$$\frac{\begin{array}{c} s_i = \text{Blocked}(\sigma) \quad \text{at\_external}\,\sigma = \text{Some}(\text{freeLock}, a) \quad \vec{s}[i \mapsto \text{Resume}(0,\sigma)] = \vec{s}' \\ \text{guess } p \quad \pi_i[a \mapsto (p, \text{None})] = \pi_i' \quad \vec{\pi}[i \mapsto \pi_i'] = \vec{\pi}' \\ L(a) = \text{Some None} \quad L[a \mapsto \text{None}] = L' \end{array}}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \mapsto \langle \mho, (\vec{s}', \vec{\pi}', L'), m'\rangle} \tag{\textsc{free lock}}$$

$$\frac{\begin{array}{c} s_i = \text{Blocked}(\sigma) \quad \text{at\_external}\,\sigma = \text{Some}(\text{Spawn}(f,a)) \quad |\vec{s}| = j \\ \vec{s}[i \mapsto \text{Resume}(0,\sigma),\, j \mapsto \text{Start}(f,a)] = \vec{s}' \\ \text{guess } \delta \quad \text{guess } \delta' \quad \delta/\pi = \pi' \\ \delta' \oplus \pi' = \pi \quad \vec{\pi}[i \mapsto \pi',\, j \mapsto \delta'/\{\}] = \vec{\pi}' \end{array}}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \overset{\text{Spa}_{ij}}{\mapsto} \langle \mho, (\vec{s}', \vec{\pi}', L), m\rangle} \tag{\textsc{spawn}}$$

Figure 3.2: Concurrent Permission Machine, synchronization steps.

$$\frac{s_i = \text{Start}(f,a) \qquad \vec{s}' = \vec{s}[i \mapsto \text{Run}(\text{initialCore}(f,a))]}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \mapsto \langle i \cdot \mho, (\vec{s}', \vec{\pi}, L), m\rangle} \quad (\textsc{start})$$

$$\frac{s_i = \text{Resume}(v,\sigma) \qquad \text{afterExternal}(\sigma, v) = \sigma' \qquad \vec{s}' = \vec{s}[i \mapsto \text{Run}(\sigma')]}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \mapsto \langle i \cdot \mho, (\vec{s}', \vec{\pi}, L), m\rangle} \quad (\textsc{resume})$$

$$\frac{\begin{array}{c} s_i = \text{Run}(\sigma) \qquad \Psi \vdash_{\text{CompCert}} \left\langle \sigma, m|_{\pi_i^1} \right\rangle \overset{\epsilon}{\mapsto} \langle \sigma', m'\rangle \\ \vec{s}' = \vec{s}[i \mapsto \text{Run}(\sigma')] \qquad \vec{\pi}' = \vec{\pi}[i \mapsto (\text{Cur}(m'), \pi_i^2)] \end{array}}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \overset{\epsilon_i}{\mapsto} \langle i \cdot \mho, (\vec{s}', \vec{\pi}', L), m'\rangle} \quad (\textsc{core})$$

$$\frac{s_i = \text{Run}(\sigma) \qquad \text{at\_external } \sigma = \text{Some}(f, \vec{x}) \qquad \vec{s}' = \vec{s}[i \mapsto \text{Blocked}(\sigma)]}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \mapsto \langle \mho, (\vec{s}', \vec{\pi}, L), m\rangle} \quad (\textsc{suspend})$$

$$\frac{(s_i = \text{Blocked}(\sigma) \wedge \text{at\_external } \sigma = \text{Some}(\text{Exit}, \_)) \ \vee \ \neg(0 \le i < |\vec{s}|)}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \mapsto \langle \mho, (\vec{s}, \vec{\pi}, L), m\rangle} \quad (\textsc{stutter})$$

$$\frac{}{\Psi \vdash_{\text{CPM}} \langle \text{nil}, (\vec{s}, \vec{\pi}, L), m\rangle \mapsto \langle \text{nil}, (\vec{s}, \vec{\pi}, L), m\rangle} \quad (\textsc{done})$$

Figure 3.3: Concurrent Permission Machine: administrative and internal steps.

by calling function f with arguments a; Run(σ_i) denotes that the thread is in the middle of sequential execution; Blocked(σ_i) denotes that the thread has called a synchronization primitive, and is waiting for the CPM to execute it; Resume(v, σ_i) means that an external call has returned value v to be fed back to the thread. A halted thread (one that has returned from its initially spawned function) is recognized by a *halted* predicate.

$\vec{\pi}$ is a list of permission map pairs, one pair for each thread (the components of these pairs are described below).

L is a function from address to option(option(permission)), indicating the state of each lock: $L(a) = \text{None}$ means that a is not a lock. Some(None) means that a is locked—permissions associated with the lock a are hence installed in the Cur component of m. Some(Some π) means a is unlocked and π is the permission that a thread would obtain by acquiring a.

m is the global memory, shared by all threads.

The CPM employs judgments $\Psi \vdash_{\text{CPM}} \langle \mho, (\vec{s}, \vec{\pi}, L), m \rangle \overset{\epsilon}{\mapsto} \langle \mho', (\vec{s}', \vec{\pi}', L'), m' \rangle$, as shown in fig. 3.2 and fig. 3.3.

Without $\vec{\pi}$ and L, it would be a rather conventional model of a cooperative-concurrent machine, with threads $\vec{s}$ operating on a memory m. The type of thread-states s contains local variables and control-stack. The initial state has a single thread (at the beginning of the main function) and an empty lock pool.

Each thread s_i has its own pair π_i of (finite) permission maps: for each address l, $\pi_i(l)$ contains a *data* permission $\pi_i^1(l)$ and a *lock* permission $\pi_i^2(l)$. When scheduled, the thread's data permissions are installed as the current permission in CompCert's memory (operation $m|_{\pi_i^1}$ in rule core), thus regulating the dynamic access to shared locations.[2] In contrast, the thread's lock permissions are never installed in the CompCert memory of a running thread, but employed by the CPM to decide whether a thread's lock requests can be granted; in this context, Readable permission grants "permission to acquire/release" and Writable means "permission to decommission (freelock) the lock back into ordinary data". In the context of both data and lock permissions, Readable can be thought of as "Shared" and Writable as "Exclusive": data is read-only when shared and can be written when held exclusively, while locks can be acquired and released when shared and can only be turned back into data when held exclusively.

[2]The installation of permission maps happens at the granularity level of instructions / individual execution steps; this permits the reuse of the CPM framework in the setting of fine-grained interleaving, as described in section 7.1.

The lock pool L maps each address a to either None (not a lock), Some($\emptyset$) (when the lock is locked) or Some(π) (when the lock is unlocked and holds a resource whose permission map is π).

As in O'Hearn's original CSL [28], resource invariants R specify the memory regions (and assertions on them) regulated by locks. As in Gotsman *et al.* [12] and Hobor *et al.* [16], locks are dynamically creatable at addresses in memory; as in Hobor et al., resources are higher-order in that they may predicate over other locks and their resource invariants, or even over this lock and its resource invariant. To support higher-order impredicative resource invariants, we use a step-indexed model of CSL, but we want to avoid putting step-indexed resource invariants into the CPM itself. To avoid the use of higher-order resource invariants in the CPM, the CPM uses guesses δ, *partial permission maps* that specify which new permissions should be installed for every address in their domain. We write δ/π to mean a permission map in which δ "overrides" π. Like π, each δ is a pair of *data* and *lock* permission, at each address.

To find δ without knowing R (in particular: concurrency operations operating on the same lock may yield different permission transfers at different points in the execution, as footprints are dynamic), the CPM uses *angelic nondeterminism.* A program is safe if there exists any sequence of angelic guesses δ such that the program does not get stuck. Chapter 6 describes how we prove that this sequence of guesses must exist: in particular, any CSL proof entails a sequence of guesses for which the CPM is nonstuck.

In two nonstuck executions of $\vdash_{\text{CPM}}$ from the same start state but with different guesses δ, the values loaded and stored (and passed as parameters to functions) are the same. Guesses δ affect only the permissions of the memory. Insufficient permission can cause a stuck state, but extra permissions do not change values loaded or stored.

Therefore the angelic nondeterminism does not affect any observable property of a nonstuck execution.

The operator $m|_{\pi_i^1}$ sets the current permissions of m to the *data* permission-map of the ith thread; respectively, $m|_{\pi_i^2}$ sets m to the *lock* permission-map.

We write $\pi \oplus \pi' = \pi''$ to indicate a kind of *join* on permission maps; it's a relation, not a function, because Readable⊕Readable could "add up" to Readable or Writable.

The annotation ϵ is an *event trace*, recording nonatomic memory operations, acquires and releases of locks, creation and destruction of locks, and creation of threads. Later, we'll use this ordered sequence of operations to define *well-synchronized* executions.

Example. Following up from the example in the previous section, we address the changes in the RELEASE, from the simplified version presented before: the ith thread's state is Blocked(σ), which signals to the machine that the external function has not been executed; when loading from m we adjust the current permissions of m to make the lookup permissible according to the threads' lock permissions (π_i^2); the CPM signals that the RELEASE has been executed by setting the ith thread to Resume($0, \sigma$); we also have to guess δ_{L} representing the new resource held in the pool of unlocked resources; we make sure that the new permissions in the lock pool ($\delta_{\mathrm{L}}/\emptyset$) and in the thread ($\delta/\pi_i$) "add up" to the old permissions in the thread (π_i); we require that the guessed $\delta, \delta_{\mathrm{L}}$ have not caused competing permissions; we modify L at address a to hold $\delta_{\mathrm{L}}/\emptyset$.

One might think that δ must uniquely determine δ_{L}; why do we need both? Remember that $\pi \oplus \pi' = \pi''$ is notation for a relation, which is not deterministic (two Readable permissions may add up to either a Readable or a Writable permission), so we use $\delta, \delta_{\mathrm{L}}$ to cope with this bit of nondeterminism.

Permission coherence

We say coherent($\vec{\pi}, L, m$) when π_i and π_j (or π_i and $L(l)$, etc.) do not give *competing* permissions at any address, nor treat any address as both *data* and *lock*. For example, $\pi_i^1(a) =$ Writable means thread i can do nonatomic reads/writes to address a; this is compatible with $\pi_j^1(a) \leq$ Nonempty (for $i \neq j$), meaning that thread j "knows that a is allocated," but not with $\pi_j^1(a) =$ Readable, nor with $\pi_j^2(a) \neq$ None. That is, if thread i sees a as a data location, then thread j cannot see the same address as a lock. Every execution of the CPM maintains coherence as an invariant.

Definition 3.2.1 (Competing permissions)**.** *Two permissions in the CompCert permission lattice, compete iff:*

1. *one of them is Freeable and the other is Nonempty or higher; or*

2. *one of them is Writable and the other is Readable or higher.*

Intuitively, Freeable means "no other thread even knows this data is allocated, so I can free it." Nonempty means "I know this data is allocated, even though I can't read or write it." With Nonempty *data* permission one can at least do pointer-equality tests; in C11, these are defined only on addresses of allocated data.

Definition 3.2.2 (Data-Lock coherent)**.** *Two permissions in the CompCert permission lattice, respect the data-lock invariant iff: The lock permission is not Freeable; and if the lock permission is ≥Readable then the data permission ≤Nonempty.*

A Readable *data* permission gives permission to read; a Readable *lock* permission gives permission to (attempt to) acquire the lock. To perform makelock, which converts a data block to a lock, one needs at least a *data* permission of Writable; the makelock takes away the Writable *data* permission (leaving either None or Nonempty *data* permission) and grants Writable *lock* permission. In turn, that *lock* permission can be split into several Readable parts, to be granted to various threads that want to contend for the lock. Eventually, these might be gathered together by a single thread into a Writable *lock* permission, which is enough to perform freelock, which converts a Writable *lock* permission to a Writable or Freeable *data* permission.

We lift definition 3.2.1 and definition 3.2.2 to permission-maps π. Recall that we write π_i^1 for thread i's *data* permission-map, and π_i^2 for thread i's *lock* permission-map. We write $L(a)^1$ for the *data* permission-map of the unlocked lock at address a, and $L(a)^2$ for its *lock* permission-map. Locked locks have empty permission-maps, since all their permissions have been (temporarily) added to the permission-map of the thread that acquired the lock.

Definition 3.2.3 (Coherent[3]). *We say that a state of the* CPM *machine is* coherent*, written as* $\text{coherent}(\vec{\pi}, L, m)$ *iff:*

1. *For all* i*,* $\pi_i^1 \leq \text{Max}(m)$ *and* $\pi_i^2 \leq \text{Max}(m)$
2. *For all* $L(a)$*,* $L(a)^1 \leq \text{Max}(m)$ *and* $L(a)^2 \leq \text{Max}(m)$
3. *For all* i, j*, permission-maps* π_i^1 *and* π_j^2 *are data-lock coherent; and if* $i \neq j$*, then* π_i^1 *and* π_j^1 *do not compete and* π_i^2 *and* π_j^2 *do not compete.*
4. *For all* $L(a)$ *and* π_i*,* $L^1(a)$ *and* π_i^2 *are data-lock coherent,* π_i^1 *and* $L^2(a)$ *are data-lock coherent,* π_i^1 *and* $L^1(a)$ *do not compete, and* $L^2(a)$ *and* π_i^2 *do not compete.*
5. *For all* a, b*,* $L^1(a)$ *and* $L^2(b)$ *are data-lock coherent; and if* $a \neq b$ *then* $L^1(a)$ *and* $L^1(b)$ *do not compete, and* $L^2(a)$ *and* $L^2(b)$ *do not compete.*

Safety

We use two notions of safety. First the intuitive one is as follows

[3]The Coq implementation details of this definition can be found in appendix A.3.1

Definition 3.2.4 (Weak CPM Safety)**.** *A CPM state is* safe for k steps*, written* safe$'_k$*, when there exists a sequence of angelic guesses δ such that the machine does not get stuck within k steps.*

Using a schedule $\mho$ is a useful fiction, however oftentimes we like to express that the program safety does not depend on the schedule. It is not enough to quantify Weak CPM Safety over all possible schedules, since we want our safety to be preserved by execution. We need a slightly stronger notion of safety that quantifies over all schedules after every step

Definition 3.2.5 (CPM Safety)**.** *A CPM state is* safe for k steps*, written* safe$_k$*, when there exists a sequence of angelic guesses δ such that the machine does not get stuck within k steps, even when it's allowed to change the tail of its schedule.*

Since a safe execution makes the "right angelic guesses", we can prove the following lemma

Lemma 3.2.6 (Safety implies coherence)**.** *If a program is safe for all k, then every state in the execution is coherent.*

3.2.3 Generality of the CPM

The C11 standard defining the behavior of concurrent C programs has the following design principles:

- Memory locations can be classified as either nonatomic (accessed by normal operations in a well-synchronized manner) or atomic (accessed by synchronization operations, not necessarily well-synchronized).
- Compilers should optimize nonatomic operations.
- Ill-synchronized accesses to nonatomic locations (i.e., data races) lead to undefined behavior.

- Between synchronization operations, code in a thread is executed as if it were sequential.
- Correctly written code should execute correctly on any processor, regardless of its relaxed memory model.

The CPM gives an operational semantics for concurrent programs that obeys these principles, making it well suited for modeling lock-based concurrent programs in C and any other language that adheres to the same principles. The CORE rule of fig. 3.3 incorporates the sequential semantics of the language into the machine, and the remaining rules give semantics to the atomic operations. Because the CPM uses co-operative scheduling, the behavior of a thread between synchronization operations is exactly its sequential behavior, and compilers can optimize code in between synchronizations as if it were sequential.

In chapter 6 we show that the CPM serves as an operational model for Concurrent Separation Logic. But it models permission coherence more generally than just CSL. For example, Gu *et al.* describe a refinement method for proving correctness of shared-memory concurrent programs [14] with a "push/pull" model for releasing and acquiring locks. The *push* and *pull* can be modeled as angelic permission transfers in the CPM while it is unknown whether they can be modeled in concurrent separation logic.

To demonstrate that our CPM is more general than our specific concurrent separation logic, we show an example of a program that our CPM can model even though our CSL front-end cannot model it. The logic presented in chapter 6, and many others similar concurrent logics (e.g., [28], [16]), will fail to prove the correctness of the following example:

```
(thread 1)          (thread 2)
p[0]=x;             if (?)
release (A);        then acquire (A);
release (B);        else acquire (B);
                    y=p[0];
```

Some CSLs fail to reason about this example because we don't know whether the resource p[0] is transferred through lock A or lock B. Some CSLs with ghost state can prove the program correct: in fact, as Jung *et al.* [19] have shown, ghost state can be used to incorporate general rely-guarantee reasoning into CSL, by creating a more general notion of invariant that is not tied to atomic accesses to a specific location. In this case, the `release` and `acquire` rules must be interpreted as a different kind of atomic access, one that allows interaction with the global invariant. The above example can also be proven correct in a rely-guarantee logics such as VCC [8] or Iris [19]. Regardless of the means used to verify the program, it can still be executed in the CPM: for each choice of lock, there exists a CPM execution that performs the appropriate permission transfer.

The current presentation of the CPM is limited to Semaphores (coarse grain synchronization); however, we believe that it can be extended to support other C11 atomic operations in modes such as SC and release/acquire

Bibliography

[1] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 2–14, New York, NY, USA, 1990. ACM.

[2] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge, 2014.

[3] Andrew W. Appel, Paul-Andre Melliès, Christopher D. Richards, and Jerôme Vouillon. A very modal model of a modern, major, general type system. In *Proc. 34th Annual Symposium on Principles of Programming Languages (POPL'07)*, pages 109–122, January 2007.

[4] Mark John Batty. *The C11 and C++11 Concurrency Model*. PhD thesis, University of Cambridge, 2014. 2015 SIGPLAN John C. Reynolds Doctoral Dissertation award and 2015 CPHC/BCS Distinguished Dissertation Competition winner.

[5] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. Verified compilation for shared-memory C. In *European Symposium of Programming*, Lecture Notes in Computer Science, pages 107–127. Springer, 2014.

[6] Hans-J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–268, New York, 2005.

[7] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *J. Autom. Reasoning*, 61(1-4):367–422, 2018.

[8] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: a practical system for verifying concurrent C. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), LNCS 5674*, 2009.

[9] Santiago Cuellar, Nick Giannarakis, Jean-Marie Madiot, William Mansky, Lennart Beringer, Qinxiang Cao, and Andrew W. Appel. Compiler correctness for concurrency: from comncurrent separation logic to shared memory assembly language. Technical Report TBD, Department of Computer Science, Princeton University, 2020.

[10] Robert W. Dockins. *Operational refinement for compiler correctness.* PhD thesis, Princeton University, July 2012.

[11] Mike Dodds, Mark Batty, and Alexey Gotsman. Compositional verification of compiler optimisations on relaxed memory. In *European Symposium on Programming*, pages 1027–1055. Springer, 2018.

[12] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *Proceedings 5th Asian Symposium on Programming Languages and Systems (APLAS'07)*, 2007.

[13] Ronghui Gu, Jeremie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Newman Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *POPL*, pages 595–608. ACM SIGPLAN, 2015.

[14] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*, pages 646–661, June 2018.

[15] Aquinas Hobor. *Oracle Semanatics.* PhD thesis, Princeton University, Princeton, NJ, November 2008.

[16] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle Semantics for Concurrent Separation Logic. In *ESOP*, pages 353 – 367, 2008.

[17] Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A theory of indirection via approximation. In *Proc. 37th Annual ACM Symposium on Principles of Programming Languages (POPL'10)*, pages 171–185, January 2010.

[18] Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. Towards certified separate compilation for concurrent programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.*, 2019.

[19] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *42nd ACM Symposium on Principles of Programming Languages (POPL'15)*, pages 637–650. ACM, 2015.

[20] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in iris. *In 31st European Conference on Object-Oriented Programming (ECOOP 2017)*, 2017.

[21] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. *Proceedings of the 44th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages - POPL 17*, 29, 2017.

[22] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016*, 2016.

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

[24] Xavier Leroy. The CompCert verified compiler, software and annotated proof, March 2019.

[25] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert memory model. In Appel [2], chapter 32.

[26] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR*, 41(1), 2008.

[27] William Mansky, Andrew W. Appel, and Aleksey Nogin. A verified messaging system. *Proc. ACM Program. Lang.*, 1(OOPSLA):87:1–87:28, October 2017.

[28] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1):271–307, May 2007.

[29] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP 2010: 24th European Conference on Object-Oriented Programming*, pages 478–503. Springer, 2010.

[30] Anton V. Podkopaev, Ori Lahav, and Viktor Vafeiadis. Promising compilation to ARMv8.3. *Proceedings of the Institute for System Programming of the RAS*, 29(5):149–164, 2017.

[31] Anton V. Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. *Proceedings of the ACM on Programming Languages 3, no. POPL*, 2019.

[32] Jaroslav Sevcik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. In *Journal of the ACM (JACM)*, page 60(3), 2013.

[33] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W Appel. Compositional CompCert. In *POPL*, volume 50, pages 275–287. ACM, 2015.

[34] James Gordon Stewart. *Verified Separate Compilation for C*. PhD thesis, Princeton University, Princeton, NJ, June 2015.