CompCertOC: Verified Compositional Compilation of Multi-Threaded Programs with Shared Stacks

ANONYMOUS AUTHOR(S)*

Sharing of stack data between threads is a common practice in multi-threaded programming (e.g., scoped threads in C++ and Rust). It is a long-standing open problem to support verified compilation of multi-threaded programs *compositionally* in presence of stack-data sharing. Although certain solutions exist on paper, none of them is completely formalized because of the difficulty in simultaneously enabling the sharing and forbidding the modification of stack memory in presence of arbitrary memory operations (e.g., pointer arithmetic).

We present a compiler verification framework that solves this open problem. To address the challenge of sharing stack data, we introduce *threaded Kripke memory relations* to support memory protection and stack-data sharing on a multi-stack memory model. We further introduce *threaded direct refinements* parameterized by the Kripke memory relations, which are capable of describing semantics preservation for compiling open modules in multi-threaded contexts. We show that threaded direct refinements are both *horizontally composable*—thereby enabling the compositional verification of open threads and heterogeneous modules—and *vertically composable*—thereby enabling composition of compiler correctness for multiple compiler passes.

We apply this framework to the *full* compilation chain of CompCert, resulting in CompCertOC, the first optimizing verified C compiler that supports compilation of multi-threaded modules with sharing of stack data and without any artificial restriction on compositionality. We demonstrate the capability of CompCertOC through compositional verification of multi-threaded heterogeneous programs (i.e., written in C and assembly) which mutually interact with each other by using standard POSIX thread primitives (e.g., dynamic thread creations and joining) while sharing stack data during their interaction.

1 Introduction

Software often consists of modules written in different languages (i.e., heterogeneous modules). To ensure the verified properties hold for their compiled binary code, it is essential to support *verified compositional compilation* (VCC), i.e., separate verification of the compilers for heterogeneous modules and composition of compiler correctness theorems. The past decade has witnessed great progress towards achieving this goal [3, 12, 22–24, 27]where the projects aiming at realistic optimizing compilers are mostly built upon CompCert, the state-of-the-art verified compiler [14].

It has been long believed that VCC can support concurrent programs. When a single concurrent thread (or process) is under focus, it behaves like an open module interacting with other threads through external function calls (context switches behave like a function call in this case). Therefore, compiler correctness for individual processes can be proved separately and composed into that for the complete program. Although the existing works have realized this vision to a certain extent [5, 8, 26], they are unable to support a common practice in multi-threaded programming, i.e., sharing of stack-allocated data among threads. [1]

A simple example for illustrating this problem is depicted in Fig. 1. It is representative of concurrent programs that delegate tasks to child threads by sharing stack data. In Fig. 1, the main thread creates a child thread for increasing its stack variable `i` by passing its address, which effectively results in sharing of `i` between threads. After the task is completed, the main thread prints the result. In essence, the two threads behave like two modules that, although sharing the same code, have different entry points (`main` and `thread`), and more importantly, may have interleaved executions because of concurrency. Context switching starting from one thread and ending with switching back behaves like an external call of that thread. Therefore, the compiler correctness theorems for the two threads should simply be those for modules focusing on different entry functions, which are then composed to form the correctness for compiling the whole program.

[1] At best, a solution exists on the paper [8] which is not formalized due to excessive complexity. See §7 for details.

```
1 void *thread(void *p) { (*(int *)p)++; }
2 int main() {
3         int i = 0; pthread_t tid;
4         pthread_create(&tid, NULL, thread, (void *)&i);
5         pthread_join(tid, NULL);
6         printf("%d\n", i);}
```

Fig. 1. A Simple Concurrent Program not Supported by VCC

Unfortunately, none of the existing approaches to VCC supports this seemly trivial example, because there are fundamental conflicts between supporting shared stack-data and the critical compositionality properties for compiler correctness, i.e., *horizontal and vertical compositionality*:

$$\textbf{Vertical Compositionality:} \quad L_1 \preccurlyeq L_2 \Rightarrow L_2 \preccurlyeq L_3 \Rightarrow L_1 \preccurlyeq L_3$$
$$\textbf{Horizontal Compositionality:} \quad L_1 \preccurlyeq L_1' \Rightarrow L_2 \preccurlyeq L_2' \Rightarrow L_1 \oplus L_2 \preccurlyeq L_1' \oplus L_2'$$

Here, a *refinement* relation $L_1 \preccurlyeq L_2$ encodes the compiler correctness between semantics of open modules before (L_1) and after (L_2) compilation. Vertical compositionality ensures that the refinements for two adjacent compiler passes can be combined into a single refinement; it is critical for supporting multi-pass compilers. Horizontal compositionality ensures that the parallel refinements for different open modules can be combined together; it is critical for compiling heterogeneous modules where L_1 and L_2 may denote the semantics of open modules written in different languages ($L_1 \oplus L_2$ denotes the semantic composition of L_1 and L_2) with complicated interactions (e.g., mutual calls). Both compositionalities require appropriate description of the *rely conditions* the open modules depend upon and the *guarantee conditions* they ensure. A great amount of work has been devoted to ensure both compositionalities with or without concurrency [3, 7, 12, 18, 19, 22, 23, 27].

Continue with the example in Fig. 1. The child thread should be able to modify the publicly accessible stack variable `i` of its parent; on the other hand, it should not modify any private stack variable of its parent (e.g., `tid`). The requirement for sharing public stack data while protecting private stack data imposes additional rely-guarantee conditions, further complicating the problem of composing compiler correctness. Because it is unclear how to solve this conflict, existing work on VCC for concurrency (among them the most well-known are Thread-safe CompCertX [5] and CASCompCert[8]) simply forbid stack memory of one thread (`i` for the main thread in Fig. 1) to be leaked to and modified by another thread. This solution deviates from the common assumption that stack memory should be treated no differently from other memory such as heap or global memory (e.g., this is the assumption of POSIX thread APIs), and cannot support common programming idiom like in Fig. 1 or more complex stack sharing mechanisms such as *scoped threads* in Rust [16].

In this paper, we propose a completely formalized solution to the problem of sharing stack-data in VCC for multi-threaded programs. Our contributions are summarized as follows:

- We introduce *Threaded Kripke Memory Relations* (TKMR), a formalization of rely-guarantee conditions for multi-threaded programs that simultaneously supports sharing of public stack data and protection of private ones. On top of that, we introduce *Threaded Direct Refinements*, a notion of compiler correctness that supports shared stacks, open threads written in heterogeneous modules, and both horizontal and vertical compositionality, which exceeds the capability of all the existing formulations of compiler correctness for concurrency.
- We verify that the full compilation chain of CompCert satisfies threaded direct refinement. By vertical compositionality of threaded direct refinements, we get CompCertOC, the first verified optimizing compiler that supports multi-threaded open programs with stack sharing. The formal development is based on the two existing extensions—Nominal CompCert [25]

```c
/* client.c */
#define N 5
typedef struct {
  int *input, *result, size; } Arg;
void* server(void *a);

int main() {
  pthread_t a;
  int input[N]={1,2,3,4,5}, result[N];
  int mask = 0;
  Arg arg = {input,result,N};

  pthread_create(&a,0,server,&arg);
  for (int i = 0;i < N;i++)
    { mask += input[i]; yield(); }
  pthread_join(a, NULL);

  for (int i = 0;i < N;i++) {
    result[i] = result[i] & mask;
    printf("%d; ", result[i]);  }
}
```

(a) Client running on the main thread

```c
/* server.c */
void encrypt (int i, int *r);
void* server(void *a) {
  int *i = ((Arg *)a)->input;
  int *r = ((Arg *)a)->result;
  int size = ((Arg *)a)->size;

  for (int j = 0;j < size;j++) {
    encrypt(input[j], result+j);
    yield();  }
  return NULL;
}
/* encrypt.s */
key: .long 42
encrypt:
Pallocframe 16 8 0 // allocate frame
Pmov key RAX
Pxor RAX RDI   // result = i XOR key
Pmov RDI (RSI) // stores to (RSI)
Pfreeframe 16 8 0 // free frame
Pret
```

(b) Server running on the child thread

Fig. 2. An Example of Multi-Threaded Program with Stack Sharing

for supporting thread-local stacks and CompCertO [12] for supporting direct refinements. It is fully formalized in Coq for CompCert v.3.13 (See the supplementary materials).

- To demonstrate the effectiveness of the above framework, we apply it to verify non-trivial multi-threaded programs that combine heterogeneity (modules written by in different languages such as C and assembly) and stack sharing (sharing stack data between parent and child threads using POSIX thread APIs). For this, we develop semantics for describing programs using POSIX thread APIs and the techniques for composing compiler correctness to derive semantic refinements between complete multi-threaded programs.

In the rest of the paper, we first introduce our key ideas for enabling stack sharing in VCC in §2. We then introduce background for the subsequent technical developments in §3. We elaborate on our ideas and contributions in §4, §5 and §6. We discuss the related work and conclude in §7.

2 Key Ideas

To illustrate the key ideas of our work, we introduce a running example of multi-threaded program composed of heterogeneous open modules in Fig. 2. It consists of a client written in C (Fig. 2a) which runs on the main thread and a server written in C and assembly (Fig. 2b) which runs on the child thread. This example consists of both sharing and protection of stack data between open modules and threads. Note that we use POSIX primitives `pthread_create` and `pthread_join` to implement thread creation and synchronization. We explicitly use `yield` function to encode context switching. Note that this is not a fundamental limitation as it can be proved at the target level that the preemptive concurrent semantics behave the same as non-preemptive ones (e.g., see CASCompCert [8]). Since this proof is independent of compiler correctness, we assume that we are only working with non-preemptive semantics with explicit yields in the remaining discussion.

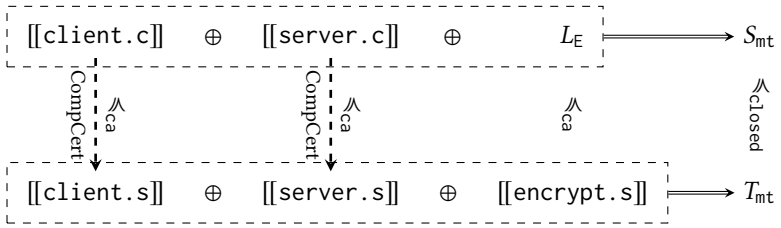
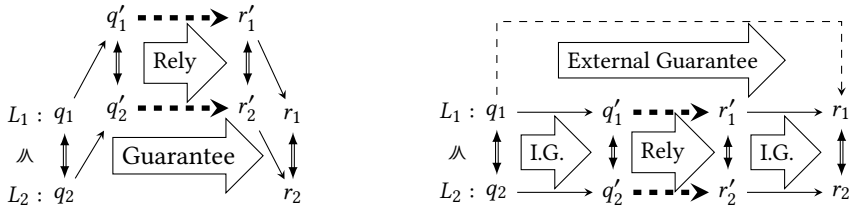


Fig. 3. Verifying the Running Example



(a) Rely-Guarantee Simulation (b) Threaded Rely-Guarantee Simulation

Fig. 4. Open Simulations for Sequential and Multi-Threaded Open Modules

This example encrypts `N` input integers into the results, where the encryption happens both at the server side (XORing inputs with a fixed encryption key 42) and at the client side (encoding the results of the server with a mask calculated from the inputs). The `main` function uses variables `input` and `result` as its buffers. The addresses and sizes of these buffers are passed to `server` through `pthread_create`. Concurrent to the execution of `server`, the client calculates a mask from the input values which is used to encode the results after the `pthread_join`. The function `encrypt` is written in assembly. It receives input value i and the pointer to result r in `RDI` and `RSI` respectively, does the encryption using XOR and stores the result. The function `server` simply calls the encryption function written in assembly for each input value. We assume the context switches happen as the end of each iteration as denoted by `yield()` in the figure.

With the running example, our goal is to verify the refinement of multi-threaded behavior of the composed program using *threaded direct refinements* as shown in Fig. 3. The threaded direct refinement $\preccurlyeq_{\mathsf{ca}}: C \Leftrightarrow \mathcal{A}$ is derived from CompCert's compilation chain where C and $\mathcal{A}$ are the languages interfaces for C and assembly, respectively. It uses a multi-threaded memory model to describe the rely-guarantee conditions for both sharing and protection of memory regions. We define the C-level specification L_E for `encrypt.s` and prove $L_E \preccurlyeq_{\mathsf{ca}}$ [[`encrypt.s`]] manually. The source and target open semantics can be composed into closed multi-threaded semantics $S_{\mathtt{mt}}$ and $T_{\mathtt{mt}}$. The multi-threaded semantics starts with `main()` on main thread. When it calls `pthread_create`, the open semantics of `server` is used to create a fresh program state on the child thread. Finally, we compose the open refinements into a closed refinement for multi-threaded semantics.

Obviously, the most important point for this verification is the definition of the compiler correctness $\preccurlyeq_{\mathsf{ca}}$ for open modules, i.e. the refinement between open semantics to enable both module linking and thread linking at the same time. Before discussing how to define it in §2.1, we first briefly review the correctness for *sequential* open modules. The semantics of open module is implemented as open *labeled transition systems*. The refinement between them is defined as forward simulations using rely-guarantee conditions which have been widely studied [12, 22, 23, 27].

Fig. 4a depicts a simulation between two open semantics L_1 and L_2. The source (target) semantics L_1 (L_2) is invoked using a query (i.e. function call) q_1 (q_2) and may issue external call q_1' (q_2') after

internal execution. When the external function returns with a reply r_1' (r_2'), L_1 (L_2) continues and finally (it may call several external calls) returns a reply r_1 (r_2) to the initial query. Such simulation needs an invariant between the source and target program states (denoted by the vertical double arrows in Fig. 4a) to stay valid through the whole execution. For the external calls, we need to assume some *rely-conditions* for unknown external executions such that the invariant between r_1' and r_2' still holds. On the other hand, we need to prove symmetry *guarantee-conditions* from q_1 (q_2) to r_1 (r_2), so that L_1 and L_2 satisfy the rely conditions from their callers.

The critical part of these rely-guarantee conditions is how to describe the possible evolutions of memory states. We use Kripke Memory Relations, or KMRs to define such relations. The KMR consists of a world type W, each world $w = (j, m_1, m_2) \in W$ is a pair of source and target memory states related by a partial injection function. Therefore, the rely-guarantee conditions on memories can be defined using the *accessibility* relation between worlds (written as $w \rightsquigarrow w'$). Such conditions often assume that some memory regions are unchanged during the external calls. For example, the rely condition for function `encrypt` in [[`server.c`]] requires that the local index variable `j` is unchanged in its reply because `j` could be optimized into a callee-saved register in [[`server.s`]]. The simulation breaks if `j` is changed by external call.

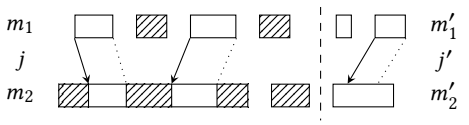


Fig. 5. Kripke Worlds Related by `injp`

These memory regions which should be protected from the environment are usually called *private* memory. One of the most important task for proving refinement between open semantics is to distinguish between protected *private* memory and shared *public* memory. A trivial but undesirable solution is to forbid stack-allocated data and make all stack memory private.

A Kripke Memory Relation for memory protection `injp` is introduced in CompCertO [12]. `injp` uses the injection j to define the *private* and *public* memory of a world (j, m_1, m_2) as depicted in Fig. 5. The source and target memories m_1 and m_2 are related by *memory injection* where the related regions (white areas) are *public*. The *unmapped* regions in m_1 (shaded areas in m_1) and *out-of-reach* regions in m_2 (shaded areas in m_2) are defined as *private* memory. The updated memories m_1' and m_2' may have newly allocated regions related by j'. The accessibility $(j, m_1, m_2) \rightsquigarrow (j', m_1', m_2')$ ensures that the private regions in m_1 and m_2 are unchanged in m_1' and m_2', i.e., they cannot be modified by the callee. The Threaded Kripke Memory Relation we discuss later is based on `injp`.

2.1 Threaded Kripke Memory Relation with Sharing and Protection of Stacks

In this work, we focus on verified compilation of open multi-threaded programs. However, we do not *explicitly* define context switches in the open semantics. In other words, the rely condition of unknown environment should uniformly describe the possible behaviors for both external calls and context switches. It is easy to realize that the refinement depicted in Fig. 4a is not strong enough to support context switches. Because the guarantee condition is provided only for a complete function execution. While the external behavior of context switch could come from any possible execution.

To satisfy the guarantee conditions in both situations, we define *threaded rely-guarantee simulation* as depicted in Fig. 4b. We add *internal guarantee* condition (denoted as I.G.) to capture the guarantees of internal executions. The *rely* and *external guarantee* conditions remain the same.

These conditions are defined using Threaded Kripke Memory Relations (TKMR) which is an extension of KMR and has two main differences. Firstly, it explicitly encodes different memory regions (i.e., global memory and stack memory). We use Nominal Memory Model [25] to define a multi-stack memory model. The memory space is divided into global memory and individual stack memory for each thread. This is distinct from the original CompCert memory model where all memory blocks reside in a single space. Each stack block is named using the thread id of its owner.

The memory state also includes a current thread id. Thus it is easy to distinguish between thread local blocks and external blocks. In the following discussion, we use w with possible superscripts or subscripts to denote Kripke worlds in TKMR which take the form (j, m, tm), meaning that the source memory m is related to the target memory tm by j under the same current thread id.

Secondly, TKMR defines two different accessibility relations, internal accessibility (denoted as $\rightsquigarrow_i$) and external accessibility (denoted as $\rightsquigarrow_e$). While the $\rightsquigarrow$ for protection in KMR only protects all the private regions from external calls, the different accessibilities here are defined as:

(1) $w \rightsquigarrow_i w'$ holds if the *private* memory of the *other* threads in w is unchanged in w'.
(2) $w \rightsquigarrow_e w'$ holds if the *private* memory of the *current* thread in w is unchanged in w'.

Using TKMR, the rely-guarantee conditions on memory evolution are presented in Fig. 6 which is an extension of Fig. 4b. w_q represents the world from initial queries (i.e. q_1 and q_1' in Fig. 4b). The semantics call external queries q_1 and q_2 and the *rely* conditions of them are represented by $\rightsquigarrow_e$ in red. $\rightsquigarrow_i$ is used for *internal guarantee* which defines legal memory evolution of any internal execution starts from either initial call (w_q) or function reply (w_{r_2}) to either function call (w_{q_1}) or the final return (w_r). Note that we require $\rightsquigarrow_i$ being transitive for composing the internal guarantee conditions for different modules as we discuss later. The *external guarantee* $w_q \rightsquigarrow_e w_r$ for the complete evolution from function call to its return is also defined using $\rightsquigarrow_e$.

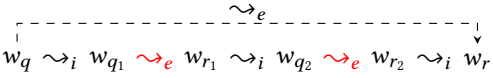


Fig. 6. World Transitions in Threaded Simulation

As discussed above, the rely conditions should work for both external calls and context switches. We first illustrate how internal accessibility is used to satisfy the rely condition for context switch in Fig. 7. The context switch between different threads is denoted by arrows between worlds for TKMR, denoted by w_i^X where X is the thread id. For example, w_0^A and w_0^B contain the same memory states except for that the current thread is changed from A to B. The initial source and target memory states are stored in w_0^A. Between the point A switching to another thread and the point the execution switches back to A, the *rely* condition requires *local* (i.e. allocated by A) private memory unchanged. This condition is satisfied as follows. Assume thread B executes w_0^B to w_1^B, the internal guarantee condition of the running module provides $w_0^B \rightsquigarrow_i w_1^B$ which says all *external* (i.e. not allocated by B) private memory is unchanged. Obviously, the private memory from A is protected. B then switches to another thread C. Similarly, the internal guarantee of C provides $w_1^C \rightsquigarrow_i w_2^C$. The private memory from A is also protected. Therefore, when C switches back to A, the rely condition $w_0^A \rightsquigarrow_e w_2^A$ is satisfied by the internal guarantee conditions of both B and C.

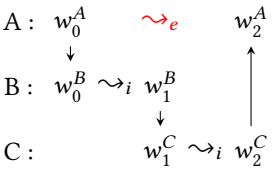


Fig. 7. Rely Satisfied by I.G.

The rely condition for sequential external calls is trivially satisfied by the external guarantee provided by the refinement of the callee module *runs on A*, regardless whether the callee function switches to other threads or not. Consider the simulation for the callee modules as depicted in Fig. 4b, the rely condition ensures that *all* private memory on A is protected from the environment. The callee only needs to ensure that its internal executions does not change the private memory *in initial memories*. This is exactly the sequential guarantee condition which is proved in Fig. 4a.

2.2 Verification of the Example using Threaded Kripke Memory Relation

We further demonstrate how to verify our running example using TKMR. We start from the `yield()` at line 15 of `client.c`, the snapshot of the *source* stack frames for both threads is depicted in Fig. 9. The shadowed regions are private regions from each thread which are optimized into registers in

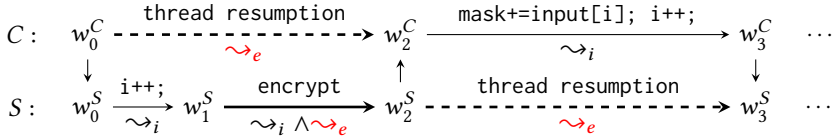


Fig. 8. TKMR Accessibility Relations for the Running Example

compiled *target* assembly program. The remaining part in $b_{\texttt{main}}$ is shared public region. The server can access them through the pointers. We omit the block for `encrypt` because it does not contain any stack data. The most important part of the verification is how to protect the private regions in stack frames while allowing the sharing of public regions.

By our assumption, context switch only occurs at `yield()` in Fig. 2. A complete loop of context switches between the two threads is depicted in Fig. 8. The execution follows the solid arrows, while vertical ones stand for context switch and horizontal ones stand for internal execution. In order to verify the running example, we need to show that the rely condition of each module can be satisfied by the guarantee conditions provided by other modules. We discuss how to prove the three rely conditions $\rightsquigarrow_e$ in the figure one by one.

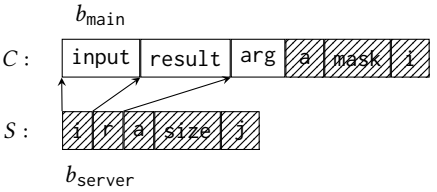


Fig. 9. Snapshot of Source Stack Frames

The $w_0^C \rightsquigarrow_e w_2^C$ for context switch is satisfied by internal guarantees. w_0^C contains the memory states at the point that the main thread switches out and w_2^C stands for the memory states when the child thread switches back. This external accessibility requires the local memory on the main thread, i.e., the variables `a`, `mask` and `i` in $b_{\texttt{main}}$ as depicted in Fig. 9 are unchanged. It is satisfied by the internal guarantee of `server` and `encrypt` which can only change the private regions in the local block $b_{\texttt{server}}$ but not in $b_{\texttt{main}}$. Note that the internal guarantee condition holds for the whole execution of `encrypt` because it does not switch to other threads. Then the $\rightsquigarrow_i$ for these two functions are transitively composed. Even if `encrypt` calls another module sequentially, we can still compose the internal guarantee condition of the callee to get the internal accessibility on thread S. Symmetrically $w_2^S \rightsquigarrow_e w_3^S$ is satisfied by the internal guarantee on main thread C.

The $w_1^S \rightsquigarrow_e w_2^S$ for external call `encrypt` is satisfied by its external guarantee. The server requires that its private variables in $b_{\texttt{server}}$ are unchanged during the external call `encrypt`. It is trivially satisfied by the external guarantee of `encrypt`. Note that the combination of internal and external guarantee of `encrypt` actually tells us that *all* private regions in w_1^S are protected in w_2^S. This is exactly the accessibility of sequential KMR `injp`.

2.3 Threaded Direct Refinements and Their Compositionality

Threaded rely-guarantee simulations relate the source and target program states directly through TKMR. We shall see that they are both horizontally and vertically composable, and the results again directly relate source and target semantics through a *single* TKMR. We will call such simulations *threaded direct refinements* interchangeably in the remaining discussion.

We demonstrate the horizontal and vertical compositionality of threaded direct refinements for achieving the verification as depicted in Fig. 3. Horizontally, we need to compose not only open modules on the same thread (i.e. `server` and `encrypt`), but also the open threads (i.e. client and server). We have discussed how the rely-guarantee conditions defined by TKMR can support inter-module and inter-thread interaction in previous subsections. Technically, the verification of our running example is divided into two steps, module linking and thread linking. We first

compose the source and target semantics into *open semantics* which do not contain any external functions. We then turn it into a *closed multi-threaded semantics* by implementing program loading and context switches. The detailed discussion can be found in §4. Note that TKMR can also be used for different composition methods. For example, it could be useful for reasoning about *open multi-threaded semantics* which runs on a certain set of threads A and assumes an unknown set of threads B as its environment. We can easily prove that the internal guarantee conditions of threads in B can satisfy the rely condition that all private memory from threads in A is unchanged.

The horizontal composition discussed above assumes that the composed simulations have symmetry rely-guarantee conditions. In order to compose heterogeneous modules in the example, we use threaded direct refinement as the compiler correctness of CompCertOC which shares the same *single* TKMR with the hand-written proof for `encrypt`. Following the work of Zhang et al. [27], it means we need to vertically compose the correctness of individual compiler passes into a single refinement $\preccurlyeq_{\mathsf{ca}}$. Although the KMR `injp` for sequential programs has been proved to be transitively composable, the same property is more complicated to prove for TKMR because we need to compose not only the external accessibility $\rightsquigarrow_e$ but also $\rightsquigarrow_i$. The key observation is that TKMR is transitively composable despite the introduction of internal guarantees. To prove this transitivity, we introduce an additional invariant stating that intermediate semantics is *hidden* from other threads. The proof will be discussed in §4.

3 Background

3.1 Block-based Memory Model and Nominal Memory Model

The block-based memory model of CompCert [15] defines a memory state m (of type `mem`) as a disjoint set of *memory blocks*. A pointer (b, o) points to the o-th byte from block b where b has type `block` and $o \in Z$ is an integer. Values of type `val` are either 32- or 64-bit integers or floats, pointers ($\mathtt{Vptr}(b, o)$) or undefined values (`Vundef`).

During the compilation, source and target memories are related via partial *injection functions* $j : \mathtt{block} \to \lfloor \mathtt{block} \times \mathtt{Z} \rfloor$ where $j(b) = \emptyset$ if b is removed and $j(b) = \lfloor (b', o) \rfloor$ if b is mapped to (b', o) in the target memory. The type of j is called `meminj`. The values v_1 and v_2 are related by j (denoted as $v_1 \hookrightarrow_v^j v_2$) if either $v_1 = \mathtt{Vundef}$, $v_1 = v_2$ are equal scalar values or pointers point to related slots, i.e. $v_1 = \mathtt{Vptr}(b_1, o_1)$, $v_2 = \mathtt{Vptr}(b_2, o_2)$ and $j(b_1) = \lfloor (b_2, o_2 - o_1) \rfloor$. We say m_1 and m_2 are related via j (denoted by $m_1 \hookrightarrow_m^j m_2$) if the values from related slots are related by j.

Nominal Memory Model [25] is an extension of the block-based memory model using *nominal techniques* [20]. It introduces a nominal interface for the memory model such that one can customize the definition of the type of memory blocks `block` for separating different memory regions. If the operations on `block` (such as generation of new block id) satisfy certain well-formedness requirements, the proof of whole compilation can still work. We use nominal memory model to define a multi-stack memory model where `mem` and `block` both contain thread id. Each thread can allocate blocks without affecting each other and recognize its local block by thread id.

3.2 A Framework for Open Simulation of Sequential Semantics

A *language interface* $A = \langle A^q, A^r \rangle$ is a pair of sets where A^q (A^r) denotes the type of queries (replies) for an open module. The language interface for C semantics is defined as $C = \langle \mathtt{val} \times \mathtt{sig} \times \mathtt{val}^* \times \mathtt{mem}, \mathtt{val} \times \mathtt{mem} \rangle$. A query $v_f[sg](\vec{v})@m$ consists of function pointer, signature, arguments and memory and a reply $v'@m'$ is return value together with updated memory. The interface for assembly is $\mathcal{A} = \langle \mathtt{regset} \times \mathtt{mem}, \mathtt{regset} \times \mathtt{mem} \rangle$ where its queries and replies have the form $rs@m$.

Open labeled transition systems (LTS) represent semantics of modules that may accept queries and provide replies at the *incoming side* and provide queries and accept replies at the *outgoing side*

(i.e., calling external functions). An open LTS $L : A \twoheadrightarrow B$ is a tuple $\langle D, S, I, \rightarrow, F, X, Y\rangle$ where A (B) is the language interface for outgoing (incoming) queries and replies, $D \subseteq B^q$ a set of initial queries, S a set of internal states, $I \subseteq D \times S$ ($F \subseteq S \times B^r$) transition relations for incoming queries (replies), $X \subseteq S \times A^q$ ($Y \subseteq S \times A^r \times S$) transitions for outgoing queries (replies), and $\rightarrow \subseteq S \times \mathcal{E}^* \times S$ internal transitions emitting events of type $\mathcal{E}$. Note that $(s, q^O) \in X$ iff an outgoing query q^O happens at s; $(s, r^O, s') \in Y$ iff after q^O returns with r^O the execution continues with an updated state s'.

Kripke relations are used to describe evolution of program states in open simulations. A Kripke relation $R : W \rightarrow \{S \mid S \subseteq A \times B\}$ is a family of relations indexed by a *Kripke world* W; for simplicity, we define $\mathcal{K}_W(A, B) = W \rightarrow \{S \mid S \subseteq A \times B\}$. The *simulation convention* relating two language interfaces A_1 and A_2 is defined as $: \mathbb{R} : A_1 \Leftrightarrow A_2 = \langle W, \mathbb{R}^q : \mathcal{K}_W(A_1^q, A_2^q), \mathbb{R}^r : \mathcal{K}_W(A_1^r, A_2^r)\rangle$

The forward open simulation [12] is denoted by $L_1 \leqslant_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2$ and formally defined as follows:

Definition 3.1. Given $L_1 : A_1 \twoheadrightarrow B_1$, $L_2 : A_2 \twoheadrightarrow B_2$, $\mathbb{R}_A : A_1 \Leftrightarrow A_2$ and $\mathbb{R}_B : B_1 \Leftrightarrow B_2$, $L_1 \leqslant_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2$ holds if there is some Kripke relation $R \in \mathcal{K}_{W_B}(S_1, S_2)$ that satisfies:

(1) $\forall\ w_B\ q_1\ q_2,\ (q_1, q_2) \in \mathbb{R}_B^q(w_B) \Rightarrow (q_1 \in D_1 \Leftrightarrow q_2 \in D_2)$

(2) $\forall\ w_B\ q_1\ q_2\ s_1,\ (q_1, q_2) \in \mathbb{R}_B^q(w_B) \Rightarrow (q_1, s_1) \in I_1 \Rightarrow \exists\ s_2, (s_1, s_2) \in R(w_B) \wedge (q_2, s_2) \in I_2.$

(3) $\forall\ w_B\ s_1\ s_2\ t\ s_1',\ (s_1, s_2) \in R(w_B) \Rightarrow s_1 \xrightarrow{t} s_1' \Rightarrow \exists\ s_2', (s_1', s_2') \in R(w_B) \wedge s_2 \xrightarrow{t}{}^{*}\ s_2'.$

(4) $\forall\ w_B\ s_1\ s_2\ q_1,\ (s_1, s_2) \in R(w_B) \Rightarrow (s_1, q_1) \in X_1 \Rightarrow$
$\exists w_A\ q_2,\ (q_1, q_2) \in \mathbb{R}_A^q(w_A) \wedge (s_2, q_2) \in X_2 \wedge$
$\forall\ r_1\ r_2\ s_1',(r_1, r_2) \in \mathbb{R}_A^r(w_A) \Rightarrow (s_1, r_1, s_1') \in Y_1 \Rightarrow \exists\ s_2', (s_1', s_2') \in R(w_B) \wedge (s_2, r_2, s_2') \in Y_2.$

(5) $\forall\ w_B\ s_1\ s_2\ r_1,\ (s_1, s_2) \in R(w_B) \Rightarrow (s_1, r_1) \in F_1 \Rightarrow \exists\ r_2, (r_1, r_2) \in \mathbb{R}_B^r(w_B) \wedge (s_2, r_2) \in F_2.$

Property (1) requires that L_1 and L_2 accept related queries; (2) establish the internal invariant $R(w_B)$ from the related incoming queries; (3) requires internal execution to preserve R; (4) requires if L_1 issues external call q_1, L_2 should issue q_2 which is related to q_1 by some world w_A derived from current program states, and when the external calls return, $R(w_B)$ is reestablished; (5) requires that the replies to initial queries are related by w_B. Note that the *rely-guarantee* conditions as depicted in Fig. 4a is implicitly defined in the $\mathbb{R}^r(w)$ relation of replies while w remembers the memory states in initial queries. We will present a concrete example in next subsection.

3.3 The Kripke Memory Relation `injp` and Direct Refinement

We present the definition of Kripke Memory Relation and a specific `injp` to define the simulation convention of the direct refinement [27]. It is derived from CompCert and used to verify the composition of sequential heterogeneous modules. The verification of multi-threaded programs is based on an extended version of this direct refinement.

Definition 3.2 (Kripke Memory Relation). A Kripke Memory Relation is a tuple $\langle W, f, \rightsquigarrow, R\rangle$ where W is a set of worlds, $f : W \rightarrow \mathtt{meminj}$ a function for extracting injections from worlds, $\rightsquigarrow \subseteq W \times W$ an accessibility relation between worlds and $R : \mathcal{K}_W(\mathtt{mem}, \mathtt{mem})$ a Kripke relation over memory states that is compatible with the memory operations. We write $w \rightsquigarrow w'$ for $(w, w') \in \rightsquigarrow$.

Definition 3.3 (Kripke Relation with Memory Protection). $\mathtt{injp} = \langle W_{\mathtt{injp}}, f_{\mathtt{injp}}, \rightsquigarrow_{\mathtt{injp}}, R_{\mathtt{injp}}\rangle$ where $W_{\mathtt{injp}} = (\mathtt{meminj} \times \mathtt{mem} \times \mathtt{mem})$, $f_{\mathtt{injp}}(j, \_, \_) = j$, $(m_1, m_2) \in R_{\mathtt{injp}}(j, m_1, m_2) \Leftrightarrow m_1 \hookrightarrow_m^j m_2$ and

$$(j, m_1, m_2) \rightsquigarrow_{\mathtt{injp}} (j', m_1', m_2') \Leftrightarrow j \subseteq j' \wedge \mathtt{unmapped}(j) \subseteq \mathtt{unchanged\text{-}on}(m_1, m_1') \wedge \mathtt{out\text{-}of\text{-}reach}(j, m_1) \subseteq \mathtt{unchanged\text{-}on}(m_2, m_2').$$

$\mathtt{unchanged\text{-}on}(m, m')$ denotes memory cells whose permissions and values are not changed from m to m'. $(b_1, o_1) \in \mathtt{unmapped}(j)$ iff $j(b_1) = \emptyset$ and $(b_2, o_2) \in \mathtt{out\text{-}of\text{-}reach}(j, m_1)$ means if $j(b_1) = \lfloor (b_2, o_2 - o_1)\rfloor$ then (b_1, o_1) is invalid (i.e. either uninitialized or already freed) in m_1. The unmapped

and out-of-reach are exactly the *private* regions in source and target memories as depicted in Fig. 5. $\rightsquigarrow_{\mathtt{injp}}$ ensures that private regions are unchanged from function calls to returns.

Definition 3.4 (Direct Refinement). The *direct refinement* is defined as a single simulation $L_{\mathtt{c}} \leqslant_{\mathtt{CA_{injp}} \twoheadrightarrow \mathtt{CA_{injp}}} L_{\mathtt{asm}}$ which directly relates C and assembly semantics. The simulation convention $\mathtt{CA_{injp}}$ is implicitly defined using injp as: $\mathtt{CA_{injp}} = \langle W_{\mathtt{CA}}, \mathbb{R}^q_{\mathtt{CA}}, \mathbb{R}^r_{\mathtt{CA}} \rangle$ where $W_{\mathtt{CA}} = (W_{\mathtt{injp}}, \mathtt{sig}, \mathtt{regset})$. A world $w = ((j, m_1, m_2), sg, rs)$ consists of related memories, the signature of incoming function and the initial register values.

$(v_f[sg](\vec{v})@m_1, rs@m_2) \in \mathbb{R}^q_{\mathtt{CA}}(w)$ holds if

(1) w corresponds to the initial queries, i.e. $w = ((j, m_1, m_2), sg, rs)$
(2) The memories are related by injp, i.e. $(m_1, m_2) \in R_{\mathtt{injp}}(j, m_1, m_2)$;
(3) The function pointer is related to the program counter in assembly, i.e. $v_f \hookrightarrow^j_v rs(\mathsf{PC})$;
(4) The arguments $\vec{v}$ are projected to registers in rs or stack slots pointed by the stack pointer $rs(\mathsf{RSP})$ in m_2 according to the C calling convention and the signature sg;
(5) The outgoing arguments in m_2 are stored in private region, i.e. not in the image of j.

Using the same world w, $(res@m_1', rs'@m_2') \in \mathbb{R}^r_{\mathtt{CA}}(w)$ holds if

(1) The source return value res is related to the register in rs' according to sg;
(2) For any callee-saved register r, $rs'(r) = rs(r)$;
(3) Stack pointer and program counter are restored, i.e. $rs'(\mathsf{RSP}) = rs(\mathsf{RSP}) \wedge rs'(\mathsf{PC}) = rs(\mathsf{RA})$;
(4) Private memory is protected, i.e. $(j, m_1, m_2) \rightsquigarrow_{\mathtt{injp}} (j', m_1', m_2')$ and $(m_1', m_2') \in R_{\mathtt{injp}}(j', m_1', m_2')$.

4 Threaded Kripke Memory Relation with Shared Stack

In order to verify the compilation of open multi-threaded programs, we first define a multi-threaded memory model. Then, we extend injp into Threaded Kripke Memory Relation tinjp with internal and external accessibilities. We further define the threaded simulation convention and open simulation. Finally, we discuss the compositionality of tinjp for threaded direct refinements.

4.1 Multi-Threaded Memory Model

We implement multi-threaded memory model as an instantiation of Nominal Memory model. The block type block and *support* type sup (denoting the shape of global memory space) is defined as: $\mathtt{block} = \mathtt{nat} \times \mathtt{positive}$, $\mathtt{sup} = \mathtt{nat} \times \mathtt{list}\ (\mathtt{list}\ \mathtt{positive})$. A block name $b = (t, p)$ consists of its thread it t and a block number p. A support set of the memory state $(t, stacks)$ also consists of the current running thread t and lists of block numbers of all existing threads. We use $\mathtt{tid}(m)$ and $\mathtt{tid}(b)$ to denote the thread id of memory and block. $stacks[t]$ consists of all block numbers for thread t, $(t, p) \in (t', stacks)$ is a valid block if $p \in stacks[t]$. When the program allocates a new block from the support $(t, stacks)$, it is named as $\mathtt{fresh\_block}((t, stacks)) = (t, max(stacks(t))+1)$. Note that the thread id start from 1 and $\mathtt{tid(b)} = 0$ means that b is a *global block*.

From the perspective of an open program, its thread id comes from the initial memory state $\mathtt{tid}(m)$. It allocates stack frames using this thread id. It is possible to decide whether b is thread local or external block by comparing $\mathtt{tid}(b)$ with $\mathtt{tid}(m)$. For multi-threaded semantics, we define additional memory operations thread_create and yield which only affects the support $(t, stacks)$. yield simply changes the thread id t to t' while $1 \leq t' \leq \mathtt{length}(stacks)$. thread_create appends a empty list after $stacks$ as the naming space for the new thread.

4.2 Refinements with Threaded Kripke Memory Relation

We define the threaded refinement using multi-stack memory model in this section. Recall that the rely-guarantee condition on memory is *implicitly* defined using KMR in simulation conventions for

sequential programs (Definition 3.4). Here we define and *explicitly* use TKMR in threaded simulation convention and forward simulation for the refinement between multi-threaded programs.

Definition 4.1 (Threaded Kripke Memory Relation). The threaded version of $\mathtt{injp}$ is defined as $\mathtt{tinjp} = \langle W_{\mathtt{tinjp}}, f_{\mathtt{tinjp}}, \leadsto_i^{\mathtt{tinjp}}, \leadsto_e^{\mathtt{tinjp}}, R_{\mathtt{tinjp}}\rangle$ where the related memories have the same thread id, i.e. $R_{\mathtt{tinjp}}(j, m_1, m_2) \Leftrightarrow m_1 \hookrightarrow_m^j m_2$ and $\mathtt{tid}(m_1) = \mathtt{tid}(m_2)$. $W_{\mathtt{tinjp}}$ and $f_{\mathtt{tinjp}}$ is defined the same as in $\mathtt{injp}$. Since a world (j, m_1, m_2) often contains related memories, we assume m_1 and m_2 have the same thread id in the following definition of internal and external accessibilities:

$$\begin{aligned}(j, m_1, m_2) \leadsto_i^{\mathtt{tinjp}} (j', m_1', m_2') \;\Leftrightarrow\; & j \subseteq j' \wedge \mathtt{tid}(m_1) = \mathtt{tid}(m_1') \\ & \wedge \mathtt{unmapped}(j) \cap \mathtt{ext}(m_1) \subseteq \mathtt{unchanged\text{-}on}(m_1, m_1') \\ & \wedge \mathtt{out\text{-}of\text{-}reach}(j, m_1) \cap \mathtt{ext}(m_2) \subseteq \mathtt{unchanged\text{-}on}(m_2, m_2').\\ (j, m_1, m_2) \leadsto_e^{\mathtt{tinjp}} (j', m_1', m_2') \;\Leftrightarrow\; & j \subseteq j' \wedge \mathtt{tid}(m_1) = \mathtt{tid}(m_1') \\ & \wedge \mathtt{unmapped}(j) \cap \mathtt{int}(m_1) \subseteq \mathtt{unchanged\text{-}on}(m_1, m_1') \\ & \wedge \mathtt{out\text{-}of\text{-}reach}(j, m_1) \cap \mathtt{int}(m_2) \subseteq \mathtt{unchanged\text{-}on}(m_2, m_2').\end{aligned}$$

This is the formal definition of $\leadsto_i$ and $\leadsto_e$ we presented in §2.1. The *private* regions in (j, m_1, m_2) are already defined by $\mathtt{unmapped}(j) \subseteq m_1$ and $\mathtt{out\text{-}of\text{-}reach}(j, m_1) \subseteq m_2$ via the injection function j in Definition 3.3. $\mathtt{ext}(m) \subseteq m$ stands for the *thread-external* blocks in m which have the different thread id with m, i.e. $b \in \mathtt{ext}(m) \Leftrightarrow \mathtt{tid}(b) \neq \mathtt{tid}(m)$. Similarly, the set of *thread-internal* blocks is defined as $\mathtt{int}(m) \subseteq m$ where $b \in \mathtt{int}(m) \Leftrightarrow \mathtt{tid}(b) = \mathtt{tid}(m)$. Note that both $\leadsto_i$ and $\leadsto_e$ require that the thread id is unchanged.

Definition 4.2 (Threaded Simulation Conventions). The *threaded simulation convention* $\mathbb{R}_{\mathsf{P}} : A_1 \Leftrightarrow A_2 = \langle W, \mathbb{R}^q : \mathcal{K}_W(A_1^q, A_2^q), \mathbb{R}^r : \mathcal{K}_W(A_1^r, A_2^r), \mathsf{P}\rangle$ is defined using a TKMR $\mathsf{P} = \langle W_{\mathsf{P}}, f_{\mathsf{P}}, \leadsto_i^{\mathsf{P}}, \leadsto_e^{\mathsf{P}}, R_{\mathsf{P}}\rangle$. Here $w_{\mathsf{P}} \in W_{\mathsf{P}}$ represents the related memories which are part of queries and replies related by $w \in W$. It is used in the internal invariant of threaded simulation to remember the memory world when the current module is recently invoked. During the simulation we need to *get* the memory world from queries or replies to define TKMR accessibilities. When queries related by $w \in W$ return, the replies need to be related by *setting* the memory world using updated memories while keeping the other information unchanged. Therefore, we require that W_{P} is a *sub-world* of W equipped with operations $\mathtt{get} : W \to W_{\mathsf{P}}$ and $\mathtt{set} : W \to W_{\mathsf{P}} \to W$.

By explicitly defining threaded simulation using TKMR, we are able to describe the memory evolutions for not only function calls, but also thread-local executions as discussed below. We often omit the superscript P of the accessibility symbol $\leadsto$.

Definition 4.3 (Threaded Forward Simulation). The threaded forward simulation is defined only between semantics using the same interfaces on both sides , i.e. in the form of $L_1 : A_1 \twoheadrightarrow A_1$ and $L_2 : A_2 \twoheadrightarrow A_2$. The internal invariant is defined using a Kripke Relation using two worlds $\mathcal{K}_{W_1,W_2}(A, B) : W_1 \to W_2 \to \{S \mid S \subseteq A \times B\}$. Given the threaded simulation convention $\mathbb{R}_{\mathsf{P}} : A_1 \Leftrightarrow A_2$, $L_1 \leqslant_{\mathbb{R}_{\mathsf{P}} \twoheadrightarrow \mathbb{R}_{\mathsf{P}}} L_2$ holds if there exists some Kripke relation $R \in \mathcal{K}_{W,W_{\mathsf{P}}}(S_1, S_2)$ that satisfies:

(1) $\forall\ w_B\ q_1\ q_2,\ (q_1, q_2) \in \mathbb{R}_B^q(w_B) \Rightarrow (q_1 \in D_1 \Leftrightarrow q_2 \in D_2)$

(2) $\forall\ w_B\ q_1\ q_2\ s_1,\ (q_1, q_2) \in \mathbb{R}_B^q(w_B) \Rightarrow (q_1, s_1) \in I_1 \Rightarrow$
$\exists\ s_2, (s_1, s_2) \in R(w_B, \mathtt{get}(w_B)) \wedge (q_2, s_2) \in I_2.$

(3) $\forall\ w_B\ w_{\mathsf{P}}\ s_1\ s_2\ r_1,\ (s_1, s_2) \in R(w_B, w_{\mathsf{P}}) \Rightarrow (s_1, r_1) \in F_1 \Rightarrow$
$\exists\ w_{\mathsf{P}}'\ r_2, (r_1, r_2) \in \mathbb{R}_B^r(\mathtt{set}(w_B, w_{\mathsf{P}}')) \wedge (s_2, r_2) \in F_2 \wedge \mathtt{get}(w_B) \leadsto_e w_{\mathsf{P}}' \wedge w_{\mathsf{P}} \leadsto_i w_{\mathsf{P}}'.$

(4) $\forall\ w_B\ w_{\mathsf{P}}\ s_1\ s_2\ t\ s_1',\ (s_1, s_2) \in R(w_B, w_{\mathsf{P}}) \Rightarrow s_1 \xrightarrow{t} s_1' \Rightarrow \exists\ s_2', (s_1', s_2') \in R(w_B, w_{\mathsf{P}}) \wedge s_2 \xrightarrow{t}{}^{*} s_2'.$

(5) $\forall\ w_B\ w_{\mathsf{P}}\ s_1\ s_2\ q_1,\ (s_1, s_2) \in R(w_B, w_{\mathsf{P}}) \Rightarrow (s_1, q_1) \in X_1 \Rightarrow$
$\exists w_A\ q_2,\ (q_1, q_2) \in \mathbb{R}_A^q(w_A) \wedge (s_2, q_2) \in X_2\ \wedge w_{\mathsf{P}} \leadsto_i \mathtt{get}(w_A) \wedge$
$\forall\ w_{\mathsf{P}}'\ r_1\ r_2\ s_1', \mathtt{get}(w_A) \leadsto_e w_{\mathsf{P}}' \Rightarrow (r_1, r_2) \in \mathbb{R}_A^r(\mathtt{set}(w_A, w_{\mathsf{P}}')) \Rightarrow (s_1, r_1, s_1') \in Y_1 \Rightarrow$

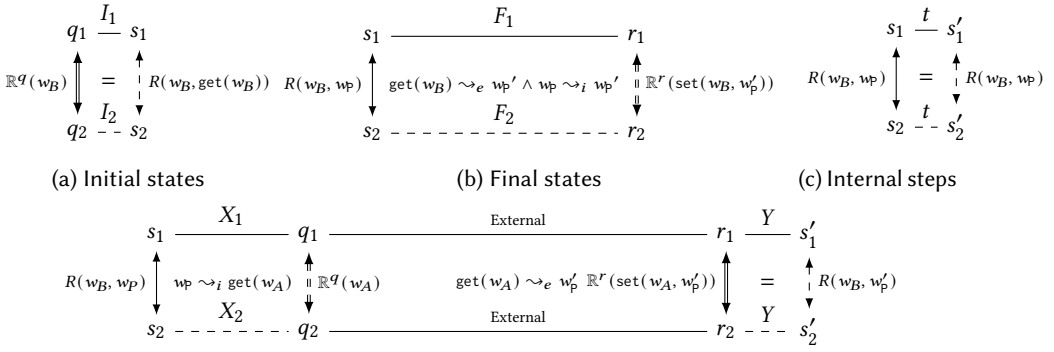


Fig. 10. Simulation Diagrams for Threaded Forward Simulation

$$\exists\, s_2',\ (s_1', s_2') \in R(w_B, w_\mathsf{P}') \wedge (s_2, r_2, s_2') \in Y_2.$$

The internal invariant $R(w_B, w_\mathsf{P})$ remembers the memories from initial queries in w_B and the memories *when the last time the module is invoked* in w_P. When $(s_1, s_2) \in R(w_B, w_\mathsf{P})$ holds, it often implies that the current memory world w_s satisfies both the external guarantee from w_B and the internal guarantee from w_P, i.e. $\mathtt{get}(w_B) \rightsquigarrow_e w_s$ and $w_\mathsf{P} \rightsquigarrow_i w_s$. Property (1) requires related queries have the same validity. The remaining properties are depicted in Fig. 10.

For initial states (Fig. 10a), we initialize R using the memories states in $\mathtt{get}(w_B)$ from the queries. For final states (Fig. 10b), the semantics return replies r_1 and r_2, we need to prove that they are related by $\mathtt{set}(w_B, w_\mathsf{P}')$ where the memory world in w_B is updated by a existential world w_P' containing current memory states. The other information in w_B is unchanged. The *external guarantee* $\mathtt{get}(w_B) \rightsquigarrow_e w_P'$ for whole execution and the *internal guarantee* $w_\mathsf{P} \rightsquigarrow_i w_\mathsf{P}'$ for the recent internal execution are satisfied at the same time. The proof of final states is straightforward using the general definition of R by picking w_P' as w_s. For internal steps (Fig. 10c), although the memories in s_1' and s_2' are changed, we still relate them using the same w_B and w_P. For external calls (Fig. 10d), when the semantics issue queries q_1 and q_2 related by w_A, we guarantee the *internal guarantee* of recent internal execution by $w_\mathsf{P} \rightsquigarrow_i \mathtt{get}(w_A)$. After the external call, we require that the replies are related by $\mathtt{set}(w_A, w_\mathsf{P}')$ where w_P' stands for the updated memory states, other information in w_A is preserved. Given the rely condition $\mathtt{get}(w_A) \rightsquigarrow_e w_\mathsf{P}'$, we reestablish the internal invariant $(s_1', s_2') \in R(w_B, w_\mathsf{P}')$ using the memory world w_P' from replies.

Since threaded simulation is only defined for symmetric type $\mathbb{R} \twoheadrightarrow \mathbb{R}$, we write $L_1 \leqslant_\mathbb{R} L_2$ for $L_1 \leqslant_{\mathbb{R}\twoheadrightarrow\mathbb{R}} L_2$ in the rest of the paper. As we discussed before, Definition 4.3 does not directly describe the multi-threaded semantics. The concurrent primitives like `pthread_create`, `pthread_join` and even the context switch operation are treated as external calls. We discuss how to prove the simulation between multi-threaded semantics using threaded simulation in §4.3.1.

Definition 4.4 (Threaded Direct Refinement). The threaded version of direct refinement is explicitly defined using `tinjp` as $\mathsf{CA}_\mathtt{tinjp} : C \Leftrightarrow \mathcal{A} = \langle W_\mathsf{CA}, \mathbb{R}^q_\mathsf{CA}, \mathbb{R}^r_\mathsf{CA}, \mathtt{tinjp}\rangle$ where $W_\mathsf{CA} = (W_\mathtt{tinjp}, \mathtt{sig}, \mathtt{regset})$. The `get` and `set` operations for the sub world $W_\mathtt{tinjp}$ is simply defined as getting and setting the first component of a $w \in W_\mathsf{CA}$.

The definition of $\mathbb{R}^q_\mathsf{CA}$ ($\mathbb{R}^r_\mathsf{CA}$) is almost the same as in Definition 3.4. The only difference is that we do not need to define the memory evolution from queries to replies as a part of $\mathbb{R}^r_\mathsf{CA}$ because it is now explicitly defined in the threaded simulation (Definition 4.3). $(res@m_1', rs'@m_2') \in \mathbb{R}^r_\mathsf{CA}(w)$ now requires that the world w contains exactly the memories of the replies, i.e. $\mathtt{get}(w) = (j', m_1', m_2')$.

4.3 Compositionality of TKMR and Threaded Direct Refinements

We present the horizontal and vertical composition of threaded direct refinement while the essence is how to compose the TKMR which describes the evolution of memory states.

4.3.1 Horizontal Composition. The horizontal composition of threaded direct refinements can be divided into two steps. We first compose open modules assuming that they run on the same thread. Then we transfer the composed open semantics into a closed LTS running on a multi-threaded abstract machine. The closed simulation between multi-threaded semantics can be derived from the threaded direct refinement which properly protects the private memory while sharing the public memory between different threads.

Theorem 4.5 (Module Linking). *Given* $L_1, L_1' : A_1 \twoheadrightarrow A_1$, $L_2, L_2' : A_2 \twoheadrightarrow A_2$ *and threaded simulation convention* $\mathbb{R} : A_1 \Leftrightarrow A_2$, *the threaded simulation can be horizontally composed as:*

$$L_1 \leqslant_{\mathbb{R}} L_2 \Rightarrow L_1' \leqslant_{\mathbb{R}} L_2' \Rightarrow L_1 \oplus L_1' \leqslant_{\mathbb{R}} L_2 \oplus L_2'$$

We omit the concrete definition of the semantic linking $L_1 \oplus L_2$ between open LTS defined in CompCertO [12]. Essentially, when L_1 calls a function defined in L_2, the *rely* condition of L_1 is satisfied by the *external guarantee* of L_2 and vice versa. The difference between this theorem and the composition of sequential simulation is the necessity to compose *internal guarantee* conditions. It is achieved with the transitivity of $\rightsquigarrow_i$.

The multi-threaded semantics is defined as a closed LTS, i.e. the semantic domain of complete sequential program from CompCert [14]. A closed LTS is a tuple $\langle S, I, F, \rightarrow \rangle$ containing program state S, predicates for initial and final state I, F and the internal transition $\rightarrow$.

Definition 4.6 (Thread Linking). We define *thread linking function* $\mathtt{MT}_C$ and $\mathtt{MT}_{\mathcal{A}}$ to transfer C and assembly open semantics into multi-threaded semantics. We omit the concrete definition for simplicity and present the key ideas using C semantics, the construction for assembly is similar.

Given $L : C \twoheadrightarrow C$, $\mathtt{MT}_C(L) = \langle S_M, I_M, F_M, \rightarrow_M \rangle$ has a program state S_M including current thread id t and a list of *thread states* which are defined using the program state of L. The current global memory can be found in the current thread state. The initial state I_M and final state F_M are defined using the *query* and *reply* of `main` function on the main thread. The main thread uses X to take a query to `main` as initialization and finally issues a reply using Y to produce the return value.

The transition step $\rightarrow_M$ has three cases. The internal step is defined as a step of L on the current running thread. The thread creation step happens if the current thread calls `pthread_create` (as an external call). We use its arguments and the current memory state to create a query $q_{\mathtt{str}}$ to `start_routine` function and use it to initialize a new thread state on the newly allocated thread. Finally, for context switches, the current thread can switch out at any point by calling the `yield` primitive as external call. Similarly, the target thread is waiting for a reply of a previous `yield`. We simply pass the current global memory (by constructing the query into a reply) to wake up the target thread and put the current thread into waiting state.

Theorem 4.7 (Thread Linking Correctness).

$$\forall\, L_1 : C \twoheadrightarrow C, L_2 : \mathcal{A} \twoheadrightarrow \mathcal{A},\ L_1 \leqslant_{\mathbb{C}} L_2 \Rightarrow \mathtt{MT}_C(L_1) \leqslant \mathtt{MT}_{\mathcal{A}}(L_2)$$

Here $\mathbb{C}$ is the simulation convention derived from CompCertOC. It is defined using $\mathtt{CA}_{\mathtt{tinjp}}$ along with several invariants on source and target semantics. The formal definition can be found in §5.3. The definition of closed simulation $\leqslant$ from CompCert is similar as Definition 3.1 while we need to maintain an internal invariant to preserve the execution of multi-threaded semantics. Each thread state is related using the invariant from the direct refinement, i.e. we have worlds $w_B \in W_{\mathtt{CA}}$ and

$w_P \in W_{\texttt{tinjp}}$ to represent the memory states at the initialization and the last invocation of each thread. At any point, we have one current thread running and others are waiting to be resumed.

The critical point is to ensure that when the current thread t yields out, its memory world w_P^t can satisfy the *rely* condition of *any* waiting thread t', i.e. $\forall\ t \neq t', w_P^{t'} \rightsquigarrow_e w_P^t$ [2]. In other words, we need to guarantee that the *private* memory regions of all waiting threads are not changed since their last switch point. This is trivially correct from the perspective of a waiting thread. Because its private memory is protected by the internal guarantee condition $\rightsquigarrow_i$ of any other thread (e.g. the thread A in Fig. 7). Therefore, these regions are guaranteed to be unchanged until it is resumed.

4.3.2 Vertical Composition. For threaded simulation, the trivial composition theorem still holds:

Theorem 4.8. *If* $L_1 : A_1 \twoheadrightarrow A_1, L_2 : A_2 \twoheadrightarrow A_2, L_3 : A_3 \twoheadrightarrow A_3$ *and* $\mathbb{R} : A_1 \Leftrightarrow A_2, \mathbb{S} : A_2 \Leftrightarrow A_3$, *then*

$$L_1 \leqslant_{\mathbb{R}} L_2 \Rightarrow L_2 \leqslant_{\mathbb{S}} L_3 \Rightarrow L_1 \leqslant_{\mathbb{R}\cdot\mathbb{S}} L_3.$$

Here $(\_ \cdot \_)$ is the concatenation of threaded simulation convention s.t. $\mathbb{R} \cdot \mathbb{S} = \langle W_{\mathbb{R}} \times W_{\mathbb{S}}, \mathbb{R}^q \cdot \mathbb{S}^q, \mathbb{R}^r \cdot \mathbb{S}^r, \mathsf{P}_{\mathbb{R}} \cdot \mathsf{P}_{\mathbb{S}} \rangle$ where $(q_1, q_3) \in \mathbb{R}^q \cdot \mathbb{S}^q \Leftrightarrow \exists\ q_2, (q_1, q_2) \in \mathbb{R}^q (q_1, q_2)$ and $(q_2, q_3) \in \mathbb{R}^q (q_2, q_3)$. (same for $\mathbb{R}^r \cdot \mathbb{S}^r$). The TKMRs are similarly composed such that $(w_1, w_2) \rightsquigarrow_i (w_1', w_2') \Leftrightarrow w_1 \rightsquigarrow_i w_2 \wedge w_1' \rightsquigarrow_i w_2'$ (same for $\rightsquigarrow_e$).

However, the composed interface exposes the internal compilation and weakens the compiler correctness. In order to get the direct refinement between C and assembly semantics, we need to define the refinement between simulation conventions. Given $\mathbb{R}, \mathbb{S} : A_1 \Leftrightarrow A_2$, we write $\mathbb{R} \sqsubseteq \mathbb{S}$ for $\mathbb{R}$ is refined by $\mathbb{S}$. We can refine the interface to get a new simulation:

Theorem 4.9. *Given* $L_1 : A_1 \twoheadrightarrow A_1, L_2 : A_2 \twoheadrightarrow A_2$ *and* $\mathbb{R}, \mathbb{S} : A_1 \Leftrightarrow A_2$,

$$\mathbb{R} \sqsubseteq \mathbb{S} \Rightarrow L_1 \leqslant_{\mathbb{R}} L_2 \Rightarrow L_1 \leqslant_{\mathbb{S}} L_2$$

The concrete definition of the $\sqsubseteq$ is complicated and can be found in our artifact. We illustrate the the key idea of it by an example. Given two verified optimization pass for C-level semantics $L_1 \leqslant_{\mathtt{c_{tinjp}}} L_2$ and $L_2 \leqslant_{\mathtt{c_{tinjp}}} L_3$ where $\mathtt{c_{tinjp}} : C \Leftrightarrow C$ is a simulation convention between C semantics using `tinjp`. We are able to compose them into $L_1 \leqslant_{\mathtt{c_{tinjp}}} L_3$ by showing $\mathtt{c_{tinjp}} \cdot \mathtt{c_{tinjp}} \sqsubseteq \mathtt{c_{tinjp}}$. The critical part of this refinement is to show that the TKMR `tinjp` is transitive. Other simulation conventions using `tinjp` can be similarly composed to get $\mathtt{CA_{tinjp}}$ for compiler correctness.

Theorem 4.10. $\mathtt{tinjp} \cdot \mathtt{tinjp} \sqsubseteq \mathtt{tinjp}$

The key point is that the *private* regions get bigger when memory injections get composed as depicted in Fig. 11. Internally, we have $(w_{12}, w_{23}) \in W_{\mathtt{tinjp}\cdot\mathtt{tinjp}}$ where $w_{12} = (j_{12}, m_1, m_2)$ and $w_{23} = (j_{23}, m_2, m_3)$. The memory blocks for variables a, b and c in different memories are depicted as boxes. a is preserved from m_1 to m_3, b is removed from m_2 to m_3 and c is removed from m_1 to m_2. On the left side of Fig. 11, b_2 (i.e. block for b in m_2) and c_2 are *private* (denoted by shaded boxes) according to w_{23} and w_{12}, respectively. On the right side, we compose the worlds w_{12} and w_{23} into $w_{13} = (j_{13}, m_1, m_3)$ such that b_1 and c_3 *become* private blocks.

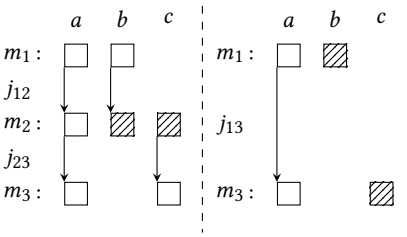


Fig. 11. Composed Memory Worlds

This property suffices for proving the *rely condition* for internal worlds. Consider the composed simulation $L_1 \leqslant_{\mathtt{c_{tinjp}}\cdot\mathtt{c_{tinjp}}} L_3$. The memory states are related by w_{12} and w_{23} through m_2 in the internal invariant. When external calls happen in w_{13}, the *rely* of environment $w_{13} \rightsquigarrow_e w_{13}'$ ensures

[2]The $\rightsquigarrow_e$ here is a slightly different accessibility which requires the memories of w^t and $w^{t'}$ have different thread id.

Table 1. Significant Passes of CompCertOC

Languages/Passes	Interfaces/Conventions	Language/Pass	Interfaces/Conventions
Clight	$C \twoheadrightarrow C$	(After Inlining)	(After Inlining)
Self-Sim	$\mathtt{ro} \twoheadrightarrow \mathtt{ro}$	Constprop	$\mathtt{ro} \cdot \mathtt{c_{tinjp}} \twoheadrightarrow \mathtt{ro} \cdot \mathtt{c_{tinjp}}$
SimplLocals	$\mathtt{c_{tinjp}} \twoheadrightarrow \mathtt{c_{tinjp}}$	CSE	$\mathtt{ro} \cdot \mathtt{c_{tinjp}} \twoheadrightarrow \mathtt{ro} \cdot \mathtt{c_{tinjp}}$
Csharpminor	$C \twoheadrightarrow C$	Deadcode	$\mathtt{ro} \cdot \mathtt{c_{tinjp}} \twoheadrightarrow \mathtt{ro} \cdot \mathtt{c_{tinjp}}$
Cminorgen	$\mathtt{c_{tinjp}} \twoheadrightarrow \mathtt{c_{tinjp}}$	Unusedglob	$\mathtt{c_{tinjp}} \twoheadrightarrow \mathtt{c_{tinjp}}$
Cminor	$C \twoheadrightarrow C$	Allocation	$\mathtt{wt} \cdot \mathtt{c_{ext}} \cdot \mathtt{CL} \twoheadrightarrow \mathtt{wt} \cdot \mathtt{c_{ext}} \cdot \mathtt{CL}$
Selection	$\mathtt{wt} \cdot \mathtt{c_{ext}} \twoheadrightarrow \mathtt{wt} \cdot \mathtt{c_{ext}}$	**LTL**	$\mathcal{L} \twoheadrightarrow \mathcal{L}$
CminorSel	$C \twoheadrightarrow C$	Tunneling	$\mathtt{ltl_{ext}} \twoheadrightarrow \mathtt{ltl_{ext}}$
RTLgen	$\mathtt{c_{ext}} \twoheadrightarrow \mathtt{c_{ext}}$	**Linear**	$\mathcal{L} \twoheadrightarrow \mathcal{L}$
RTL	$C \twoheadrightarrow C$	Stacking	$\mathtt{LM_{tinjp}} \twoheadrightarrow \mathtt{LM_{tinjp}}$
Self-Sim	$\mathtt{c_{tinjp}} \twoheadrightarrow \mathtt{c_{tinjp}}$	**Mach**	$\mathcal{M} \twoheadrightarrow \mathcal{M}$
Tailcall	$\mathtt{c_{ext}} \twoheadrightarrow \mathtt{c_{ext}}$	Asmgen	$\mathtt{mach_{ext}} \cdot \mathtt{MA} \twoheadrightarrow \mathtt{mach_{ext}} \cdot \mathtt{MA}$
Inlining	$\mathtt{c_{tinjp}} \twoheadrightarrow \mathtt{c_{tinjp}}$	**Asm**	$\mathcal{A} \twoheadrightarrow \mathcal{A}$

that b_1 and c_3 are unchanged. Therefore, we can only update the value in a_2 and keep b_2 and c_2 unchanged to construct an updated m_2' and prove $w_{12} \rightsquigarrow_e w_{12}'$ and $w_{23} \rightsquigarrow_e w_{23}'$.

However, this property becomes a drawback for *guarantee* conditions. The internal execution $w_{12} \rightsquigarrow_i w_{12}'$ may change b_1 while we need it to be protected in $w_{13} \rightsquigarrow_i w_{13}'$ if b_1 is a *thread-external* block. In order to avoid such situation, we maintain an invariant stating the absence of mappings like b and c *when they are thread-external blocks*. It is possible because m_2 is hidden from other threads and we only construct *public thread-external* blocks for it. $\rightsquigarrow_e$ has the same problem but we can exclude such mappings by specific construction of w_{12} and w_{23} from w_{13} at the initial state. The more concrete proof can be found in the supplementary materials.

5 CompCert for Concurrent Open Modules

We now discuss how to verify the compiler passes of CompCert and compose them into the direct refinement $\preccurlyeq_{\mathtt{ca}}$ as the correctness of CompCertOC. CompCert compiles `Clight` programs into `Asm` programs through 19 passes, including optimization passes on the `RTL` intermediate language. Our development is based on the CompCertO with direct refinements [27]. First, we modify the proofs for all passes by proving the internal guarantees. The proofs share similar pattern as discussed in §5.1. The verification of optimization passes using value analysis is especially non-trivial, the details can be found in the supplementary materials. Second, we prove the properties for refining the threaded simulation conventions around `tinjp`. Therefore, we are able to absorb other TKMRs into `tinjp` and compose the semantics invariants and structured simulation conventions around `tinjp`. Finally, the concatenation of all passes can be refined into a single simulation convention $\mathbb{C}$ (which is based on $\mathtt{CA_{tinjp}}$ defined in Definition 4.4) as the interface of threaded direct refinements.

5.1 Threaded Open Simulation for Individual Passes

The important compiler passes and their simulation types are listed in Table 1 together with the intermediate languages and their interfaces (which are in bold fonts). The passes marked in red are the optimization passes at the RTL level. Notice that the language and simulation interfaces are symmetric on incoming and outgoing sides, which is necessary for supporting context switches. We additionally insert two self-simulations (in blue) for refining the composed simulation convention into a direct refinement. Several passes using the identity simulation convention do not affect the composition and hence are omitted in Table 1.

5.1.1 Simulation Conventions and Semantic Invariants. We first discuss the simulation conventions presented in Table 1. The simulation conventions in the form of $\mathtt{I_P} : \mathcal{X} \Leftrightarrow \mathcal{X}$ uses TKMR $\mathtt{P}$ to relate the source and target semantics using the same interface $\mathcal{X}$. `ext` is a simplified version of `tinjp` which assumes source and target memories have the same structure. Therefore no private memory needs to be protected and the only rely-guarantee condition is the invariance of thread id.

$\mathtt{CL} : \mathcal{C} \Leftrightarrow \mathcal{L}$ and $\mathtt{MA} : \mathcal{M} \Leftrightarrow \mathcal{A}$ are *structure conventions* which relate queries and replies in different types. They assume that the memory states in different interfaces are the same. While another structure convention $\mathtt{LM_{tinjp}} : \mathcal{L} \Leftrightarrow \mathcal{M}$ for `Stacking` pass is parameterized over `tinjp`. This is because the stack pointer, return address and outgoing arguments are allocated on the stack frames after this pass. The source and target memories are different and these private regions in target memory need to be protected.

`ro` and `wt` are *semantic invariants* which require that the source and target queries (replies) are the same. They are used together with another simulation convention to assume some invariants on the *source* semantics. `wt` ensures that the arguments and return value are well-typed. `ro` ensures that the dynamic values of read-only *constants* are always the same as its initial value.

5.1.2 Verification of `Stacking` *pass.* We are able to reuse the proofs in CompCertO to prove thread forward simulation for each pass by extending the invariant using $w_\mathtt{P}$ to prove the internal guarantee condition. The proof for different passes have similar pattern. We illustrate how to prove internal guarantee conditions using `Stacking` as an example. Its correctness is defined as:

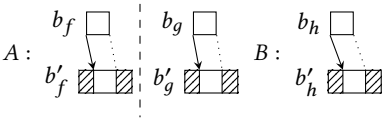


Fig. 12. Snapshot for `Stacking` pass

Lemma 5.1. $\forall (M : \mathtt{Linear})(M' : \mathtt{Mach}), \mathtt{Stacking}(M) = M' \Rightarrow [\![M]\!] \leqslant_{\mathtt{LM_{tinjp}}} [\![M']\!]$.

$\mathtt{LM_{tinjp}}$ is similar to $\mathtt{CA_{tinjp}}$ (Definition 4.4) but with a different type. its world type $W_\mathtt{LM}$ also contains $W_\mathtt{tinjp}$ for memories, the function signature *sg* and the register set *rs*. $\mathtt{LM_{tinjp}}$ also requires that the callee-saved registers are the same from the target query to reply. In order to prove this simulation, we need to find an invariant $R : \mathcal{K}_{W_\mathtt{LM}, W_\mathtt{tinjp}}$. We simplify the source and target program states into just memories by ignoring the other parts like control stack or registers. $(m_1, m_2) \in R(w_B, w_\mathtt{P})$ is essentially defined as $\exists j, w_B \rightsquigarrow_e (j, m_1, m_2) \wedge w_\mathtt{P} \rightsquigarrow_i (j, m_1, m_2)$. We write w_s for the internal memory world (j, m_1, m_2) in following discussion. The initial states (Fig. 10a)) and final states (Fig. 10b) cases are straightforward according to this definition. For external calls (Fig. 10d), The world w_s for related states s_1 and s_2 corresponds to the memories of issued queries $\mathtt{get}(w_A)$. After external execution, we compose $w_B \rightsquigarrow_e \mathtt{get}(w_A)$ and $\mathtt{get}(w_A) \rightsquigarrow_e w'_\mathtt{P}$ to get $w_B \rightsquigarrow_e w'_\mathtt{P}$ ($\rightsquigarrow_e^{\mathtt{tinjp}}$ is transitive by its definition). $w'_\mathtt{P} \rightsquigarrow_i w'_\mathtt{P}$ trivially holds in $R(w_B, w'_\mathtt{P})$.

We illustrate how to prove the internal step (Fig. 10c) by a snapshot of stack frames in Fig. 12. The shaded regions are *private* regions at the target level. The proof goal is to show the internal execution of an open module can only modify the private regions *allocated by itself*. Here M and M' are called by f and run function g on thread A. h is another function runs on thread B. The $w_B \rightsquigarrow_e w_s$ and $w_\mathtt{P} \rightsquigarrow_i w_s$ from internal invariant protect the private regions in b'_f and b'_h, respectively. Although g can visit the stack frames of f and h, the source program M can only access the public regions in b_f and b_h. Therefore, forward simulation ensures that M' does not change *local private data* in b'_f and *external private data* in b'_h.

5.2 Properties for Refining Thread Simulation Conventions

The trivial concatenation of threaded simulation conventions is discussed in Theorem 4.8. We discuss the necessary refinement lemmas of threaded simulation conventions in this subsection.

First, the following lemma composes C-level conventions using TKMRs around $\mathtt{c}_{\mathtt{tinjp}}$:

Lemma 5.2. (1) $\mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{tinjp}} \sqsubseteq \mathtt{c}_{\mathtt{tinjp}}$ (2) $\mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{ext}} \sqsubseteq \mathtt{c}_{\mathtt{ext}}$
(3) $\mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{tinjp}} \sqsubseteq \mathtt{c}_{\mathtt{tinjp}}$ (4) $\mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \sqsubseteq \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}}$.

Here property (1) is a direct result of $\mathtt{tinjp} \cdot \mathtt{tinjp} \sqsubseteq \mathtt{tinjp}$ (Theorem 4.10). (2) and (3) are trivial because ext does not introduce any private memory. (4) is an extended version of (1) composing the ro invariant on memories together with tinjp.

Lemma 5.3. *For* $K \in \{\mathtt{ext}, \mathtt{tinjp}\}$, (1) $\mathtt{c}_K \cdot \mathtt{CL} \equiv \mathtt{CL} \cdot \mathtt{ltl}_K$ (2) $\mathtt{mach}_K \cdot \mathtt{MA} \equiv \mathtt{MA} \cdot \mathtt{asm}_K$.

Here $\mathbb{R} \equiv \mathbb{S} \Leftrightarrow \mathbb{R} \sqsubseteq \mathbb{S} \wedge \mathbb{S} \sqsubseteq \mathbb{R}$. Since simulation conventions CL and MA assume the same memory for source and target, we can freely transfer the conventions parameterized by TKMR through them. For the special structural $\mathtt{LM}_{\mathtt{tinjp}}$, we compose other simulation conventions *onto* it as follows:

Lemma 5.4. (1) $\mathtt{ltl}_{\mathtt{tinjp}} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{LM}_{\mathtt{tinjp}} \sqsubseteq \mathtt{LM}_{\mathtt{tinjp}}$ (2) $\mathtt{CL} \cdot \mathtt{LM}_{\mathtt{tinjp}} \cdot \mathtt{MA} \sqsubseteq \mathtt{CA}_{\mathtt{tinjp}}$

Property (1) is based on the same compositionality of TKMRs as property (2) in Lemma 5.2; (2) composes the structural relations to hide the intermediate languages. Finally, for the source invariants, the permutation and elimination lemmas for wt can be proved as follows:

Lemma 5.5. (1) $\mathtt{ro} \cdot \mathtt{wt} \equiv \mathtt{wt} \cdot \mathtt{ro}$ (2) *For* $K \in \{\mathtt{tinjp}, \mathtt{ext}\}$, $\mathtt{c}_K \cdot \mathtt{wt} \sqsubseteq \mathtt{wt} \cdot \mathtt{c}_K$
(3) $\mathtt{wt} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{wt} \sqsubseteq \mathtt{wt} \cdot \mathtt{c}_{\mathtt{tinjp}}$

5.3 Composing the Threaded Direct Refinement for CompCertOC

We first insert the self-simulations as presented in Table 1. The self-simulation for each intermediate language can be trivially proved using the same definition of invariant R as discussed in §5.1.2. The trivial composition of all passes results into $L_1 \leqslant_{\mathbb{R}} L_2$ where

$$\mathbb{R} = \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{LM}_{\mathtt{tinjp}} \cdot \mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA}$$

The simulation convention for the whole compiler is formally defined as:

$$\mathbb{C} = \mathtt{ro} \cdot \mathtt{wt} \cdot \mathtt{CA}_{\mathtt{tinjp}} \cdot \mathtt{asm}_{\mathtt{ext}}$$

$\mathtt{CA}_{\mathtt{tinjp}}$ is already defined in Definition 4.4. Note that we ignore the the source invariants ro, wt and the last $\mathtt{asm}_{\mathtt{ext}}$ in previous sections for simplicity. It is straightforward to prove that a well-behaved C semantics preserves ro and wt. $\mathtt{asm}_{\mathtt{ext}}$ is also irrelevant because the self-simulation holds for any assembly program. The final correctness of CompCertOC is presented below:

Theorem 5.6 (Compiler Correctness).

$$\forall\ (M : \mathtt{Clight})(M' : \mathtt{Asm}),\ \mathtt{CompCertOC}(M) = M' \Rightarrow [\![M]\!] \leqslant_{\mathbb{C}} [\![M']\!].$$

Based on Theorem 4.9, we only need to show $\mathbb{R} \sqsubseteq \mathbb{C}$ which is proved by a sequence of refined simulation conventions $\mathbb{R} \sqsubseteq \mathbb{R}_1 \sqsubseteq \ldots \sqsubseteq \mathbb{R}_n \sqsubseteq \mathbb{C}$ as presented above. The red letters mark the simulation conventions refined in each step using properties defined in §5.2. Detailed discussion can be found in supplementary materials.

(1) $\mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{LM}_{\mathtt{tinjp}} \cdot \mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA}$
(2) $\mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{LM}_{\mathtt{tinjp}} \cdot \mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA}$
(3) $\mathtt{ro} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{LM}_{\mathtt{tinjp}} \cdot \mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA}$
(4) $\mathtt{wt} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{LM}_{\mathtt{tinjp}} \cdot \mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA}$
(5) $\mathtt{wt} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{LM}_{\mathtt{tinjp}} \cdot \mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA}$
(6) $\mathtt{ro} \cdot \mathtt{wt} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{tinjp}} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{LM}_{\mathtt{tinjp}} \cdot \mathtt{MA} \cdot \mathtt{asm}_{\mathtt{ext}}$
(7) $\mathtt{ro} \cdot \mathtt{wt} \cdot \mathtt{CA}_{\mathtt{tinjp}} \cdot \mathtt{asm}_{\mathtt{ext}}$

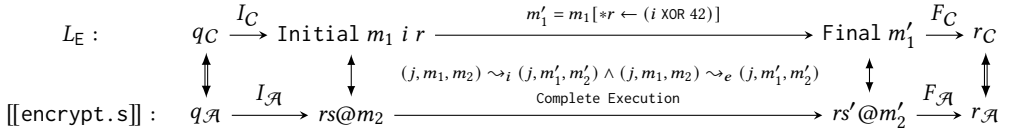


Fig. 13. Simulation Diagram for $L_E \leqslant_{CA_{tinjp}}$ [[encrypt.s]]

6 Application and Evaluation

In this section, we give a formal verification of the running example as depicted in Fig. 3. We also evaluate our Coq development of CompCertOC comparing to CompCertO with direct refinement.

6.1 Verification of Heterogeneous Multi-threaded Modules

We first define the C-level specification L_E for encrypt.s.

Definition 6.1. LTS of L_E is defined as

S_E := {Initial m i r, Final m};
I_E := {(Vptr(b_e, 0)[int → ptr → void]([i, r]))@m, Initial m i r)};
$\rightarrow_E$:= {Initial m i r, Final m') | $m' = m[*r \leftarrow (i \text{ XOR } 42)]$}
F_E := {(Final m, Vundef@m)}.

Here X_E and Y_E are empty sets because encrypt does not contain external calls. We then prove the threaded direct refinement of encrypt as:

Lemma 6.2. $L_E \leqslant_{\mathbb{C}}$ [[encrypt.s]].

The simulation convention $\mathbb{C}$ is expanded as ro · wt · CA_{injp} · asm_{ext}. The source invariants on L_E holds according to its definition. The self-simulation using asm_{ext} holds for any program written in CompCert assembly. The proof of $L_E \leqslant_{CA_{injp}}$ [[encrypt]] is depicted in Fig. 13. We need to verify the internal and external guarantee conditions of the whole execution which are straightforward because encrypt does not allocate any new blocks and only changes the public memory pointed by r in both source and target semantics.

We first compose Lemma 6.2 with the compilation correctness of client.c and server.c using Theorem 4.5 to get the thread direct refinement between open semantics:

Lemma 6.3 (Threaded Direct Refinement for Composed Modules).

$$[[\texttt{client.c}]] \oplus [[\texttt{server.c}]] \oplus L_E \leqslant_{\mathbb{C}} [[\texttt{client.s}]] \oplus [[\texttt{server.s}]] \oplus [[\texttt{encrypt.s}]]$$

We also prove the equivalence of syntactic and semantics linking of assembly modules:

Theorem 6.4. $\forall (M\ M' : \texttt{Asm}), [[M]] \oplus [[M']] \leqslant_{\texttt{id}} [[M + M']]$.

Where id is the identity simulation convention which can be absorbed into any other convention. We first vertically compose Lemma 6.3 and Lemma 6.4 and eliminate id by $\mathbb{C} \cdot \texttt{id} \sqsubseteq \mathbb{C}$. Finally we apply the correctness of thread linking (Theorem 4.7) to get the final result:

Theorem 6.5 (Behavior Refinement for the Complete Program).

$$\texttt{MT}_C([[\texttt{client.c}]] \oplus [[\texttt{server.c}]] \oplus L_E) \leqslant \texttt{MT}_A([[\texttt{client.s} + \texttt{server.s} + \texttt{encrypt.s}]])$$

6.2 Evaluation

Our Coq development took about 9 person-months and contains 15.8k lines of code (LOC) on top of CompCertO. We added 0.9k LOC to implement the multi-stack memory model thanks to the generality of Nominal Memory Model, 2.1k LOC to define the framework of threaded forward

Table 2. Comparison of Work on Verified Compositional Compilation

	CCC	CCM	CCO	CCTSO	CCX	CASCC	**CCOC**
Concurrency	No	No	No	Yes	Yes	Yes	**Yes**
Stack Sharing	No	No	No	Yes	No	No	**Yes**
Direct Refinement	No	No	Yes	Closed	No	No	**Yes**
Adequacy	No	Yes	Yes	Closed	Yes	No	**Yes**
Vertical Composition	Yes	RUSC	Yes	No	CCAL	Yes	**Yes**
Horizontal Composition	Yes	RUSC	Yes	No	CCAL	Yes	**Yes**
Heterogeneous Modules	Yes	Yes	Yes	No	Yes	Yes	**Yes**

simulation. We modified the proof of 14 compilation passes (listed in Table 1 using TKMR) by *adding* 2.9k LOC on the top of 20.9k LOC in CompCertO, which is about 14% increase. The proof of different passes share the same pattern as we discussed for `Stacking` in §5.1.2. We only need to ensure that the modules only change private memory allocated by themselves. We added 7.6k LOC to verify the refinements of thread simulation conventions while the majority are used to prove the vertical composition of TKMRs (Lemma 5.2). Finally, we defined the multi-threaded semantics based on open semantics and proved the thread linking by adding 2.3k LOC.

7 Related Work and Conclusion

In this work, we are concerned with VCC of imperative programs with pointers. Compared with functional programs or languages without pointers/references, this significantly complicates the memory protection on shared stacks. In this setting, we compare CompCertOC (CCOC) with other work on VCC not supporting concurrency, including Compositional CompCert (CCC) [23], CompCertM (CCM) [22], CompCertO (CCO) [12, 27], and those with concurrency, including Thread-safe CompCertX (CCX) [5], CASCompCert [8] and CompCertTSO [21]. Table 2 summarizes this comparison where each row displays one feature the verified compilers support or do not support (texts in brown color indicate that a weaker variant of the feature is supported). "Direct refinement" means that the semantics of source and target programs are related at their *native* language interfaces without exposing any intermediate semantics; it is important for smooth support of compositionality and for usability by third-parties. "Adequacy" means that the syntactic and semantic composition of target programs are equivalent (i.e., Theorem 6.4); it is critical for connection with the binary objects produced by linkers. We elaborate on the comparison below.

Verified Compositional Compilation without Concurrency. Compositional CompCert (CCC) is the pioneering work on VCC of imperative programs [23]. Its *interaction semantics* provide the simulation-based foundation for VCC which is adopted and enhanced by many projects (including this one). Its main limitation is that every language semantics and simulation must conform to the C interface, with which direct refinements and adequacy cannot be supported. CompCertM (CCM) fixes this problem by introducing *Refinement Under Self-related Contexts* or RUSC [22] which is the collection of all intermediate semantics and simulation relations in compilation. Although adequacy is recovered, a direct refinement is still not possible because the fixed collection exposes intermediate semantics. Instead, CompCertO (CCO) defines the transitive composition of all intermediate simulations as compiler correctness [12]. It is later shown that this transitive composition is equivalent to a direct refinement by exploiting the memory protection of Kripke Memory Relation [27]. Therefore, CompCertO for the first time supports all the important properties for verified compilation of heterogeneous modules and provides the basis for developing CompCertOC.

Verified Compositional Compilation with Concurrency. An early work on verified compilation of concurrent programs is CompCertTSO (CCTSO) [21] which is inherently a whole-program compiler

for multi-threaded programs. Although CCTSO supports stack sharing, it cannot support open modules and separate compilation due to the closed nature of its compiler correctness definition.

Thread-safe CompCertX (CCX) [5] extends CompCertX [3]—an extension of CompCert supporting mixed C and assembly programs—with the support of open threads. The compositionality supported by CCX is related to Concurrent Certified Abstraction Layers (CCAL) [4] which disallows mutual calls between modules (i.e., no callback function). Moreover, because CCX puts all memory blocks into a single space like the vanilla CompCert, it cannot support thread-local stacks. To avoid the complexity brought by this limitation, CCX forbids sharing of stack data. CompCertOC solves this problem by instantiating the nominal memory model [25] with thread-local stacks.

CASCompCert (CASCC) is the first extension of CompCert that supports general recursion and concurrency [8]. Since CASCC is built on top of Compositional CompCert, it inherits its limitations including forcing all languages into a C-like interface, resulting in loss of direct refinements and adequacy. CASCC supports thread-local stacks by dividing CompCert's memory state into stack and global memory space. However, it is unclear how to enable stack sharing while maintaining compositionality of refinements in CASCC. An appendix of 22 pages in the technical report of CASCC describes a solution on paper [9]. However, due to its complexity, this solution has not been formalized. By contrast, CompCertOC solves this problem by uniformly applying Threaded Kripke Memory Relations to all compiler passes while retaining full compositionality.

Relationship with other Concurrency Models. In this paper, we are concerned only with cooperative multi-threading with sequential consistent programs, and without explicit synchronization primitives like locks. However, those are not fundamental limitation as CASCompCert has shown that *1)* preemption in multi-threading can be pushed out of compiler correctness into source/target level verification with sequential consistency [8], and *2)* preemptive concurrent execution is equivalent to non-preemptive one where context switches only happen at synchronization points. We plan to add the supports of preemption and synchronization primitives by combining our compiler verification with program verification tools such as separation logics or certified concurrent abstraction layers.

We believe our ideas also apply to stronger concurrency models such as linearizability [6]. A more difficult problem is to deal with out-of-order execution incurred by relaxed memory models. For this, we need to develop new rely/guarantee simulation frameworks that allow both stack sharing and out-of-order execution, which is left for future work. Recent work on verified compiler optimizations for weak memory model based on promising semantics has shed light on the solutions [2, 11, 13, 26].

Conclusion. We presented CompCertOC, the first verified compiler that supports heterogeneous open modules and threads with stack sharing and full compositionality. The key enabling technique is the Threaded Kripke Memory Relations for simultaneously enforcing protection of private stack memory and enabling sharing of public stack memory. We demonstrated the effectiveness of CompCertOC by verifying non-trivial multi-threaded programs mixing C and assembly with stack sharing by using POSIX thread APIs. In the future, we would like to connect CompCertOC with program verification frameworks such as concurrent separation logics [1, 10, 17] to achieve end-to-end verification of concurrent programs.

References

[1] Andrew Appel. 2011. Verified Software Toolchain. In *Proc. 20th European Symposium on Programming (ESOP'11)*, Gilles Barthe (Ed.). LNCS, Vol. 6602. Springer, Saarbrucken, Germany, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

[2] Minki Cho, Sung-Hwan Lee, Dongjae Lee, Chung-Kil Hur, and Ori Lahav. 2022. Sequential reasoning for optimizing compilers under weak memory concurrency. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 213–228. https://doi.org/10.1145/3519939.3523718

[3] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan(Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*, Sriram K. Rajamani and David Walker (Eds.). ACM, New York, NY, USA, 595–608. https://doi.org/10.1145/2775051.2676975

[4] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, GA, 653–669.

[5] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjober, Hao Chen, David Costanzo, and Tahnia Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *Proc. 2018 ACM Conference on Programming Language Design and Implementation (PLDI'18)*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, New York, NY, USA, 646–661. https://doi.org/10.1145/3192366.3192381

[6] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

[7] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The Marriage of Bisimulations and Kripke Logical Relations. In *Proc. 39th ACM Symposium on Principles of Programming Languages (POPL'12)*, John Field and Michael Hicks (Eds.). ACM, New York, NY, USA, 59–72. https://doi.org/10.1145/2103656.2103666

[8] Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019. Towards Certified Separate Compilation for Concurrent Programs. In *Proc. 2019 ACM Conference on Programming Language Design and Implementation (PLDI'19)*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, New York, NY, USA, 111–125. https://doi.org/10.1145/3314221.3314595

[9] Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019. Towards Certified Separate Compilation for Concurrent Programs. https://plax-lab.github.io/publications/ccc/ccc-tr.pdf

[10] Ralf Jung, David Swasey, Filip Sieczkowski, Ksper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*. 637–650.

[11] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *Proc. 44th ACM Symposium on Principles of Programming Languages (POPL'17)*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, New York, NY, USA, 175–189. https://doi.org/10.1145/3009837.3009850

[12] Jérémie Koenig and Zhong Shao. 2021. CompCertO: Compiling Certified Open C Components. In *Proc. 2021 ACM Conference on Programming Language Design and Implementation (PLDI'21)*. ACM, New York, NY, USA, 1095–1109. https://doi.org/10.1145/3453483.3454097

[13] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. In *Proc. 2020 ACM Conference on Programming Language Design and Implementation (PLDI'20)*. ACM, New York, NY, USA, 362–376. https://doi.org/10.1145/3385412.3386010

[14] Xavier Leroy. 2005–2023. The CompCert Verified Compiler. https://compcert.org/.

[15] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2.* Research Report RR-7987. INRIA. 26 pages. https://hal.inria.fr/hal-00703441

[16] Rust Standard Library. 2024. std::thread::scope. https://doc.rust-lang.org/std/thread/fn.scope.html

[17] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and German Andres Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP'14*. 290–310.

[18] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: a Compositionally Verified Compiler for a Higher-Order Imperative Language. In *Proc. 2015 ACM SIGPLAN International Conference on Functional Programming (ICFP'15)*, Kathleen Fisher and John H. Reppy (Eds.). ACM, New York, NY, USA, 166–178. https://doi.org/10.1145/2784731.2784764

[19] Daniel Patterson and Amal Ahmed. 2019. The Next 700 Compiler Correctness Theorems (Functional Pearl). *Proc. ACM Program. Lang.* 3, ICFP, Article 85 (August 2019), 29 pages. https://doi.org/10.1145/3341689

[20] Andrew M. Pitts. 2016. Nominal Techniques. *ACM SIGLOG News* 3, 1 (2016), 57–72. https://doi.org/10.1145/2893582.2893594

[21] Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013), 22:1–22:50. https://doi.org/10.1145/2487241.2487248

[22] Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2020. CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.* 4, POPL, Article 23 (January 2020), 31 pages. https://doi.org/10.1145/3371091

[23] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*. ACM, New York, NY, USA, 275–287.

https://doi.org/10.1145/2676726.2676985

[24] Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. *Proc. ACM Program. Lang.* 3, POPL, Article 62 (January 2019), 30 pages. https://doi.org/10.1145/3290375

[25] Yuting Wang, Ling Zhang, Zhong Shao, and Jérémie Koenig. 2022. Verified Compilation of C Programs with a Nominal Memory Model. *Proc. ACM Program. Lang.* 6, POPL, Article 25 (January 2022), 31 pages. https://doi.org/10.1145/3498686

[26] Junpeng Zha, Hongjin Liang, and Xinyu Feng. 2022. Verifying Optimizations of Concurrent Programs in the Promising Semantics. In *Proc. 2021 ACM Conference on Programming Language Design and Implementation (PLDI'22)*. ACM, New York, NY, USA, 903–917. https://doi.org/10.1145/3519939.3523734

[27] Ling Zhang, Yuting Wang, Jinhua Wu, Jérémie Koenig, and Zhong Shao. 2024. Fully Composable and Adequate Verified Compilation with Direct Refinements between Open Modules. *Proc. ACM Program. Lang.* 8, POPL, Article 72 (Jan. 2024), 31 pages. https://doi.org/10.1145/3632914