

Towards Verified Shared-memory Cooperation for C

Lennart Beringer
Princeton University

Gordon Stewart
Princeton University

Robert Dockins
Portland State University

Andrew W. Appel
Princeton University

Abstract

We report on the ongoing design of a novel architecture for verified separate compilation of C programs, in the context of the CompCert certified C compiler.

1. Introduction

Shared-memory cooperation, or the coordinated use of memory by several static or dynamic execution units, occurs ubiquitously in modern systems software: *sequential* applications exchange pointers across the boundaries of modules; *concurrent* threads interact via memory synchronization and through the communication of pointers to shared data structures; nearly *all* programs communicate via memory with libraries and contain pointer-valued system calls to the underlying operating system. Correct compilers—those that preserve program safety and functional specifications—must be careful, accordingly, to respect a program’s effects on memory.

Yet modern optimizing compilers for system languages such as C routinely perform code transformations that *alter* memory behavior. Typical modifications include the relocation or elimination of load and store operations, the coalescing of allocation events, and the addition of almost all kinds of memory operations (including freeing) for the purpose of frame stack management. To take a concrete example, consider the CompCert certified C compiler [7]. In order to limit pointer aliasing and perform efficient register allocation, CompCert eliminates at a very early stage all memory accesses to local variables whose addresses are not taken. These variables, which are stored in a variable environment modeling the register bank, are in effect “removed from the memory” (though some of the variables may later be spilled back into memory after register allocation).

CompCert’s variable relocation and spilling phases are important for generating efficient code, yet these phases also complicate the compiler’s specification as it relates to memory. Correctness of variable relocation, spilling, and indeed of any other phase that may adjust the memory layout, must ensure that the memory behavior of a program is appropriately preserved by the transformation. However, it is not clear what “preservation of memory behavior” means when the compiler may itself introduce or remove memory effects! The difficulties are made even more acute when translation units may be *separately compiled*, since a pointer passed as an argument between modules may need to be translated depending on the compiler intermediate languages in which the modules are expressed.

Versions 1.0 (2006) to 1.13 (2013) of the CompCert compiler bypass these tricky issues by simply disallowing events containing pointers into the heap. Events in the CompCert model—which defines program equivalence at top level as equivalence of traces, proved by bisimulation—are external function calls, together with their arguments and return values, which must be integer or float scalars or pointers to statically initialized data. As a result, basic interactions with system calls such as Linux `sys_read` and `sys_write` cannot be validated because the calls take pointer ar-

guments, nor are programming abstractions such as lock-based concurrency supported in the current CompCert model. Crucially, CompCert’s current correctness proof provides no guarantees on the interactions of separately compiled modules.

In this talk, we report on ongoing work at Princeton to adapt CompCert to support shared-memory interaction—of separately compiled modules and of concurrent threads. This work takes place in the larger context of the Verified Software Toolchain project [1]. The primary goal of the talk is to give an overview of the main technical challenges that arise in the shared-memory setting, some of the most interesting of which are the following.

1) *Formulating a uniform operational model of shared-memory interaction.* Program modules that interact with their environments, either by transferring pointers in external calls to other modules or via concurrent synchronization, should do so in a way that hides local computation (on local state and local control) from computation on shared memory. We outline a uniform operational model that encompasses such interactions. Our model is *operational* because we wish to recast the semantics of CompCert’s intermediate languages (all of which are currently given by small-step operational semantics) in the new, shared-memory model with as little disruption to CompCert’s current proofs as possible. Our model is also *uniform* in the sense that it supports shared-memory interaction of multi-language programs (as are typical in, *e.g.*, operating systems) written in a mix of C, IA32 assembly, or any programming language that shares CompCert’s memory model.

2) *Recovering semantics preservation under compilation, in the presence of shared-memory effects.* In order to recover guarantees about the behaviors of compiled programs that interact via shared memory, it is necessary to adapt CompCert’s semantics preservation proofs—currently formulated as forward simulations to deterministic target languages—to the shared-memory setting. This adaptation involves a) a translation of CompCert’s simulation proofs to the shared-memory interaction semantics described above; and b) a proof that shared-memory simulations compose along the phases of the compiler. In line with some recent results on transitivity of related methods for proving program equivalence [3, 4, 6], transitivity in the presence of shared-memory interaction turns out to be surprisingly tricky.

3) *Identifying the mutual memory invariants necessary to support separate compilation.* CompCert’s correctness proof exhibits a simulation invariant for each compiler phase. For phases that reorganize memory (for example, to lay out concrete activation records or to perform spilling and reloading), these invariants typically require that certain portions of the memory, such as spill locations, remain unchanged over calls to external functions. When external functions are themselves compiled code—as in the separate compilation setting—the mutual invariants that govern the interactions among program modules grow increasingly complicated. We discuss the issues that arise, including the give-and-take between what a compiled module or thread may do to its own memory versus what it may do to the memory of other modules or threads. For ex-

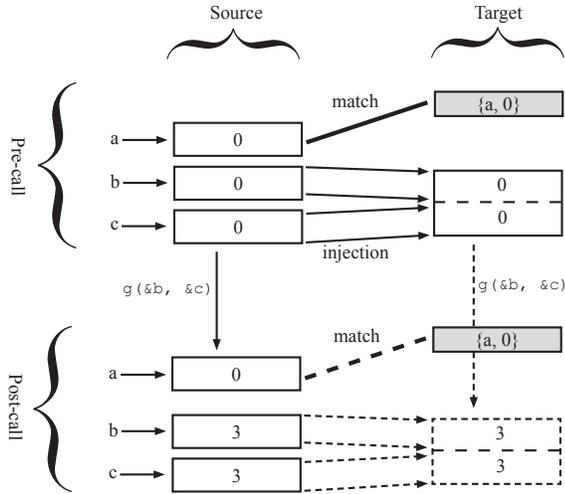


Figure 1. Schematic CompCert simulation diagram for f 's call to external function g . The block containing the unaddressed local variable a (in Source) is relocated out of memory into the local variable environment (gray, in Target). The blocks containing variables b and c are coalesced in the injection from Source to Target (in recent versions of CompCert, stack block coalescing is now a distinct phase).

ample, in order to preserve compiler invariants, one invocation of a module may write to a spill location in a stack frame it allocated, but may not overwrite a spilled register in, e.g., its caller's stack frame.

1.1 Related Work

Compiler correctness has a long and rich history. Some of the most appealing approaches were developed in the setting of (typed) lambda calculi [8, 9]. Indeed, proof techniques such as logical relations and TT-closure exploit type structure to provide an elegant treatment of code executing in context, with support for higher-order functions. Recently, significant progress has been achieved in extending these techniques to low-level code [4, 5, 6] and to the treatment of memory and other effects [2, 11]. However, little progress has been reported on transferring these techniques to the “wild west” of system languages such as C. For a model of concurrent threads running in the context of a weakly consistent TSO memory, CompCertTSO [10] provides one solution, but we are unaware of a comprehensive account of how the memory rearrangements performed by optimizing compilers interact with modular compilation in more general modes of interaction, in the absence of rich type structure.

2. A Taste of the Technical Challenges

CompCert models memory as a set of *blocks*, the sizes of which are fixed at allocation time. Addresses are pairs of a block and an *offset*, which is an integer pointing to a particular location within the block. Pointer arithmetic is allowed within blocks but not across blocks. To model local variable pointer arithmetic, CompCert allocates a fresh block for each local variable. For example, a stack-allocated `char` array of size n would be allocated as an n -byte block, whereas a local integer variable would be allocated as a block of size 4 (on a 32-bit architecture).

Because CompCert may alter a program's memory layout during compilation, the memory model must also support a number of *memory transformations*. For example, consider the following two C modules A and B.

```

//Module A
extern void
g(int *d, int *e);

int f(void) {
  int a = 0, b = 0, c = 0;
  g(&b, &c);
  return a + b + c;
}

//Module B
void g(int *d, int *e) {
  *d = 3;
  *e = 3;
}

```

Module A declares an external function g taking two integer pointers as parameters and returning `void`. It also defines an *internal* function f calling g with the addresses of f 's local variables b and c as arguments. Module B defines g to perform side effects on the pointers that are passed.

Now imagine CompCert is compiling the code in Module A through a phase that removes unaddressed local variables from memory, in order to allocate them in registers. In the program above, the unaddressed variable a in function f of Module A will be relocated out of memory. The schematic view of the memory state before and after this transformation, directly before the external function g is called, is shown in the top half of Figure 1.

The simulation invariant on pre-call states asserts that the value in memory at address $\&a$ —before variable relocation has occurred (Source)—equals the value of the variable a in the local variable environment after variable relocation (Target). In the diagram, we depict this constraint as a bold line labeled “match” connecting the two boxes.

In order to prove that relocating a out of memory is correct, CompCert must show that g will succeed in the target memory, assuming it succeeded in the source memory before the transformation. It also must re-establish the simulation relation on the post-call states that result from calling g . These two proof steps correspond to the completion of the lower-right half of the diagram in Figure 1 (Post-call, Target). In other words, we must find a state that is the transformation of the post-call source memory in the lower left, and is reachable via a transposition of the external call to the pre-call target state in the upper right.

The state in the lower right additionally needs to satisfy the “match” invariant with respect to the lower left state. But a problem arises here. In order to permit as many external functions as possible, we would like to allow external functions to make unconstrained updates to the memories they receive at call points. Yet the simulation invariant asserted by CompCert's variable relocation phase (and indeed, those of other phases of the compiler as well) makes assumptions about the memory it receives when the external call returns, with respect to the “private” local variable environment of f . Namely, the value in memory at address $\&a$ after the call should match the value assigned to a in the local variable environment.

One solution is to allow external functions simply to mutate memory everywhere, even at those addresses that appear free in simulation invariants. However, after each external call, simulation proofs would then be required to witness a new *local* state, in addition to a new memory state, that matches the external call's effects on memory. This is a clear violation of information hiding. In order to support the interoperation of modules defined in different programming languages, with different notions of local state and control, it is more convenient, and indeed, more in keeping with the principle of abstraction, if external functions do not modify those parts of memory in one compiler intermediate representation that are related by a simulation invariant to private local state in another representation.

References

- [1] A. W. Appel. Verified Software Toolchain. In *ESOP*, pages 1–17, 2011.
- [2] N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations: Higher-order store. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 301–312. ACM, 2009.
- [3] L. Birkedal and A. Bizjak. A note on the transitivity of step-indexed logical relations. 2012.
- [4] C. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *POPL*, pages 59–72. ACM, 2012.
- [5] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *ACM SIGPLAN Notices*, volume 46, pages 133–146. ACM, 2011.
- [6] C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. The transitive composability of relation transition systems. *MPI-SWS, Tech. Rep. MPI-SWS-2012-002*, 2012.
- [7] X. Leroy. The CompCert verified compiler, software and commented proof, Mar. 2011.
- [8] G. Plotkin. *Lambda-definability and logical relations*. School of Artificial Intelligence, University of Edinburgh, 1973.
- [9] J. Reynolds. On the relation between direct and continuation semantics. *Automata, languages and programming*, pages 141–156, 1974.
- [10] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. *ACM SIGPLAN Notices*, 46(1):43–54, 2011.
- [11] J. Thamsborg and L. Birkedal. A Kripke logical relation for effect-based program transformations. *ACM SIGPLAN Notices*, 46(9):445–456, 2011.