Chamois: agile development of CompCert extensions for optimization and security

David Monniaux, Sylvain Boulmé

Chamois: agile development of CompCert extensions for optimization and security

David Monniaux[1] and Sylvain Boulmé[1]

[1]Univ. Grenoble Alpes, CNRS, Grenoble INP

CompCert is the only formally-verified C compiler, and one of the very few formally verified compilers altogether. It is intended for use for safety-critical applications. This paper describes the improvements that we and associates have brought to CompCert: new VLIW target, new optimizations, and security features.

1 Introduction

Most compilers have no formal proof of correctness: their reliability is established by testing and by their track record for a certain kind of code on a certain platform. In industries such as avionics, it is required that features of the source code can be traced to the target code and the converse. One traditional approach then is to disable most optimizations, so that the source and assembly programs match closely, but this can cost a lot in performance. [Bed+11]

One solution is to use a *formally verified compiler*: a machine-checked proof of correctness states that if the compiler succeeds in compiling, then the semantics of the source and target codes match. There are few formally verified compilers: the only ones that we know of for general-purposes languages, are CompCert [Ler09b; Ler09a] for C and CakeML [Kum+14] for a dialect of ML. In this tool paper, we describe our extensions to CompCert.

CompCert's moderate optimization capabilities significantly improve upon disabling optimizations in a conventional compiler. However they are, as of 2023, still inferior to those of mainstream compilers such as gcc and LLVM. Furthermore, prior to 2023, CompCert was still missing support for desirable security features provided by mainstream compilers. The goal of Chamois[1] is to narrow the gap between CompCert and mainstream compilers.

2 Optimizations

We have improved CompCert in a variety of ways, which we shall discuss briefly.

2.1 Instruction selection

We added a full back-end for the Kalray KVX family of VLIW cores. We added support for certain instructions of certain processors, such as bitfield setting, clearing and extraction on ARM. In addition, we changed the operation sequences emitted for certain operations (such as signed division by two).

[1]`https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert`

2.2 Instruction scheduling

Many processors used in safety-critical or cheaper applications are *in-order*: they execute instructions in the order they occur in the program. If the operands of an instruction are not yet available when it executes, it and all instructions behind it in the pipeline *stall* until they become available. It is important that the compiler reorders instructions according to a *schedule* that minimizes stalling by taking into account instruction *latencies* (the clock cycles between when the instruction reads its operands and when it writes its results) as well as what combinations of instructions the processor can issue at the same clock cycle.

We added two scheduling passes. The first [Six21; Six+22] operates over the RTL intermediate representation, a form of "three-address code" that operates over an unbounded number of pseudo-registers, with memory accessed through "load" and "store" operations. It first divides the code into linear *superblocks*, each with one single entry point, a main exit and possibly some side exits. Superblocks are carved by heuristics that predict, at each branch point, the most common successor, or, often, conservatively opt not to identify one and terminate the superblock there. Furthermore, it is possible to use profiling information and user-provided annotations (`__builtin_expect()`, as in gcc and LLVM).

Each superblock is expressed in the BTL intermediate representation, a variant of RTL where control, instead of stepping through individual instructions, takes big steps across structured blocks, possibly containing conditionals, with one single entry point and possibly multiple exit points. Scheduling reorders instructions so that live outgoing values match between the original and reordered codes, and that no new trapping condition is introduced. For instance, the superblock `c=a/b; if (b>100) goto X;`, where `c` is not live at `X`, may be replaced by `if (b>100) goto X; c=a/b;`, but the converse replacement is valid only on machines where division by zero does not trap.[2]

Two simple *alias analyses* are used to allow swapping loads and stores over non-overlapping locations: one based on CompCert's existing value analysis, and another dealing with non-overlapping offsets from the same base address. In addition, a mechanism (currently being improved) avoids introducing too many live pseudo-registers, which may result in spilling.

A second scheduling pass [Six21; SBM20] is run after register allocation (KVX and AArch64 only). This pass can thus take into account loads and stores that have been added because of values spilled to the stack frame, as well as loads and stores induced by register being saved during procedure calls. A *peephole optimizer* [Gou21] replaces accesses to consecutive memory locations by "load/store pair/quadruplets of registers" instructions. On VLIW processors (KVX) this pass also forms *instruction bundles*.

In both passes, following Tristan [Tri09] and Tristan and Leroy [TL08], an untrusted transformation is followed by a formally verified checker that symbolically executes, in a term algebra, both the original and transformed programs, and checks equivalence. It uses hash-consing, nontrivial to model in a functional context [BJM14; Bou21].

2.3 Code restructuring

We perform *loop peeling* (unrolling of the first iteration of a loop[3] and *loop rotation*[4] as particular cases of "morphisms" [Gou+23]: each node in the transformed program corresponds to a node in the original program, and instructions are to be preserved between corresponding nodes. We also provide an opposite transformation, which merges bisimilar nodes.

[2] On certain processors, such as x86, division by zero results in a system trap which, by default, results in the program being terminated by the operating system. We modified CompCert to account for division by zero not trapping on other architectures, so that division operations can be rescheduled over branches or moved out of loops.

On most processors, loading from an incorrect memory location results in a "segmentation violation" trap, again terminating the program. We added support to CompCert for the "non trapping" flag of load operations on the Kalray KVX processor.

[3] This amounts to replacing `while (c) {b}` by `if(c) {{b} while(c) {b}}`, but on unstructured code.

[4] Replacing `while(c) {b}` by `if(c) {do {b} while(c)}`

2.4 Global common subexpression elimination

Common subexpression elimination identifies that some expressions at different positions in the program are identical and thus that one can avoid recomputing their values and instead reuse previously computed values. CompCert featured a *local* common subexpression elimination, which would not propagate information across control-flow joins.

Our new pass [MS22] identifies when the result of evaluating an operation, or a load from memory, is known to always lie in a certain pseudo-register at a given control location. Then, if that control location computes the same operation, it can be replaced by a "move" from that pseudo-register. In the simplest case, any "store" to memory cancels all known relationships involving "loads", but a simple alias analysis, dealing with accesses relative from the same base addressing, allows cancelling only those involving locations that cannot be proved not to overlap with the store. This pass was extended to remove redundant branches when their condition is known to always hold. Combined with loop peeling, this pass performs a form of *loop-invariant code motion*.

2.5 Lazy code motion and strength reduction

A second form a loop-invariant code motion moves loop-invariant expressions out of loops without need for unrolling (unless the expression may trap).This pass [Gou+23; Gou23] transforms BTL code, then runs a formally verified checker, which again performs symbolic execution, but initializes pseudo-registers according to invariants provided by the transformation pass. The checker also verifies these invariants hold inductively.

Lazy code motion was extended to *strength reduction*: replacing costly operations within loops by cheaper versions. Typically, an expression used within a loop contains a multiplication of an index by a constant (say, for computing the address of data in an array), with the index being incremented by a constant at every iteration, in which case it can be replaced by a computation of an initial value out of the loop and incrementing by a constant. That is, if i is initialized with n and then incremented at every loop iteration, $t + 4i$ can be computed by initializing it with $t + 4n$ and then incrementing it by 4 at every iteration. The formally verified checker was modified to rewrite strength-reduced expressions, so that $t + 4(i + 1)$ is considered equivalent to $(t + 4i) + 4$.

2.6 Tail recursion optimization

CompCert performs *tail call elimination*: tail calls to functions, which would normally retain the current stack frame, are replaced by the destruction of the current stack frame followed by a jump to the head of the function. If applied to a recursive function, this optimization creates a loop that destructs and recreates the stack frame at every iteration. We improve upon this by bypassing stack frame destruction and creation [Mon24].

3 Security features

We have added features that block certain software attacks and proved they do not disturb normal executions. In the future, we will also consider hardware attacks, and proving that countermeasures block certain attacks.

3.1 Branch target identification

On some platforms (x86, AArch64), Chamois can optionally add instructions that specify locations that are legal targets for a jump or function call. This reduces opportunities for "return address programming" and other techniques used to exploit software vulnerabilities.

3.2 Return address authentication

AArch64 provides *pointer authentication*. [App21; App19; Aza19; Tec17] Special instructions add some authentication bits to pointers, computed using keys that the operating system sets up in special CPU registers. Before the pointer is used, these bits are checked and removed; an invalid pointer is produced if they are incorrect. This prevents easy exploitation of vulnerabilities such as buffer overflows: the intruder cannot predict the keys and thus the value of the authentication bits to supply. If vulnerable software reads an intruder-supplied pointer from memory and de-authenticates it before using it to access memory, then the access traps.

Applying this approach to pointers in general needs some language-based mechanism for tagging which pointers are authenticated (with which key) and which are not; we do not have that mechanism, which would entail modifying CompCert's front-end and most likely adding a new "tagged pointer" type. We currently just authenticate return addresses [Mon24].

3.3 Canaries

A well-known attack method is to overflow a buffer allocated on the stack to overwrite the return address of a function. A *canary*[5] is a piece of data wedged between the local variables and the return address. Buffer overflows most often overwrite consecutive pieces of memory and thus overwrite the canary. Before restoring the return address and exiting the function, the value of the canary is checked, and the program aborts if it does not match. The canary value is randomized at program start and may not be predicted by the attack. We implemented this security feature, which had been standard for years in gcc and LLVM, and proved that it does not perturb legal executions [Mon24]. We discovered using these canaries that some benchmark programs that we were using exhibited undefined behavior (e.g. due to assuming a 32-bit platform).

4 Conclusion

Various optimizations and security features have been implemented, and more are on the way. Many of these optimizations are composed of an untrusted transformation followed by a formally verified checker. It is necessary to test, on representative and also "trick" examples, if it happens that the checker does not accept the results [Mon+23]. The main difficulty so far has been to identify what optimizations would be interesting and what causes performance discrepancies. The other difficulty is that some possible improvements would entail deep changes in the semantics of intermediate representations. It is often difficult to predict, from the outside, the degree of changes necessary.

Acknowledgements

Xavier Leroy kindly improved the abstract domain used in the value analysis. This results in more precise alias information, which helps several of our optimizations.

References

[App19] Apple. *Pointer Authentication*. Documentation for Apple's fork of LLVM. 2019. URL: `https://github.com/apple/llvm-project/blob/apple/main/clang/docs/PointerAuthentication.rst`.

[5] The name comes from canaries formerly used in mines to warn miners of carbon monoxide poisoning by dying before the miners.

[App21] Apple. *ARMv8.3 Pointer Authentication in xnu*. 2021. URL: https://opensource.apple.com/source/xnu/xnu-7195.50.7.100.1/doc/pac.md.

[Aza19] Brandon Azad. *Examining Pointer Authentication on the iPhone XS*. Google Project Zero. 2019. URL: https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html.

[Bed+11] Ricardo Bedin França et al. "Towards Optimizing Certified Compilation in Flight Control Software". In: *Workshop on Predictability and Performance in Embedded Systems (PPES 2011)*. Vol. 18. OpenAccess Series in Informatics. Grenoble,France: Dagstuhl Publishing, 2011, pp. 59–68. URL: http://hal.archives-ouvertes.fr/inria-00551370/.

[BJM14] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. "Implementing and Reasoning About Hash-consed Data Structures in Coq". In: *Journal of Automated Reasoning* (June 2014), pp. 1–34. ISSN: 0168-7433. DOI: 10.1007/s10817-014-9306-0. HAL: hal-00816672. URL: https://hal.archives-ouvertes.fr/hal-00816672.

[Bou21] Sylvain Boulmé. "Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles)". See also http://www-verimag.imag.fr/ boulme/hdr.html. Habilitation à diriger des recherches. Université Grenoble-Alpes, Sept. 2021. URL: https://hal.science/tel-03356701.

[Gou+23] Leo Gourdin et al. "Formally Verifying Optimizations with Block Simulations". In: *OOPSLA*. To appear. Oct. 2023.

[Gou21] Léo Gourdin. "formally verified postpass scheduling with peephole optimization for AArch64". In: *AFADL*. 2021. URL: https://www.lirmm.fr/afadl2021/papers/afadl2021_paper_9.pdf.

[Gou23] Léo Gourdin. "Lazy Code Transformations in a Formally Verified Compiler". In: *Proceedings of the 18th ACM International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems, ICOOOLPS 2023, Seattle, WA, USA, 17 July 2023*. Ed. by Eric Jul and Dimi Racordon. ACM, 2023, pp. 3–14. DOI: 10.1145/3605158.3605848.

[Kum+14] Ramana Kumar et al. "CakeML: A Verified Implementation of ML". In: *Principles of Programming Languages (POPL)*. ACM Press, Jan. 2014, pp. 179–191. DOI: 10.1145/2535838.2535841. URL: https://cakeml.org/popl14.pdf.

[Ler09a] Xavier Leroy. "A Formally Verified Compiler Back-end". In: *J. Autom. Reason.* 43.4 (2009), pp. 363–446. DOI: 10.1007/s10817-009-9155-4.

[Ler09b] Xavier Leroy. "Formal verification of a realistic compiler". In: *Commun. ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814.

[Mon+23] David Monniaux et al. "Testing a Formally Verified Compiler". In: *Tests and Proofs - 17th International Conference, TAP 2023, Leicester, UK, July 18-19, 2023, Proceedings*. Ed. by Virgile Prevosto and Cristina Seceleanu. Vol. 14066. Lecture Notes in Computer Science. Springer, 2023, pp. 40–48. DOI: 10.1007/978-3-031-38828-6_3.

[Mon24] David Monniaux. "Memory Simulations, Security and Optimization in a Verified Compiler". In: *Certified Programs and Proofs 2024*. Ed. by Brigitte Pientka and Sandrine Blazy. Brigitte Pientka and Sandrine Blazy and Amin Timany and Dmitriy Traytel. London, United Kingdom: Association for computing machinery (ACM), Jan. 2024. DOI: 10.1145/3636501.3636952. HAL: hal-04336347.

[MS22] David Monniaux and Cyril Six. "Formally Verified Loop-Invariant Code Motion and Assorted Optimizations". In: *ACM Trans. Embed. Comput. Syst.* 22.1 (Dec. 2022). ISSN: 1539-9087. DOI: 10.1145/3529507.

[SBM20] Cyril Six, Sylvain Boulmé, and David Monniaux. "Certified and efficient instruction scheduling: application to interlocked VLIW processors". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 129:1–129:29. DOI: 10.1145/3428197. HAL: hal-02185883.

[Six+22] Cyril Six et al. "Formally verified superblock scheduling". In: *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*. Ed. by Andrei Popescu and Steve Zdancewic. ACM, 2022, pp. 40–54. DOI: 10.1145/3497775.3503679. URL: https://doi.org/10.1145/3497775.3503679.

[Six21] Cyril Six. "Compilation optimisante et formellement prouvée pour un processeur VLIW". PhD thesis. Grenoble Alpes University, France, 2021. URL: https://tel.archives-ouvertes.fr/tel-03326923.

[Tec17] Qualcomm Technologies. *Pointer authentication on ARMv8.3. Design and Analysis of the New Software Security Instructions*. 2017. URL: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf.

[TL08] Jean-Baptiste Tristan and Xavier Leroy. "Formal verification of translation validators: a case study on instruction scheduling optimizations". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. Ed. by George C. Necula and Philip Wadler. ACM, 2008, pp. 17–27. DOI: 10.1145/1328438.1328444. URL: https://doi.org/10.1145/1328438.1328444.

[Tri09] Jean-Baptiste Tristan. "Formal verification of translation validators". PhD thesis. Paris Diderot University, France, 2009. URL: https://tel.archives-ouvertes.fr/tel-00437582.