Compiler Optimization Verification and Maintenance

Abstract

Due to the complexity of a compiler, it is difficult for compiler developers to prove the correctness of output of intermediate representations or profiling information manually after applying optimizations. Moreover, it is important to choose good test cases to test the optimization code's changes, however, it is not simple to build separated test cases to cover every possible optimizations. This proposal proposes automatic techniques to prove the correctness of an IR, the correctness of its annotations for correct profiling information, and to generate test cases according to the kinds of optimizations. Moreover, the test case size can be reduced by using bug isolation techniques. We focus our techniques on loop nest optimizations in the GNU C compiler with SPEC-CPU2006 benchmark test suites.

1. Introduction

Applying the proper compiler optimizations to improve application's performance is as important as writing good code. For compiler users, the most important reason to use compiler optimizations is to improve performance under certain metrics. Thus, a compiler writer's key concern is to write efficient and reliable optimizations that produce a more efficient program without producing incorrect results. Particularly, compiler writers should insure that each phase which has new optimizations that produce legal intermediate representations and it does not affect other phases or the correctness of the output of compilation. However, a compiler is one of the most complex software frameworks, thus it is neither easy nor simple to prove the correctness of the compiler manually. Alongside the intermediate representations between two phases of the compiler, there are some annotations which are visible to each phase, but invisible to compiler writers. Moreover, it is not easy to choose proper test cases reflecting the characteristics of each optimization to help compiler developers to insure the correctness of their changes effectively and accurately.

The major goal of this proposal is to build an optimization development maintenance tool that helps compiler developers by:

1. checking whether each optimization phase produces a legal intermediate representation. Thus it will not affect to other phases and, it will produce the correct output,
2. checking whether annotations between phases are correct and produce the correct profiling information for developers, and
3. generating a small but effective test case from real applications to test optimization that developers have just applied.

2. Background

This section explains the IR, the optimization phase, and the bug isolation technique targeted by this proposal.

2.1. Intermediate Representation

An IR(Intermediate Representation) is an internal form to represent the code being analyzed and translated by a compiler [13]. In most cases, optimizing a program means that we optimize and transform the IR of a code instead of the source code itself. Thus, it is important to check the correctness of the IR during such optimizations. Different compilers use different formats of IR such as graphical IR, linear IR, or hybrid IR using both graphical and textual information.

The SSA(Static Single Assignment) form is one of the hybrid IRs [13]. In this form, we can have only one single definition for the each variable uses. This simplifies analysis for the compiler and eventually it helps compiler writers to apply optimizations. Nowadays, several open source compilers are using SSA form as an IR, such as, the SGI compiler [3], the JikesRVM [2], the GCC compilers version 4.0 and above [1], and so on.

2.2. Loop Nest Optimization Phase

Loop Nest Optimization (LNO) performs transformations on a loop nest for optimizations [5]. This phase does not build any control flow graph and optimizations are driven by data dependency analysis. LNO includes loop unrolling, loop fusion, loop fision, loop interchange, and so on.

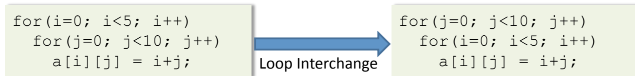

Figure 1. Example of Loop Interchange

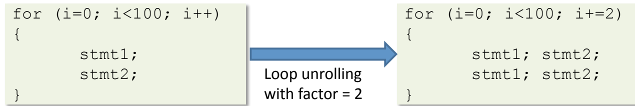

Figure 2. Example of Loop Unrolling with Factor 2

For example, it is good to exchange the outer loop and the inner loop like Figure 1 by using loop interchange for better locality. Another example is to use unroll loops to make loop bodies bigger to improve the performance like Figure 2. This proposal only focuses on this phase to prove the correctness of the output IR and annotations.

2.3. Automated Bug Finder

This technique is introduced in LLVM (Low Level Virtual Machine) used to reduce test case size by throwing out portions of a program which are checked to be correct. This will locate the portions of the program that might cause the crash and simplifies the debugging process[4]. In a similar way, since we only focus on loop optimization in this proposal, we find the proper loop body affected by such optimizations and generate test cases.

3. Related Work

Figure 3 shows the diagram of the state of the art in compiler verification and validation. There are two approaches - static and dynamic approaches. Static approaches do not involve any runtime information. In the beginning of compiler verification research, many people used static approaches, only focused on specific features of the code and proved the correctness manually. For example, McCarthy et al. [37] is considered to be the first work in compiler verification and they used static approaches that only focused on arithmetic expression.

Dynamic approaches involve runtime information from either the compiler or external tools. These approaches have been developed by modifying a compiler infrastructure or using translation validation.

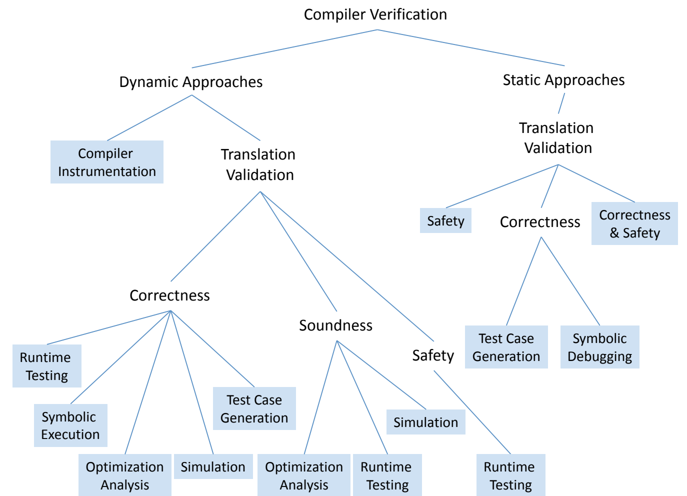

Figure 3. Diagram of the State of the Art in Compiler Verification and Validation

Translation validation is one of methods that is used to test the correctness of a compiler both in static and dynamic methods, but more frequently in dynamic approaches. It does not require any compiler modification. Without compiler instrumentation, independent tools are used to analyze the compiler's results and check the correctness of results. There are two projects related to translation validation - the verifix projects and the TVOC projects. The Verifix projects have proposed techniques and formalisms for compiler result checkers, decomposition of compilers and notions of semantical equivalence of source and target programs [10]. The TVOC projects are built on translation validation. It proves the correctness of compiler components instead of the entire compiler. The input of their tool is an intermediate representation in which several optimizations have been applied and the output shows whether applied optimizations are valid or not.

3.1. Static Approaches

Static approaches have been used to prove the safety, correctness, and both. Xu et al. [49] determines statically whether it is safe for untrusted machine code to be loaded into a host system. Denney et al. [16] demonstrated a framework to prove the safety property and the correctness of programs statically.

There are several approaches to prove correctness of programs statically using translation validation. Benton [8] interpreted program properties as relations, rather than predicates. They used static analyses for imperative programs and they expressed and proved the correctness of optimiz-

ing transformation using elementary logical and denotation techniques. Cousot et al. [14] describe how they prove the correctness by using source-to-source transformation with abstract interpretation.

Some people used symbolic debugging information to prove the correctness of a program statically. McNerney [38] implements a specialized program equivalence prover using abstract interpretation within functional blocks instead of tree comparison to validate the register allocation of compiler. Hennessy [23] checked the correctness of optimization by using symbolic debugging. Coutant et al. [15] developed DOC, a prototype solution for Debugging Optimized Code. DOC is a modified C compiler and source level symbolic debugger. Brooks et al. [11] introduced new compiler-debugger interfaces to provide visual feedback and debugging of optimized code.

3.2. Dynamic Approaches

Most research for compiler validation and verification are dynamic approaches. We can classify these into two main approaches - one is using compiler instrumentation, another is using translation validation without any compiler modification.

3.2.1. Compiler Instrumentation: Rinard[46]'s work is based on an instrumented compiler, producing output from input and checking the correctness by using a proof-checker. Another work from Rinard [47] is also based on instrumented compilers. They introduce the formal foundation and architectural design of a credible compiler. They present two logics, one for sub proofs of correctness and another for how the logics are sound. They manually proved optimizations are correct.

3.2.2. Translation Validation:

- Correctness Proof: The most closely related approach to our proposed method is from Zimmermann [52]. They summarize the results on the correctness of the transformations in compiler back-ends achieved in the Verifix projects. They focus on the correctness of intermediate representation transformation rules and the correctness of the whole transformation stemming from the transformation rules. Their proof strategies for term-rewrite rules work for local optimizations perfectly. However, they need different transformation rules such as local graph transformation rules for global optimizations such as code motion, or instruction scheduling. These parts need more work to work more properly but this part is future research. Another close approach to ours is Necula [41]'s approach that used symbolic execution and introduces translation validation based on symbolic execution for the GNU C compiler. They manually compared the intermediate representation of programs between each pass and prove the correctness of semantics. This work is done manually, thus it has the limitation to apply to bigger applications. We also focus on the correctness of intermediate representation, but our approach uses a different approach to prove the correctness. Instead of focusing on rules, we focus on real instance of intermediate representation and prove the correctness of them automatically. Our approach also proves the correctness of annotations for the correct profiling information which helps compiler developers in the debugging process.

  There are several approaches to prove compiler correctness by using semantic equivalence [24][50], different approaches in translation validation [19][20][22], prove carrying code techniques [40], mechanical verification [28][17], stage compilation frameworks [43], and certificate proof or certifying compilation [21][33][10]. Some approaches used formal proof of the correctness of specific compiler optimizations or issue specific compiler problems such as software pipelining optimizations[34], resource usage improvements[7], and targeting to reactive procedural programs[45].

  Most approaches listed above involve runtime execution. However, some research used only optimization analysis to prove the correctness of compilers[27][31][36]. Some research used simulation of compiler correctness proof [48][9][51].

  Some approaches proved the correctness of a compiler manually [44][29].

- Soundness Proof: There are several research projects focusing on a soundness proof of compiler by using stack-based control abstractions [18], implementing soundness checkers for specific language[32], using simulation[25], and using semantic equivalence checks with temporal logic[26].

- Safety Proof: Necula et al.[42] guarantee certain requirements or properties of a compiler program, such as type safety or the absence of stack overflows or memory safety. Colby et al. [12] applied proof carrying code and a certifying compiler to Java for type safety checking. Amme et al. [6] presented a mobile code representation based on static single assignment (SSA) to allow more optimizations, and check type safety better than using Java byte code and virtual machine. Menon et al.[39] presented a verifiable intermediate representation to embed, propagate, and preserve safety information in high performance compilers for safe languages. For intermediate representations en-

coding safety information, they use the SSA intermediate representation.

4. Challenges and Goals

This section presents challenges presented in the state of the art of the compiler validation fields.

4.1. Challenges

There are three main open issues in this area:

4.1.1. The correctness of IRs: It is important to check the correctness of IRs since most optimizations will be applied to them by modifying them before code generation. If we do not insure their correctness, they can cause incorrect results, compilation crashes, runtime errors, and so on. Moreover, it is difficult to find which IR we did not correctly generate and which caused all the problems. There are several state of the art approaches to prove IR correctness, but they are all done manually by using techniques such as symbolic execution. Even though we do not use any additional tool, in some compilers such as, the Open64 compiler, we can use debug mode to find which phase caused the problem. However, this does not give the exact point of the phase where we made the incorrect implementation.

4.1.2. The correctness of annotations: Besides IR correctness, we need to insure whether annotations from each phase are correct. Annotations are important to generate the correct profiling information, and this helps compiler developers in the debugging process. For example, for heuristic decision in a compiler, compiler developers often use branch probability information and this comes from annotations after a specific optimization phase. The sum of all branch probabilities from one basic block should be 1. If we do not insure this and simply trust the profiling information, compiler developers can make an incorrect decision from this incorrect profiling information. Thus, it is important to check whether annotations are correct or not, but there are no approaches to prove annotations besides IR correctness.

4.1.3. Test case size: It takes time and effort to build the practical test cases that represent the characteristics of optimizations we applied. If the test case size is too big, we might have more chance to have portions of the program to apply optimizations we want to test. However, the size of IRs also increases and sometimes IRs have completely different forms from original source code. Thus, it is hard to find the exact portion of IRs corresponding to the portion of programs. It is important to keep test size small enough but have the key portion of programs to use for certain optimizations. There are several researches on this such as test case minimizations or bug isolations [35][30]. However, they cannot be applied to this research directly. For example, bug isolation is used to reduce test case size by finding the portion of programs that cause crashes. If it does not crash, the test case size minimization will fail. Optimizations do not cause any crash usually, but it will cause different problems like performance degradation.

The common problem in each issue is that we do not have any automated processes and it requires human intervention.

4.2. Goals

We have three goals to resolve current issues in this research.

1. We prove the correctness of IRs automatically, and give more accurate information to compiler developers.
2. We prove the correctness of annotations from each phase automatically, and insure the profiling information for compiler developers is always reliable.
3. We generate small test cases but containing key characteristics of optimizations we want to test automatically.

5. Proposed Research

This section presents new proposed methodologies for this research. Each section describes the detail of our methodologies, all performed automatically.

5.1. Test Case Minimization

We follow the same approach as an automated bug finder techniques from LLVM [4]. They minimize programs until they find the portion of programs that caused the crash. We minimize programs until we find the portion of programs that needs to be optimized as shown in Figure 5. After we recognize this portion, we generate complete small programs that can be run without any other parts in the original programs.

5.2. IR Correctness Proof

We extract the key characteristics of the IR, thus we only prove the correctness for parts that cover those key characteristics. For example, when we test loop nest optimization,

1. We first extract portions of the IR corresponding to any loop bodies in the original source code.
2. For non-loop bodies in IR, we check if it is changed after applying optimization.

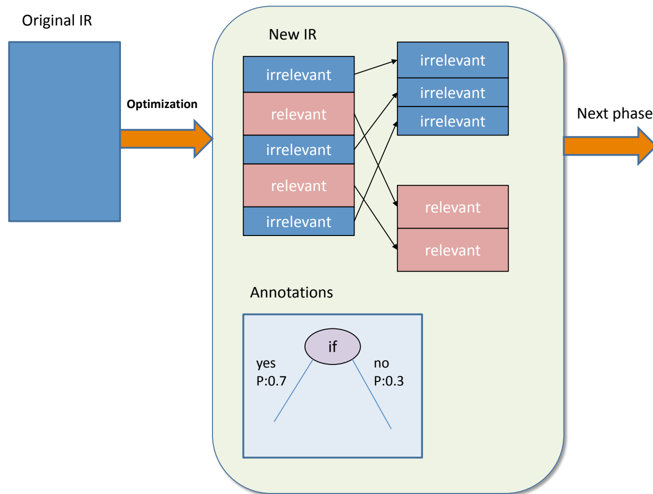

Figure 4. IR and Annotation correctness Proof

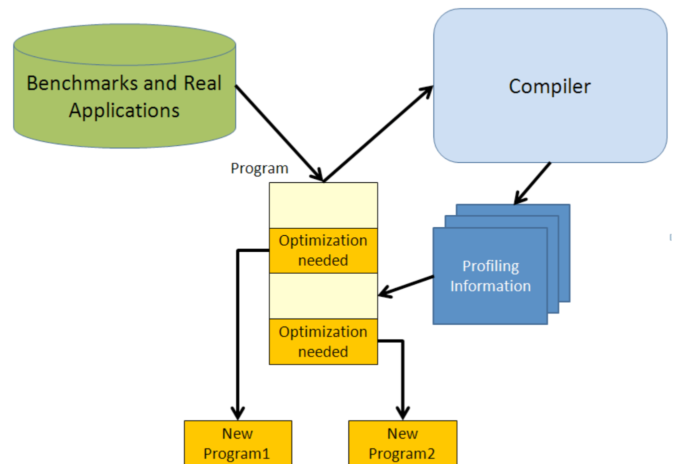

Figure 5. IR and Annotation correctness Proof

3. For loop bodies, we check if optimizations are applied correctly. For example, for unrolling factor, check if it was unrolled in specified number correctly.

If we want to test copy propagation, we need to use more information such as def-use chains. We will first extract portions of the IR based on def-use chain analysis and see if this portion needs to be optimized with copy propagation. Then we follow the same procedure.

Figure 4 shows how this procedure works. We first decide which properties of IR would present the key characteristics and we check whether parts that were supposed to change have been changed correctly. In the same way, we check whether some parts that are not supposed to change have changed and how it affects the IR correctness. To compare this, we will use a reference compiler which has the correct implementation of optimizations. With test cases we build from Section 5.1, we generate IR from the reference compiler and the modified compiler and generate source code of them. In this way, we can compare source code from each IR and check if test cases have been changed correctly according to kinds of optimization. If we are testing new optimization never implemented before, we need to build hand-coded optimization results to prove IR correctness.

5.3. Annotation Correctness Proof

We access to annotations after each phase and check if they are correct. For example, in Figure 4, if we want to check that branch probability information is correct, we calculate if the sum of all possible branches from one basic block is 1 or not. In the similar way, we check for all other annotations from each phase and make sure their value is correct or not.

6. Evaluation Plan

This section describes how we will evaluate our proposed methodologies. From this evaluation, we want to answer to those questions:

1. Can our methodology detect the incorrect implementation of compiler optimization accurately?
2. Can our methodology detect the incorrect annotations from each IR?
3. Can our test case minimization techniques successfully reduce test case sizes?

First, we choose optimizations we want to test by changing the current implementation in the GNU C compiler. We change the current correct implementation into incorrect version. For example, if unrolling factor was n, we unroll less than n times and see if our proposed methodology can recognize this. This process only generates fault seeds for our evaluation.

Second, we generate test cases by using test case minimization techniques with profiling information from the correct version of compiler. This process will evaluate our test case minimization techniques.

Third, we measure whether our methodology can successfully recognize the incorrect implementation from IR and annotation correctness proof. This process will evalute our IR and annotation correctness proof.

Fourth, we compare our methodology to the most closely related approach from Necula [41]'s approach based on symbolic execution. This process will evaluate how our methodology can detect incorrect implementation effectively.

7. Summary of Foreseen Contributions

By using proposed methodologies, compiler developers can insure IRs after each phase are legal to pass to the next phase. Moreover, they can find the exact point where they made incorrect implementation. They can use profiling information without worrying about its correctness by insuring the correctness of annotations from each phase. Also, they can minimize test case sizes by using new techniques based on bug isolation. The most important contribution is all of these proposed methodologies are done automatically with minimal human intervention.

References

[1] GCC, the GNU Compiler Colloection. http://gcc.gnu.org/.

[2] JikesRVM (Research Virtual Machine). http://jikesrvm.org/.

[3] SGI Pro64 Compiler. http://oss.sgi.com/projects/Pro64/.

[4] The LLVM Compiler Infrastructure. http://llvm.org/.

[5] The PathScale Ekopath Compiler Suite - User Guide.

[6] W. Amme, N. Dalton, J. von Ronne, and M. Franz. Safetsa: a type safe and referentially secure mobile-code representation based on static single assignment form. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 137–147, New York, NY, USA, 2001. ACM.

[7] D. Aspinall, L. Beringer, and A. Momigliano. Optimisation validation. *Electronic Notes in Theoretical Computer Science*, 176(3):37–59, 2007.

[8] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 14–25, 2004.

[9] J. O. Blech, L. Gesellensetter, and S. Glesner. Formal verification of dead code elimination in isabelle/hol. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 200–209, Washington, DC, USA, 2005. IEEE Computer Society.

[10] J. O. Blech and A. Poetzsch-Heffter. A certifying code generation phase. In *Proceedings of the International Workshop on Compiler Optimization Meets Compiler Verification (COCV)*, 2007.

[11] G. Brooks, G. J. Hansen, and S. Simmons. A new approach to debugging optimized code. *SIGPLAN Not.*, 27(7):1–11, 1992.

[12] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 95–107, New York, NY, USA, 2000. ACM.

[13] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[14] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 178–190, New York, NY, USA, 2002. ACM.

[15] D. S. Coutant, S. Meloy, and M. Ruscetta. Doc: a practical approach to source-level debugging of globally optimized code. *SIGPLAN Not.*, 23(7):125–134, 1988.

[16] E. Denney and B. Fischer. Correctness of source-level safety policies. *Lecture Notes in Computer Science*, 2805:894–913, 2003.

[17] A. Dold and V. Vialard. A mechanically verified compiling specification for a lisp compiler. In *FST TCS '01: Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 144–155, London, UK, 2001. Springer-Verlag.

[18] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 401–414, 2006.

[19] T. Gaul, A. Heberle, W. Zimmermann, and W. Goerigk. Construction of verified software systems with program-checking: An application to compiler back-ends. In *Proceedings of the Workshop on Run-Time Result Verification (RTRV)*, 1999.

[20] T. Gaul and W. Zimmermann. Practical construction of correct compiler implementations by runtime result verification. In *Proceedings of SCI*, 2000.

[21] S. Glensner. Using program checking to eusure the correctness of compiler implementations. *Journal of Universal Computer Science*, 9(3):191–222, 2003.

[22] W. Goerigk, A. Dold, G. Goos, A. Heberle, F. W. V. Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler correctness and implementation verification: The verifix approach. In *Proceedings of the International Conference on Compiler Construction (poster session)*, 1996.

[23] J. Hennessy. Symbolic debugging of optimized code. *ACM Trans. Program. Lang. Syst.*, 4(3):323–344, 1982.

[24] C. Jaramillo, R. Gupta, and M. L. Soffa. Comparison checking: An approach to avoid debugging of optimized code. In *Proceedings of the Conference on European Software Engineering (ESEC) and Symposium on the Foundations of Software Engineering (FSE)*, pages 268–284, 1999.

[25] A. Kanade, A. Sanyal, and U. Khedker. A pvs based framework for validating compiler optimizations. In *Proceedings of the International Conference on Software Engineering and Formal Methods (SEFM)*, pages 108–117, Washington, DC, USA, 2006. IEEE Computer Society.

[26] A. Kanade, A. Sanyal, and U. Khedker. Structuring optimizing transformations and proving them sound. *Electronic Notes in Theoretical Computer Science*, 176(3):79–95, 2007.

[27] D. Kozen and M.-C. Patron. Certification of compiler optimizations using kleene algebra with tests. In *CL '00: Proceedings of the First International Conference on Computational Logic*, pages 568–582, London, UK, 2000. Springer-Verlag.

[28] S. S. Kulkarni, B. Bonakdarpour, and A. Ebnenasir. Mechanical verification of automatic synthesis of fault-tolerant programs. *Lecture Notes in Computer Science*, 3573:36–52, 2005.

[29] D. Lacey, N. D. Jones, E. V. Wyk, and C. C. Frederiksen. Compiler optimization correctness by temporal logic. *Journal of Higher Order and Symbolic Computation*, 17(3):173–206, 2004.

[30] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 417–420, New York, NY, USA, 2007. ACM.

[31] S. Lerner, T. D. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 220–231, 2003.

[32] S. Lerner, T. D. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 364–377, 2005.

[33] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 42–54, 2006.

[34] R. Leviathan and A. Pnueli. Validating software pipelining optimizations. In *Proceedings of the International conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 280–287, New York, NY, USA, 2002. ACM.

[35] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26, New York, NY, USA, 2005. ACM.

[36] Y. Matsuno and H. Sato. A type system for optimization verifying compilers. *Information and Media Technologies*, 1(2):695–711, 2006.

[37] J. Mccarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *In Proceedings of Symposia in Applied Mathematics. AMS*, pages 33–41. American Mathematical Society, 1967.

[38] T. S. McNerney. Verifying the correctness of compiler transformations on basic blocks using abstract interpretation. In *Proceedings of the symposium on Partial Evaluation and semantics-based Program Manipulation (PEPM)*, pages 106–115, New York, NY, USA, 1991. ACM.

[39] V. Menon, N. Glew, B. R. Murphy, A. McCreight, T. Shpeisman, A.-R. Adl-Tabatabai, and L. Petersen. A verifiable ssa program representation for aggressive compiler optimization. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 397–408, 2006.

[40] G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, April 1997.

[41] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 83–94, 2000.

[42] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. *SIGPLAN Not.*, 33(5):333–344, 1998.

[43] M. Philipose, C. Chambers, and S. J. Eggers. Towards automatic construction of staged compilers. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 113–125, 2002.

[44] A. Pnueli, M. Siegel, and E. singerman. Translation validation. In *Proceedings of the TACAS*, 1998.

[45] A. Pnueli and G. Zaks. Validation of interprocedural optimizations. In *Proceedings of the International Workshop on Compiler Optimization Meets Compiler Verification (COCV)*, 2008.

[46] M. Rinard. Credible compilation. Technical report, 1999.

[47] M. C. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the Workshop on Run-Time Result Verification*, 1999.

[48] M. Wand. Compiler correctness for parallel languages. In *Funcional Programming Languages and Computer Architecture (FPCA)*, pages 120–134, 1995.

[49] Z. Xu, B. P. Miller, and T. W. Reps. Safety checking of machine code. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 70–82, 2000.

[50] A. Zaks and A. Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *Proceedings of the World Congress on Formal Methods (FM)*, pages 35–51, 2008.

[51] S. V. Zelenov and S. A. Zelenova. Model-based testing of optimizing compilers. In *Proceedings of the International Conference on Testing on Communicating Systems (TESTCOM) and the International Workshop on Formal Approaches to Testing of Software(FATES) (tEstCom/Fates)*, pages 365–377, 2007.

[52] W. Zimmermann. On the correctness of transformations in compiler back-ends. *Lecture Notes in Computer Science*, 4313:74–95, 2006.