Simulation-based Code Duplication for Enhancing Compiler Optimizations*

David Leopoldseder
Johannes Kepler University
Linz, Austria
david.leopoldseder@jku.at

Abstract

Compiler optimizations are often limited by control flow, which prohibits optimizations across basic block boundaries. Duplicating instructions from merge blocks to their predecessors enlarges basic blocks and can thus enable further optimizations. However, duplicating too many instructions leads to excessive code growth. Therefore, an approach is necessary that avoids code explosion and still finds beneficial duplication candidates.

We present a novel approach to determine which code should be duplicated to improve peak performance. Therefore, we analyze duplication candidates for subsequent optimizations by simulating a duplication and analyzing its impact on the compilation unit. This allows a compiler to find those duplication candidates that have the maximum optimization potential.

ACM Reference format:
David Leopoldseder. 2017. Simulation-based Code Duplication for Enhancing Compiler Optimizations. In *Proceedings of Splash 2017 Student Research Competition, Vancouver, Canada, October 2017 (SPLASH'17 SRC)*, 2 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Code duplication [4, 10, 11], is a compiler optimization that moves code from control flow merge blocks to their predecessor blocks. The compiler can *specialize* the duplicated code to the types and values used in predecessor branches, which potentially *enables* subsequent optimizations.

2 Related Work

Duplication approaches for very long instruction word processors [4, 6–8] aim to enlarge basic blocks via tail duplication in order to enable the compiler to perform better instruction selection and scheduling.

Bodík et al. [2] use duplication to perform complete partial redundancy elimination [9].

Mueller and Whalley [10, 11] use code duplication to optimize branches. In [10] they mention the enabling effect of code replication on subsequent optimizations.

These approaches lack awareness about the impact of a duplication on subsequent optimizations, as they do not analyze the optimization potential after duplications.

3 Approach

Our approach is based on simulating the effects of duplications which allows the compiler to estimate the peak performance impact of each possible duplication without the need to actually perform it.

We implemented the approach in a global three-tier algorithm. The algorithm first finds optimization opportunities *(simulation)*, then classifies those opportunities based on their impact *(trade-off)* and finally performs beneficial duplications *(optimization)*.

Simulation Tier We simulate duplications by tentatively moving instructions from merge blocks to their predecessors. We then perform optimizations on those copies and save their optimization potential. This allows us to estimate the impact of every possible duplication without performing the actual transformation. The compiler can then weight up between different duplication candidates.

Simulation incurs significantly less overhead compared to backtracking-based approaches since it does not require to maintain consistent data dependencies for the program, it can be done locally to the merge block.

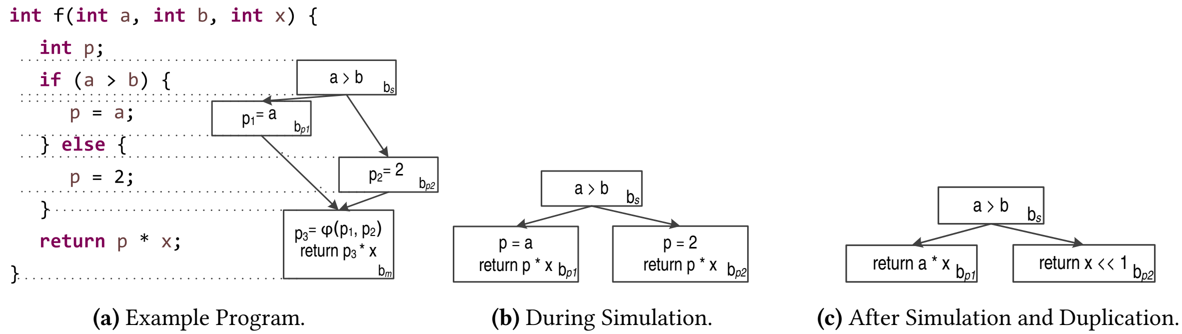
Additionally, after simulation we know for each instruction which optimization triggered on it. Therefore, we can apply those optimizations to the merge block without the need to inspect the entire program.

Figure 1 illustrates a duplication simulation: Figure 1a shows a simple program, Figure 1b shows the control-flow graph after moving the instructions of the merge block b_m into its predecessors. Figure 1c shows the program after applying optimizations on the duplicated code. We see that *copy propagation* and *strength-reduction* removed useless assignments and optimized a multiplication to a shift operation.

Trade-off Tier We use a trade-off function that decides whether a duplication should be performed which tries to maximize peak improvements and minimize code size increase.

*This research project is partially funded by Oracle Labs.

SPLASH'17 SRC, October 2017, Vancouver, Canada
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



(a) Example Program. **(b)** During Simulation. **(c)** After Simulation and Duplication.

Figure 1. Duplication Simulation Sample Program

Optimization Tier The last step of the algorithm duplicates and optimizes those candidates for which the trade-off tier indicated a sufficient optimization benefit.

4 Results

We implemented a prototype of the algorithm in the Graal compiler [5, 12, 13]. The evaluation results show that the duplication optimization significantly increases peak performance of certain applications.

Experiments We measured the performance impact of our optimization with the Java DaCapo [1], the Scala-DaCapo [14] and the JavaScript octane [3] benchmark. We used two configurations: duplication *enabled* and duplication *disabled*.

The generated code with duplication enabled shows peak performance increases from 0% to 30%. None of the benchmarks showed decreased peak performance with duplication enabled. Compile time increases range from 5% to 30% and code size increases range from 2% to 28%.

5 Conclusion

Our work contributes a novel approach to find optimization opportunities enabled by code duplication. Based on the approach we derived a three-tier algorithm that finds and performs beneficial duplication optimizations. We implemented the algorithm in the Graal compiler and show that significant peak performance increases can be obtained by it.

References

[1] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*.

[2] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. 1998. Complete Removal of Redundant Expressions. In *PLDI*.

[3] Stefano Cazzulani. 2012. Octane: The JavaScript benchmark suite for the modern web. *Retrieved December* 21 (2012), 2015.

[4] Pohua P. Chang, Scott A. Mahlke, and Wen-mei W. Hwu. 1991. Using Profile Information to Assist Classic Code Optimizations. *Softw. Pract.& Exper.* (1991).

[5] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation. In *APPLC*.

[6] Joseph A. Fisher. 1995. Instruction-level Parallel Processors. Chapter Trace Scheduling: A Technique for Global Microcode Compaction.

[7] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Quellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. 1995. Instruction-level Parallel Processors. Chapter The Superblock: An Effective Technique for VLIW and Superscalar Compilation.

[8] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. 1992. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *MICRO 25*.

[9] E. Morel and C. Renvoise. 1979. Global Optimization by Suppression of Partial Redundancies. *Commun. ACM* (1979).

[10] Frank Mueller and David B. Whalley. 1992. Avoiding Unconditional Jumps by Code Replication. In *PLDI*.

[11] Frank Mueller and David B. Whalley. 1995. Avoiding Conditional Branches by Code Replication. In *PLDI*.

[12] OpenJDK 2013. Graal Project. (2013). http://openjdk.java.net/projects/graal

[13] OpenJDK 2017. HotSpot Virtual Machine. (2017). http://openjdk.java.net/groups/hotspot/

[14] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da Capo con Scala: design and analysis of a scala benchmark suite for the java virtual machine. In *OOPSLA*.