Simulation-Based Code Duplication for Enhancing Compiler Optimizations*†

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

Abstract

The scope of compiler optimizations is often limited by control flow, which prohibits optimizations across basic block boundaries. Code duplication can solve this problem by extending basic block sizes, thus enabling subsequent optimizations. However, duplicating code for every optimization opportunity may lead to excessive code growth. Therefore, a holistic approach is required that is capable of finding optimization opportunities and classifying their impact.

This paper presents a novel approach to determine which code should be duplicated in order to improve peak performance. The approach analyzes duplication candidates for subsequent optimizations opportunities. It does so by simulating a duplication and analyzing its impact on other optimizations. This allows a compiler to weight up multiple success metrics in order to choose those duplications with the maximum optimization potential. We further show how to map code duplication opportunities to an optimization cost model that allows us to maximize performance while minimizing code size increase.

CCS Concepts • **Software and its engineering → Just-in-time compilers**; **Dynamic compilers**; *Virtual machines*;

Keywords Code Duplication, Tail Duplication, Compiler Optimizations

ACM Reference Format:
David Leopoldseder. 2017. Simulation-Based Code Duplication for Enhancing Compiler Optimizations. In *Proceedings of 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3135932.3135935

*This research participates in both SPLASH SRC and Doctoral Symposium tracks.
†This research project is partially funded by Oracle Labs.

SPLASH Companion'17, October 22–27, 2017, Vancouver, Canada
© 2017 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion'17)*, https://doi.org/10.1145/3135932.3135935.

1 Motivation

Code duplication, often referred to as tail duplication [4] or replication [9, 10], is a compiler optimization that removes code after control merges and copies it into predecessor blocks. This enables a compiler to *specialize* the duplicated code to the types and values used in predecessor branches. Code specialization can *enable* other optimizations. Therefore, a sophisticated duplication strategy is necessary in order to select those duplications that lead to a maximum benefit.

2 Problem

Discovering optimization opportunities after code duplication is a non-trivial task. It requires global knowledge about the data-flow and the control-flow of a program that can only be obtained by complex analysis. There are different approaches to find such opportunities. However, there is no unified approach that finds all kinds of optimizations that can be enabled by code duplication. Even with global knowledge about what kinds of optimization opportunities exist, duplicating the code to enable all of them may lead to excessive code growth [5]. This often has a negative impact on compile time as it increases the workload for subsequent optimizations. Without holistic knowledge about the impact of a duplication a compiler may perform unnecessary (in terms of optimization potential) or even harmful (in terms of code size) transformations.

Many approaches use code duplication to perform optimizations. The subsequent list summarizes the most common ones.

Duplication approaches for very long instruction word processors [4] 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 [8].

Mueller and Whalley [9, 10] use code duplication to optimize away *conditional* and *unconditional* branches. In [9] they mention the enabling effect of code replication on subsequent optimization passes.

We improve upon the work of the presented approaches by evaluating the impact of a duplication on subsequent optimizations before performing it. This allows us to determine if a duplication is indeed beneficial for performance.

3 Approach

This paper presents a novel approach to find optimization opportunities after code duplication. The approach is based on a duplication simulation that finds beneficial (in terms of peak performance) duplication candidates. We implemented the approach in a three-tier algorithm that finds and performs beneficial duplication optimizations. It does so by analyzing a compilation unit and determining all possible optimizations *after* code duplication. The approach is applicable for static and dynamic compilers. It works with common intermediate representations (IR) that use the notion of a control flow graph (CFG). For simplicity, we assume that the IR is in static-single-assignment form [6].

We propose a sequential three-tier algorithm to perform code duplication. The tiers are named *simulation*, *trade-off* and *optimization*. The simulation tier finds optimization opportunities after simulated code duplication. The trade-off tier fits the opportunities into an optimization cost model and the optimization tier performs beneficial duplications. The remainder of this paper describes each of them.

Simulation We illustrate the simulation tier with the sample program f in Figures 1 and 2, which uses a simple CFG-based IR in SSA form. To find subsequent optimization opportunities we simulate the impact of a duplication by performing a depth-first traversal on the dominator tree of the program. Figure 2a shows the dominator tree of f. Every time we process a block like b_{p1} or b_{p2}, which has a merge successor b_m in the CFG, we simulate a duplication by copying b_m and appending it to b_p. Figure 2b shows those two copies: b_{m^1} and b_{m^2}. In the original program, b_m is not dominated by b_{p1} nor b_{p2}. However, after duplication, b_{m^1} and b_{m^2} are appended to b_{p1} and b_{p2}, and are therefore also dominated by them (in fact, they can be viewed as extending b_{p1} and b_{p2}). After processing b_{p^i} we continue simulation in b_{m^i}, which is indicated by the red arrows in Figure 2b. The algorithm performs the simulation traversal for all combinations b_p and b_m of a program.

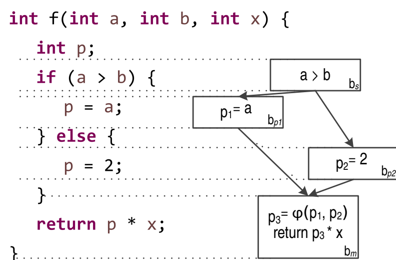


Figure 1. Sample Program with CFG.

In order to find optimization opportunities during the traversal of the copies of b_m, it is necessary for each optimization to determine if it can be applied after duplication. We use the notion of *precondition* and *action*, presented by Chang et al. [4], and split up our optimization phases into two parts. The *precondition* is a predicate which holds if a given IR pattern can be optimized. The *action* step performs the actual optimization. We build *applicability checks* (AC) for all common optimizations. These determine if an optimization can be applied to a given instruction. Before descending into b_{m^1} or b_{m^2}, we replace all φ-functions in them with their inputs on the respective branch. This way ACs behave as if the original program did not contain any merges.

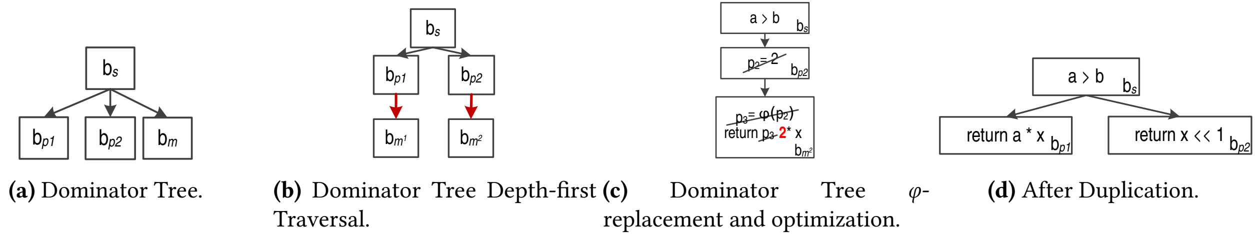
An example of an AC can be seen in Figure 2c, which shows the algorithm during the processing of b_{m^2}. The φ-function p_3 was replaced by the constant 2. Our algorithm tries various optimizations. *Copy propagation* can replace p_3 by p_2 and finally by the constant 2. In `return 2 * x`, *strength reduction* can replace the multiplication with the shift `x << 1`. The optimization potential detected by the ACs is saved and the action steps are performed.

We iterate all instructions of the copies of b_m and apply the ACs on each of them. If an AC triggers, we save the optimization potential and perform the associated action step. The action step can change the instructions in b_{m^1} and b_{m^2}. Note that all of this happens on the copies of b_m, and not the original merge block b_m. Thus we simulate the impact of a duplication.

The algorithm saves the optimization potential and applies the action step on the instructions of b_{m^2}, i.e., it replaces `2 * x` with `x << 1`. Figure 2d shows the program f after duplicating b_m into its predecessors. Simulating a duplication by traversing a copy of b_m yields the same results as doing the actual code duplication. It allows us to save all optimizations that are possible after duplication without the need to modify the original program.

Simulation incurs significantly less overhead compared to backtracking-based approaches since it does not require to maintain consistent data dependencies for the program.

Trade-off We only want to duplicate those sections of code that increase peak performance. Thus, to avoid code explosion, we propose to trade-off between different duplication opportunities based on their impact, both in terms of peak performance and code size. We modify the AC functions such that they not only return whether an optimization is possible, but to also return a performance estimation for the improvement of the optimization. We call those improvement estimates *benefit*. We propose to quantify duplication opportunities by summing up the improvement estimates of the collected optimization opportunities. This allows the compiler to trade-off between all collected duplication opportunities.



(a) Dominator Tree.

(b) Dominator Tree Depth-first Traversal.

(c) Dominator Tree φ-replacement and optimization.

(d) After Duplication.

Figure 2. Duplication Simulation Sample Program

The code size impact, positive or negative, of a duplication opportunity represents its *cost*. Together with the benefit we formulate a global trade-off function that decides if a duplication opportunity is worth to be performed as an optimization task. We derive a function $f(\texttt{benefit}, \texttt{cost})$ that decides whether a duplication should be performed while continuously trying to maximize the sum of all benefits and to minimize the sum of all costs. We propose using profiling information to first optimize those parts of the program that benefit a lot from duplication.

Optimization In the last step of our algorithm we duplicate and optimize those instructions of the merge block for which the simulation tier indicated a sufficient optimization benefit. In other words, we make those simulated optimizations permanent.

4 Evaluation Methodology

To evaluate the presented duplication optimization, we are following an iterative approach. Our first step is to test the hypothesis that enabling optimizations via duplication indeed increases peak performance. We do so by implementing our algorithm in the Graal compiler [7].

We are proposing an incremental experiment to measure crucial success-metrics such as peak performance, code size and compile-time. We use state of the art benchmarks like Java DaCapo [1], Scala-DaCapo [11] and SPECjvm2008 [12]. To evaluate the performance impact on dynamic languages we also use the JavaScript octane [3] benchmark executed on GraalJS, a JavaScript implementation on-top of Truffle [13]. Truffle is a framework for dynamic language implementations provided by Graal.

In Section 3 we proposed the usage of a performance estimator to determine the benefits of a duplication opportunity. We plan to conduct experiments validating the correlation between the performance estimations and the real performance of an application. This is necessary to ensure that our cost model is valid.

Finally, we are planning to evaluate the impact of our trade-off function $f(\texttt{benefit}, \texttt{cost})$ on each success metric. In a first approach we implemented it based on a linear cost model, however, we are planning to evaluate other approaches as well.

5 Conclusion

This paper proposed a novel approach for code duplication that allows a compiler to find beneficial optimization opportunities after code duplication. It combines simulation and optimization into a global three-tier algorithm that enables subsequent optimizations after code duplication. The algorithm first simulates duplications to find optimization opportunities. Then, it trades off between those opportunities by using a performance estimator to quantify them based on their optimization potential and code size effects. In the final step, it performs beneficial duplications that maximize the peak performance while minimizing the code size impact.

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. 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] Keith D Cooper, Kathryn S Mckinley, and Linda Torczon. 1998. Compiler-Based Code-Improvement Techniques. (1998).

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *TOPLAS* (1991).

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

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

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

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

[11] 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*.

[12] Standard Performance Evaluation Corporation. 2008. SPECjvm2008. (2008). http://www.spec.org/jvm2008/

[13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Onward*.