SPLASH: G: Simulation-based Code Duplication for Enhancing Compiler Optimizations

David Leopoldseder
JKU Linz
Austria
david.leopoldseder@jku.at

Abstract

Compiler optimizations are often limited by control flow merges because such merges prohibit optimizations across basic block boundaries. Code duplication can tackle this problem by duplicating instructions from merge blocks to their predecessors enabling the compiler to specialize duplicated instructions to predecessor blocks.

Excessive code duplication can lead to exponential code growth. Thus, a sophisticated approach is required to find those duplication candidates that can improve the performance of an application.

This paper presents a code duplication approach that uses simulation to determine which code duplications increase peak performance. It does so by simulating the effects of a duplication on a compilation unit and analyzing its impact on subsequent optimizations. This allows a compiler to weight up multiple success metrics in order to select those duplications with the maximum optimization potential.

1 Problem & Motivation

Control flow merges often limit compiler optimizations by prohibiting optimizations across basic block boundaries. Optimizations that depend on the data flow of a program can often be specialized to the predecessor blocks of a merge. Code duplication [21, 22], a compiler optimization that hoists, i.e. duplicates, instructions from merge blocks to predecessors, can tackle this problem. The compiler can then *specialize* the duplicated code to the types and values used in predecessor branches, which potentially *enables* subsequent optimizations.

Consider the program from Figure 1a. The variable `phi` is assigned two values: `x` and the constant `0`. The usage of the variable `phi`, the addition, could be optimized if it would have the constant `0` instead of the variable phi as input. Figure 1b shows the pseudo code after duplicating the instruction `return 2 + phi` into the predecessor branches and after a *copy propagation* step that removes the variable `phi`. In this example, we created an optimization opportunity in the instruction `2 + 0` that can be optimized via *constant folding* to the code in Figure 1c.

ACM SRC Grand Finals, 2018
2018.

Code duplication inherently trades-off code size versus peak performance. It improves peak performance at the cost of increased code size. Duplicating code heuristically or tentatively often leads to exponential code growth. Therefore, a sophisticated approach is required that allows a compiler to determine which duplication candidates carry a sufficient peak performance increase and should thus be considered for optimization. Since the duplication of instructions into predecessor blocks is compile-time intensive, we require a solution that does not perform a full duplication to determine its effect on subsequent optimizations.

Contributions In this paper we present a simulation-based code duplication optimization that allows compilers to simulate the effects of a code duplication without actually performing it. We claim three major contributions:

- Our approach determines which duplications will increase peak performance, allowing the compiler to gain a maximal performance increase at a moderate code size and compile time increase.
- We developed a fast, dominance-based, duplication simulation algorithm to deterministically find all subsequent optimizations that are enabled by a duplication.
- We devised a duplication optimization cost model that tries to maximize peak performance while minimizing code size and compilation time. This model is used to guide code duplication.

2 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 our approach in an algorithm entitled *Dominance-based Duplication Simulation (DBDS)* [19]. The basic algorithm is depicted in Figure 2. The simulation tier discovers optimization opportunities after code duplication. The trade-off tier fits those opportunities into an optimization cost-model that tries to maximize peak performance while minimizing code size and compilation time. The outcome is a set of duplication transformations that should be performed as they lead to sufficient peak performance improvements. The optimization tier then performs those duplications together with the subsequent optimizations whose potential was detected by the simulation tier.

```
int foo(int x){
 final int phi;
 if(x>0)
  phi = x;
 else
  phi = 0;
 return 2 + phi;
}
```

(a) Initial Program.

```
int foo(int x){
 if(x>0)
  return 2 + x;
 else
  return 2 + 0;
}
```

(b) After Duplication.

```
int foo(int x){
 if(x>0)
  return 2 + x;
 else
  return 2;
}
```

(c) After Optimization

Figure 1. Constant Folding (CF) Optimization Opportunity.

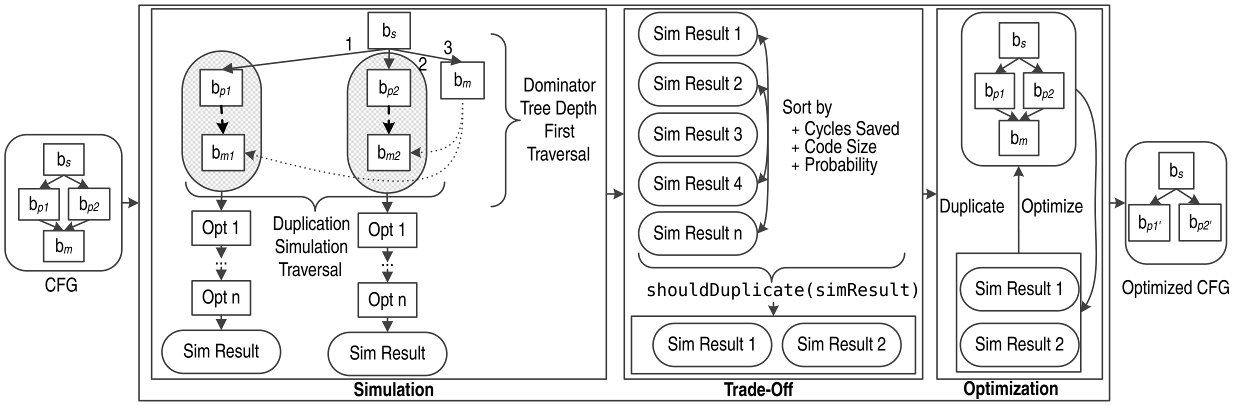


Figure 2. DBDS Algorithm Schematic.

2.1 Simulation

We use the algorithm schematic from Figure 2 and the concrete example program f from Figure 3a, which uses Graal IR [11, 18, 24], to illustrate the simulation tier. Graal IR is a sea-of-nodes-based [9] directed graph in SSA form [10]. Each IR node produces at most one value. Data flow is represented with upward edges and control flow with downward edges. Control-flow nodes represent side-effecting instructions that are never re-ordered, whereas data-flow nodes are *floating*. Their final position in the generated code is purely determined by their data dependencies.

In order to find optimization opportunities, we simulate duplication operations for each predecessor-merge pair of the CFG and apply optimizations on them. If an optimization triggers during simulation, we remember the optimization potential in relation to the predecessor-merge pair. The result of the simulation is the optimization potential for each potential duplication. The entire approach is based on a *depth-first* traversal of the dominator tree [5] as outlined in the simulation part of Figure 2.

Traversing the dominator tree during simulation is beneficial as it allows us to use the information of dominating conditions for optimization. In a depth-first traversal of a `true` branch we know its condition holds. We use this additional control-flow-sensitive information for optimizations.

During the depth first traversal, every time we process a basic block b_{pi}, which has a *merge* block successor b_m in the CFG (as seen by the gray background in Figure 2), we pause the current traversal and start a so-called *duplication simulation traversal* (DST). The DST re-uses the context information of the paused depth-first traversal and starts a new depth-first traversal at block b_{pi}. However, in the DST we process block b_m directly after block b_{pi} as if b_{pi} would dominate b_m. In other words, we *extend* the block b_{pi} with the instructions of block b_{mi}. The index i indicates a specialization of b_m to the predecessor b_{pi}.

This way we simulate a duplication operation. This is the case because in the original CFG, duplicating instructions from b_{mi} into b_{pi} effectively appends them to the block b_{pi}. As every block trivially dominates itself, the first instruction of b_{pi} dominates its last instruction and therefore also the duplicated ones.

Consider the sample program f in Figure 3a and its dominator tree in Figure 3b. Program f consists of 4 basic blocks: the start block b_s, the `true` branch b_{p1}, the `false` branch b_{p2} and the merge block b_m. We simulate a duplication operation by starting two DSTs at block b_{p1} and b_{p2}. This can be seen in Figure 3c by the dashed arrows. We process b_{mi} in both DSTs as if it were dominated by b_{pi}. We perform each DST until the next merge or split instruction occurs.

During the DSTs we need to determine which optimizations are enabled after duplication. Therefore, we split up our optimization phases into two parts, the *precondition* and the *action* step. This scheme was presented by Chang et al. [8]. The *precondition* is a predicate that holds if a given IR pattern can be optimized. The associated action step performs the actual transformation. Based on the preconditions we derive boolean functions called *applicability checks* (AC) that determine if a precondition holds on a given IR pattern. We build ACs and action steps for all optimizations that are possible enabled by a code duplication. [1]

Based on the action step's result we compute a numeric peak performance increase estimation by using a static performance estimator for each IR node. The performance estimator returns a numeric run time estimation (called *cycles*) as well as a code size estimation for each IR node. We compute a *cycles saved (CS)* measurement which tells us if a given optimization might increase peak performance. We compute code size increase in a similar fashion. The performance estimator is based on a performance cost model for IR nodes. Each IR node is given a *cycle* and *size* estimation that allows us to compare two nodes (instructions) with each other.

We want to avoid copying any code during DST. However, the code in b_{mi} still contains phi instructions and not the input of a phi on the respective branch. Therefore, we introduce the concept of so-called *synonym* mappings. A *synonym map* maps a φ node to its input on the respective DST predecessor of b_{mi}. Before we descend into b_{mi}, we create the synonym map for all φ instructions in b_{mi} based on their inputs from b_{pi}. We can see such a mapping in Figure 3d, which shows the algorithm during the DST of the blocks $b_s \rightarrow b_{p2} \rightarrow b_{m2}$. The *synonym of* relation in Figure 3d shows a mapping from the constant 2 to the φ node. The constant 2 is a synonym of the φ on the predecessor b_{p2}.

During the simulation traversal we iterate all instructions (nodes) of the block b_{mi} and apply ACs on them. If an AC triggers, we perform the associated action step of the optimization, compute the CS and save it in a data structure associated with the block pair $< b_{pi}, b_{mi} >$. Additionally, we save new IR nodes as synonyms for the original nodes. The ACs access input nodes via the synonym map.

Figure 3d shows the steps of the algorithm during the traversal. We save value and type information for each involved IR node and update it regularly via the synonym mapping. Eventually, we iterate the division operation (`x / φ`) in b_{m2} and apply a *strength reduction* [1] AC on it. It returns `true` and we perform the action step. The action step returns a new instruction (`x >> 1`), which we save as a synonym for the division node. Our static performance estimator yields that the original division needs 32 cycles to execute while the shift only takes 1 cycle. Therefore, the cycles saved (CS) is computed as $32 - 1 = 31$, i.e., we estimate that performing the duplication and subsequent optimizations reduces the run time of the program by 31 cycles.

For completeness, we illustrate the optimized program f in Figure 3e, which shows that all optimizations found during simulation are indeed performed after duplication.

The result of the simulation tier is a list of simulation results capturing the code size effect and the optimization potential of each possible duplication in the compilation unit (see trade-off part in the middle of Figure 2).

2.2 Trade-off

We want to avoid code explosion and unnecessary (in terms of optimization potential) duplication transformations. To do so, we consider the benefits of the duplication candidates discovered during the simulation tier. Based on their optimization potential (*benefit*) and their *cost* we select the most promising ones for duplication. This can be seen in the middle part of Figure 2. We take the candidates from the simulation tier and sort them by benefit, cost and probability. We then decide for each candidate if it is beneficial enough to perform the associated duplication. The decision is made by a trade-off function, that tries to maximize peak performance while minimizing code size increase. The trade-off function is based on cost and benefit. We formulate it as the following heuristic:

$$\begin{aligned} c &\ldots \text{Cost} \\ b &\ldots \text{Benefit} \\ p &\ldots b_{pi} \text{ Probability} \\ cs &\ldots \text{Compilation Unit Size} \\ MS &\ldots \text{Max Compilation Unit Size} \\ BS &\ldots \text{Benefit Scale Factor} = 256 \\ (b \times p \times BS) &> \ c \wedge (cs < MS) \end{aligned} \tag{shouldDuplicate}$$

We decide upon the value of duplications by computing their cost / benefit ratio. We allow the cost to be 256× higher than the benefit. We derived the constant 256 during empirical evaluation. All duplication candidates for which `shouldDuplicate` returns `true` are passed to the optimization tier.

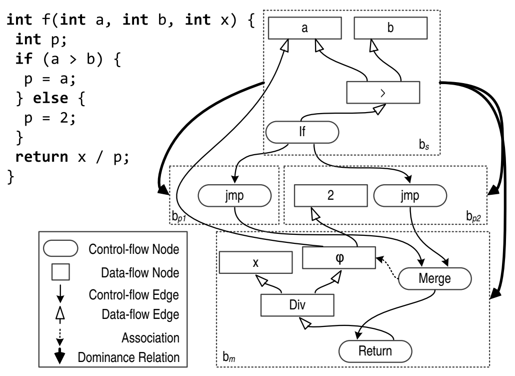
2.3 Optimization

Based on the decisions made during the trade-off we perform code duplication and the action steps of all the selected optimizations.

3 Evaluation

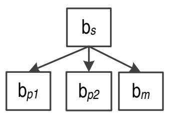
Implementation We implemented our approach in the Graal compiler [23, 27], a Java-to-machine code just-in-time compiler running on top of the HotSpot [15] Java Virtual Machine.

[1]Optimizations enabled by code duplications include: constant folding, strength reductions, conditional elimination, partial escape analysis and scalar replacement.

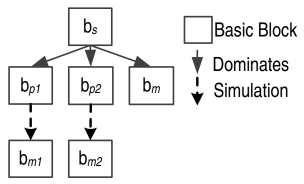


```
int f(int a, int b, int x) {
 int p;
 if (a > b) {
  p = a;
 } else {
  p = 2;
 }
 return x / p;
}
```

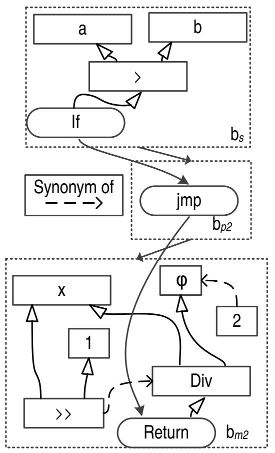
(a) Example program f.



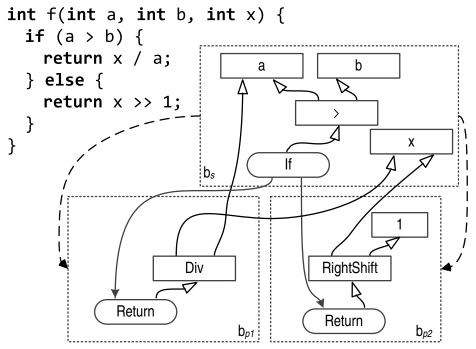
(b) Program f dominator tree.



(c) Program f Duplication Simulation.



(d) Example during duplication simulation.



```
int f(int a, int b, int x) {
  if (a > b) {
    return x / a;
  } else {
    return x >> 1;
  }
}
```

(e) Example After Duplication.

Figure 3. Sample Program f.

Results We evaluated the proposed simulation-based code duplication optimization on the Java DaCapo [3], Scala DaCapo [25] and JavaScript Octane [6] benchmarks[2]. We used a cluster of Sun X3-2 (Intel Sandy Bridge Xeon E5-2260) machines for the evaluation. We tested two configurations: Graal without our optimization (baseline) and a configuration with the simulation-based duplication enabled. We measured three metrics: peak performance (computed from run time or throughput), code size and compile time. We normalized all numbers to the baseline and present them in the boxplot [13] in Figure 4.

Our experiments show that our simulation-based code duplication can reach peak performance improvements of up to 40%, with a mean peak performance increase of 5.89%, while it generates a mean code size increase of 9.93% and a mean compile time increase of 18.44% (using geometric means).

[2] We executed the JavaScript benchmarks with GraalJS [27], a JavaScript runtime implemented on-top of Truffle [28] that uses partial evaluation [14] to generate machine code.

4 Related Work

Mueller and Whalley [21, 22] proposed *replication* to optimize away branches. Their approach relates to ours in that we also duplicate code to remove conditional branches.

Splitting [7] is an approach developed in the context of the *Self* compiler to tackle the problems of virtual dispatch [17]. We improved upon this idea by using a performance estimator, to estimate the peak performance increase of a duplication transformation, and by profiling information from the VM to only optimize those parts of the program that are often executed.

Advanced Scheduling approaches [12, 16, 20] apply tail duplication [8] in order to increase the instruction level parallelism [26] which is needed to optimize code for VLIW processors which require elaborate compiler support to generate efficient code.

Ball [2] estimates the effects of subsequent optimizations on inlined functions by using a data flow analysis on the inlinee to derive *parameter dependency sets* for all variables. We improve upon their approach by enabling a larger set of optimizations including control-flow dependent ones.

Bodík et al. [4] perform complete partial redundancy elimination using duplication. We improved upon their work by computing a general set of optimizations that are enabled by duplication.

References

[1] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-performance Computing. *ACM Comput. Surv.* 26, 4 (Dec. 1994).

[2] J. Eugene Ball. 1979. Predicting the Effects of Optimization on a Procedure Body. *SIGPLAN Not.* 14, 8 (Aug. 1979).

[3] 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 2006.*

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

[5] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. 1997. Value numbering. *Software-Practice and Experience* (1997).

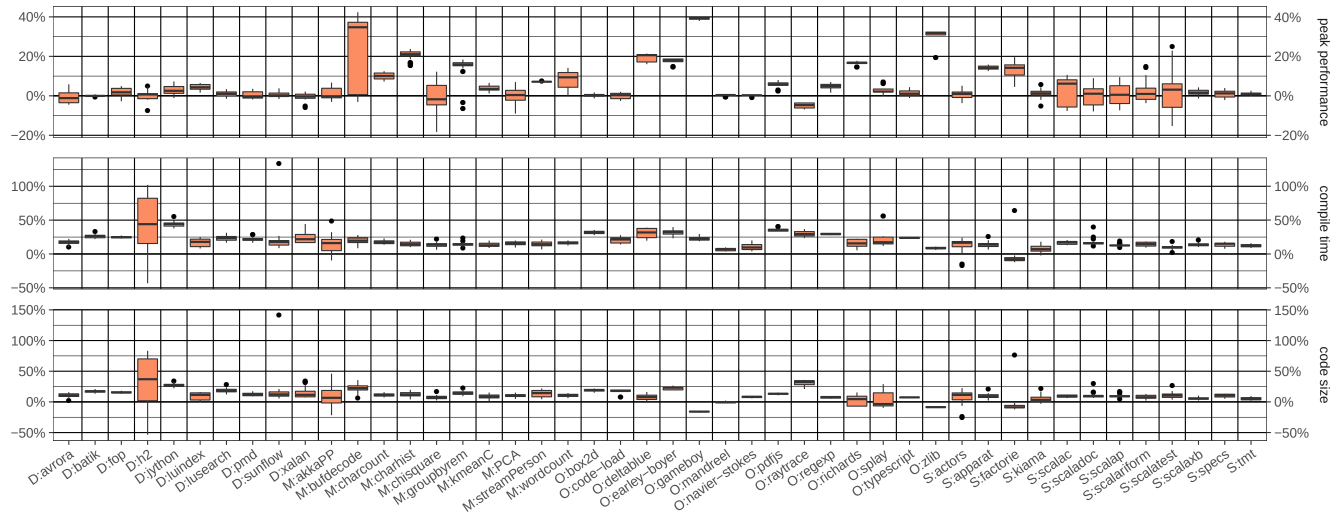


Figure 4. DBDS Benchmarks Combined. Benchmark prefixes specify the benchmark suite: D = DaCapo, S = ScalaDaCapo, M = MicroBenchmarks and O = Octane. Metrics: peak performance (higher is better), compile-time (lower is better), code size (lower is better).

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

[7] Craig David Chambers. 1992. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-oriented Programming Languages*. Ph.D. Dissertation.

[8] Pohua P. Chang, Scott A. Mahlke, and Wen-mei W. Hwu. 1991. Using Profile Information to Assist Classic Code Optimizations. *Softw. Pract. Exper.* 21 (1991).

[9] Cliff Click and Keith D. Cooper. 1995. Combining Analyses, Combining Optimizations. *ACM Trans. Program. Lang. Syst.* 17, 2 (March 1995).

[10] 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. *ACM Trans. Program. Lang. Syst.* 13 (1991).

[11] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *VMIL 2013*.

[12] Joseph A. Fisher. 1995. Instruction-level Parallel Processors. IEEE Computer Society Press, Chapter Trace Scheduling: A Technique for Global Microcode Compaction.

[13] Michael Frigge, David C. Hoaglin, and Boris Iglewicz. 1989. Some Implementations of the Boxplot. *The American Statistician* 43, 1 (1989). http://www.jstor.org/stable/2685173

[14] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* (1999).

[15] HotSpot JVM 2013. Java Version History (J2SE 1.3). (2013). http://en.wikipedia.org/wiki/Java_version_history

[16] 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. IEEE Computer Society Press, Chapter The Superblock: An Effective Technique for VLIW and Superscalar Compilation.

[17] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. 2000. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. In *OOPSLA 2000*.

[18] David Leopoldseder, Lukas Stadler, Christian Wimmer, and Hanspeter Mössenböck. 2015. Java-to-JavaScript Translation via Structured Control Flow Reconstruction of Compiler IR. In *DLS 2015*. ACM.

[19] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations. In *CGO 2018*.

[20] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. 1995. Instruction-level Parallel Processors. IEEE Computer Society Press, Chapter Effective Compiler Support for Predicated Execution Using the Hyperblock.

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

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

[23] OpenJDK 2017. GraalVM -New JIT Compiler and Polyglot Runtime for the JVM;. (2017). http://www.oracle.com/technetwork/oracle-labs/program-languages/overview/index-2301583.html

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

[25] 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 *OOSPLA 2011*.

[26] David W. Wall. 1991. Limits of Instruction-level Parallelism. In *ASPLOS 1991*.

[27] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *PLDI 2017*.

[28] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. In *DLS 2012*.