Dominance-Based Duplication Simulation (DBDS)*

Code Duplication to Enable Compiler Optimizations

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

Lukas Stadler
Oracle Labs
Linz, Austria
lukas.stadler@oracle.com

Thomas Würthinger
Oracle Labs
Zurich, Switzerland
thomas.wuerthinger@oracle.com

Josef Eisl
Johannes Kepler University Linz
Austria
josef.eisl@jku.at

Doug Simon
Oracle Labs
Zurich, Switzerland
doug.simon@oracle.com

Hanspeter Mössenböck
Johannes Kepler University Linz
Austria
hanspeter.moessenboeck@jku.at

Abstract

Compilers perform a variety of advanced optimizations to improve the quality of the generated machine code. However, optimizations that depend on the data flow of a program are often limited by control-flow merges. Code duplication can solve this problem by hoisting, i.e. duplicating, instructions from merge blocks to their predecessors. However, finding optimization opportunities enabled by duplication is a non-trivial task that requires compile-time intensive analysis. This imposes a challenge on modern (just-in-time) compilers: Duplicating instructions tentatively at every control flow merge is not feasible because excessive duplication leads to uncontrolled code growth and compile time increases. Therefore, compilers need to find out whether a duplication is beneficial enough to be performed.

This paper proposes a novel approach to determine which duplication operations should be performed to increase performance. The approach is based on a duplication simulation that enables a compiler to evaluate different success metrics per potential duplication. Using this information, the compiler can then select the most promising candidates for optimization. We show how to map duplication candidates into an optimization cost model that allows us to trade-off between different success metrics including peak performance, code size and compile time.

We implemented the approach on top of the GraalVM and evaluated it with the benchmarks Java DaCapo, Scala DaCapo, JavaScript Octane and a micro-benchmark suite, in terms of performance, compilation time and code size increase.

We show that our optimization 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 mean compile time increase of 18.44%.

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

Keywords Code Duplication, Tail Duplication, Compiler Optimizations, Just-In-Time Compilation, Virtual Machines

ACM Reference Format:
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 *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3168811

1 Introduction

Compiler optimizations are often restricted by control-flow merges that prohibit optimizations across basic block boundaries. Code duplication can solve this problem by copying code from merge blocks into predecessor blocks. This enables a compiler to *optimize* the duplicated code for the respective predecessor.

Consider the pseudo code example in Figure 1a. The variable `phi` is assigned two inputs, the parameter `x` in the `true` branch and the constant `0` in the `false` branch. The usage of `phi` in the addition could be optimized if its input would be the constant `0` instead of the variable `phi`. In Figure 1b we can see the example after duplicating the merge block into the predecessor branches. *Copy propagation* eliminates the variable `phi`. This creates the constant folding opportunity `2 + 0`, that can be optimized to the code in Figure 1c.

However, duplicating code tentatively at every merge is not feasible in terms of code size and compile time. Therefore, we need to limit the number of duplications by determining before-hand which of them are beneficial for performance. To do so, we need to solve three problems:

*This research project is partially funded by Oracle Labs.

CGO'18, February 24–28, 2018, Vienna, Austria

```
1 int foo(int x){
2  final int phi;
3  if(x>0) phi = x;
4  else phi = 0;
5  return 2 + phi;
6 }
```

(a) Initial Program.

```
1 int foo(int x){
2  if(x>0){
3   return 2 + x;
4  }else {
5   return 2 + 0;
6  }
7 }
```

(b) After Duplication.

```
1 int foo(int x){
2  if(x>0){
3   return 2 + x;
4  }else {
5   return 2;
6  }
7 }
```

(c) After Optimziation.

Figure 1. Constant Folding (CF) Optimization Opportunity.

P1 We need to determine which duplications will increase peak performance.

P2 To solve P1, we need to know which optimizations will be enabled by a certain duplication.

P3 Finding optimization opportunities after duplication requires complex data and control-flow analysis. Therefore, we need to find a way to perform this kind of analysis in acceptable time in a JIT compiler.

P1 A problem that was not solved yet by other approaches, is the question of *what* to duplicate to increase peak performance. Duplicating code at every merge is not feasible. It can lead to uncontrolled code growth and compile time increases. Thus, we present a novel solution to this problem. We use a *static performance estimator* [20] to estimate the peak performance increase of a single duplication operation before performing it. This allows us to select those duplication candidates that promise a high benefit at a low cost (code size and compile time).

P2 Every instruction in SSA [10] which has a φ as its input is a potential target for follow-up optimizations after code duplication. However, determining which optimizations are enabled by a duplication and on which instructions is a non-trivial task. We present an algorithm to find such opportunities that allows a compiler to decide *before-hand*, which duplications carry the best cost / benefit trade-off.

P3 Finding optimization opportunities after duplication is compile-time intensive, as it requires to know which instructions are optimizable after a duplication. There are two possible approaches to determine if an instruction can be optimized after duplication: *backtracking* and *simulation*. *Backtracking* performs a duplication and any subsequent optimizations and then determines whether this lead to a benefit. If no progress was made, it restores the version of the intermediate representation (IR) without the duplication. *Simulation*, on the other hand, determines *before-hand*, if an optimization will be possible after duplication. Although, simulation may be significantly faster that backtracking, it is not straightforward to simulate a duplication and its subsequent optimizations. We present a fast *dominance-based duplication simulation* (DBDS) algorithm that allows a compiler to find optimization opportunities after duplications.

In summary this paper contributes the following:

- We present a novel algorithm for duplication simulation that solves the problem of finding optimization opportunities after duplications. We argue that a simulation-based approach is favorable over a backtracking-based approach (Sections 2 to 4).
- We present a duplication optimization cost model that tries to maximize peak performance while minimizing code size and compilation time (Sections 4 and 5).
- We integrated our approach in an SSA-based Java bytecode-to-native code just-in-time (JIT) compiler (Section 5).
- To validate our claims, we performed extensive experiments using the Java DaCapo, the Scala DaCapo, the JavaScript Octane benchmarks, and a micro-benchmark suite, showing performance improvements of up to 40% with a mean peak performance increase of 5.89% (Section 6).

2 Optimization Opportunities after Code Duplication

In this section we discuss basic optimization opportunities after code duplication.

Constant Folding (CF) can be applied after duplication if, e.g., one operand of an instruction is a *constant* and the other one is a φ. We illustrated this with Figure 1, which shows a trivial program. The instruction `2 + phi` references a φ instruction. After duplication, the instructions of the merge block are copied into the predecessor blocks and the φ is eliminated. Constant folding can then optimize the duplicated addition in the false branch to the constant 2.

Conditional Elimination (CE) [30], i.e., the process of symbolically proving conditional statements also profits from duplication. Consider the example in Listing 1. In case the first condition does not hold and `i <= 0`, the second condition `p > 12` is known to be `true`. We can duplicate the rest of the method into the predecessor branches and eliminate the condition in the `else`-branch to produce the code seen in Listing 2.

Partial Escape Analysis and Scalar Replacement (PEA) [31] opportunities arise when an object *escapes* through the usage in a φ instruction [10]. This frequently happens in Java and Scala because of *auto-boxing* [21]. Listing 3 shows a PEA example. There would be no need for the `A` object to be allocated, except that it is used by a φ instruction. After

```
1  int foo(int i){
2   int p;
3   if(i > 0){
4    p = i;
5   }else{
6    p = 13;
7   }
8   // merge block m
9   if(p > 12){
10   return 12;
11  }
12  return i;
13 }
```

Listing 1. CE Opportunity.

```
int foo(int i){
 if (i > 0) {
  if (i > 12) {
   return 12;
  }
  return i;
 } else {
  return 12;
 }
```

Listing 2. CE Opportunity After Duplication.

```
1  class A{
2   int x;
3   A(int x){this.x=x;}
4  }
5  int foo(A a) {
6   A p;
7   if (a == null) {
8    p = new A(0);
9   } else {
10   p = a;
11  }
12  return p.x;
13 }
```

Listing 3. PEA Opportunity.

```
int foo(A a) {
 if (a == null) {
  return 0;
 } else {
  return a.x;
 }
}
```

Listing 4. PEA Opportunity After Duplication.

```
1  class A{int x;}
2  static int s;
3  int foo(A a,int i){
4   if(i > 0){
5    // Read1
6    s = a.x;
7   } else {
8    s = 0;
9   }
10  // Read2
11  return a.x;
12 }
```

Listing 5. Read Elim Opportunity.

```
class A{int x;}
static int s;
int foo(A a,int i){
 if(i > 0){
  // Read1
  int tmp = a.x;
  s = tmp;
  return tmp;
 } else {
  s = 0;
  // Read2
  return a.x;
 }
}
```

Listing 6. Read Elim Opportunity After Duplication.

duplication, (P)EA can deduce that the allocation is no longer needed. Thus, scalar replacement can reduce it to the code in Listing 4.

Read Elimination is the process of eliminating redundant reads. Due to control-flow restrictions, only fully redundant reads [4] can be eliminated. However, we can promote partially redundant reads to be fully redundant by duplicating them. Consider the example in Listing 5. `Read2` is redundant if the `true`-branch (`i > 0`) is taken. By duplicating `Read2` into both predecessors, it becomes fully redundant in the `true`-branch, and can be eliminated. This can be seen in Listing 6.

3 Finding Optimization Opportunities after Code Duplication

To determine which duplications will increase peak performance (**P1**), we must determine which optimizations will be enabled by a duplication .

3.1 Backtracking

One possible way of determining what to duplicate is to tentatively perform a duplication at a merge, and then backtrack if no improvement was possible. Consider the illustration of a backtracking-based approach in Algorithm 1. To determine if progress was made, we first copy the control-flow graph (CFG) as a backup and perform a duplication on the original CFG. Then we apply a set of selected optimizations on it. If any of those optimizations is able to optimize our IR, we re-start with the *changed* CFG. If no improvement was made, we backtrack to the *copied* (original) CFG and *advance* to another merge.

Backtracking has three disadvantages: First, copying the CFG for every predecessor-merge pair is compile-time intensive and thus typically not suitable for a (JIT) compiler. We did experiments in our implementation in Graal and the copy operation increased compilation time by a factor of 10. The main problem of the copy operation is that we need to copy the entire IR and not only the portions which are relevant for duplication. This is the case because we do not know which parts of the IR are changed by subsequent optimizations.

Second, the compile-time impact of a duplication is not known in advance. In the context of SSA form, code duplication can require complex analysis to generate valid φ instructions for usages in dominated blocks.

Finally, large compilation units ($>100k$ instructions) contain thousands of control-flow merges. Applying optimization phases after every duplication to determine if an optimization triggered is therefore not feasible. Even though the presented opportunities can all be computed in (nearly) linear time over the number of instructions or the IR, processing the full IR every time we performed a duplication is not acceptable for JIT compilation.

3.2 Simulation

We argue that *simulation*-based approaches do not suffer the problems of backtracking-based approaches. The main idea of simulation-based duplication is to determine the impact (in terms of optimization potential on the whole compilation unit) *before* performing any code transformation. This allows a compiler to only perform those transformations that promise a *sufficient peak performance increase (benefit)*. The main requirement for this to be practical, is that simulating a duplication is sufficiently less complex in compilation

```
Data: ControlFlowGraph cfg
Result: Optimized ControlFlowGraph cfg
bool progressMade ← true;
outer: while progressMade do
    progressMade ← false;
    for BasicBlock b in cfg.mergeBlocks() do
        bool change ← false;
        ControlFlowGraph copy = cfg.copy();
        for BasicBlock p in b.predecessors() do
            duplicate (cfg, p, b);
            for opt o in {CE, CF, PEA, ReadElim, StrengthRedux} do
                change ← o.do(cfg);
            end
        end
        if change then
            /* The CFG and basic block list changed, thus we
               need to restart.                                */
            progressMade ← true;
            continue outer;
        else
            /* Backtrack and advance one merge.                */
            cfg ← copy;
        end
    end
end
```

Algorithm 1. Backtracking-based Duplication

```
Data: ControlFlowGraph cfg
Result: Optimized ControlFlowGraph cfg
simResults ← [];
for BasicBlock b in cfg.mergeBlocks() do
    for BasicBlock p in b.predecessors() do
        simCFG ← CFG after simulated duplication of b into p;
        simResult ← result of partial opt (CE, CF, ...) applied to simCFG;
        simResults.add (simResult);
    end
end
sort simResults by benefit and cost;
for SimResult s in simResults do
    if s.worthDuplicating() then
        duplicate (s.merge,s.predecessor);
        for opt o in {CE, CF, PEA, ReadElim, StrengthRedux} do
            o.applyPartial(s.predecessor)
        end
    end
end
```

Algorithm 2. Simulation-based Duplication.

time than performing the actual transformation. Duplication simulation should avoid the complex part of the duplication transformation, maintaining the semantic correctness of all data dependencies, while still allowing valid decisions about the optimization potential of the involved instructions after duplication.

Algorithm 2 outlines the basic idea. Before performing any duplication we simulate each duplication and perform *partial* optimizations on the simulation result. Those optimizations are local to the predecessor blocks of the merge and thus faster than a full optimization over the entire IR. For each partial optimization we save the optimization potential (see Section 4). In addition, we store an *estimated code size increase* for the duplication. We perform the simulation for each predecessor-merge pair and store all of the results in a list. The results are sorted ascending by benefit (optimization potential, i.e., expected performance increase) and cost (code size increase), to optimize the most promising candidates first. This is important in case not all candidates are duplicated due to code size restrictions. We then iterate each simulation result and decide if the related transformation is worth it.

4 Dominance-based Duplication Simulation

In order to perform beneficial code duplications, we propose a *three-tier* algorithm combining *simulation*, *trade-off analysis*, and *optimization*. The basic idea of this DBDS algorithm follows Algorithm 2 from Section 3.2 and 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.

4.1 Simulation

We use the algorithm schematic from Figure 2 and the concrete example program f from Figure 3a, which uses Graal IR [11, 12, 19], 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. The IR is a superposition of the control-flow and the data-flow graph. 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 partial optimizations on them. If a partial 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. Every split in the control-flow graph narrows the information for a dominating condition's operands. For example, an instruction `if (a != null)` has two successors in the dominator tree: the `true` and the `false` branch. In a depth first traversal of the `true` branch

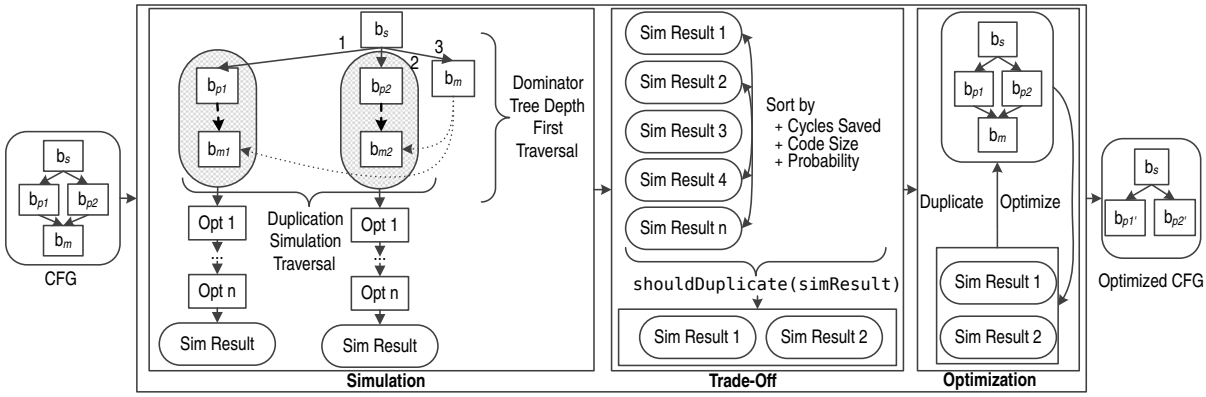


Figure 2. DBDS Algorithm Schematic.

we know that `(a != null)` holds. We use this additional control-flow-sensitive information for optimization.

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 first instruction after the next merge or split instruction.

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 presented in Section 2. Additionally, we modify the action steps to not change the underlying IR but to return new (sub)graphs containing the result of the optimization. We use the result of the action steps to estimate a peak performance increase and a code size increase. We compare the resulting IR nodes of the action step with the original IR. There are several possible results of the action step:

- Empty: The action step was able to eliminate the node(s).
- New Node: The action step returns new nodes that represent the semantics of the old ones and will replace them.
- Redundant Node: The action step found out that an instruction in a dominating block already computed the same result, so the redundant node can be eliminated.

Based on this comparison 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 which nodes are new or deleted and can therefore 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 (see Section 5.3 for details).

As stated, we want to avoid copying any code during DST. However, the code in b_m 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_m 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.

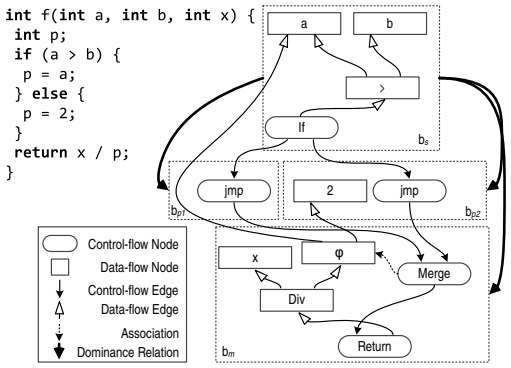
The result of the simulation tier is a list of simulation results capturing the code size effect and the optimization potential of each possible duplication in the compilation unit (see trade-off part in the middle of Figure 2).

4.2 Trade-off

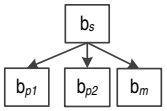
We want to avoid code explosion and unnecessary (in terms of optimization potential) duplication transformations. To do so, we consider the benefits of the duplication candidates discovered during the simulation tier. Based on their optimization potential (*benefit*) and their *cost* we select the most 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 `boolean` function `shouldDuplicate`$(b_{pi}, b_m, benefit, cost)$ (see Section 5.4). It decides if a given duplication transformation should be performed. All duplication candidates for which `shouldDuplicate` returns `true` are passed to the optimization tier.

4.3 Optimization

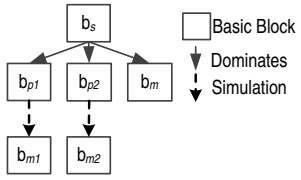
The last tier of the algorithm is the actual optimization step. Based on the decisions made during the trade-off we perform code duplication and the action steps of all the selected optimizations.



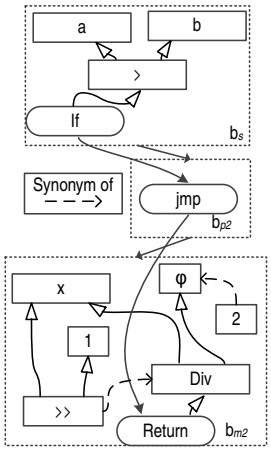
(a) Example program f.



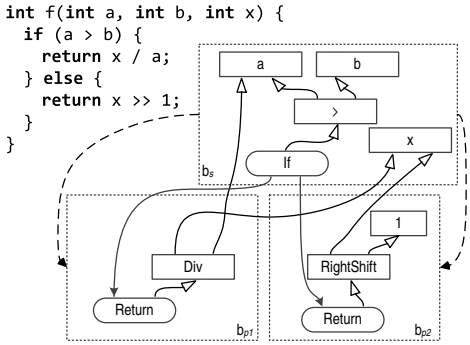
(b) Program f dominator tree.



(c) Program f Duplication Simulation.



(d) Example during duplication simulation.



(e) Example After Duplication.

Figure 3. Sample Program f.

5 Implementation

We implemented the DBDS algorithm in the Graal compiler [11, 25], a dynamically optimizing compiler for the HotSpot VM [26]. Graal is a Java just-in-time (JIT) compiler, which is itself written in Java. We use a particular version of HotSpot called GraalVM in which Graal replaces C2 [27]

```
@NodeInfo(cycles = CYCLES_8, cyclesRationale =
    "tlab alloc + header init", size = SIZE_8)
public abstract class AbstractNewObjectNode {...}
```

Listing 7. Node cost annotation for AbstractNewObjectNode.

as HotSpot's standard top-tier JIT compiler. Graal generated code is comparable in performance to C2 generated code [28].

5.1 System Overview

Graal translates Java bytecode to machine code in multiple steps. The compilation process is divided into a front end and a back end. From the parsed bytecodes Graal IR is generated [11]. The front end performs platform-independent high-level Java optimizations such as inlining and partial escape analysis [31]. The IR is then lowered into a platform specific version on which additional optimizations and register allocation are done. In a final step machine code is emitted.

5.2 Implementation Details

We implemented the DBDS algorithm in the front end of Graal. We first simulate duplications on all predecessor-merge pairs in our IR, sort them by benefit and cost and finally perform the optimization step. The entire DBDS algorithm (simulate → trade-off → optimize) is applied iteratively with a maximum upper bound of 3 iterations. This is necessary because we currently do not support duplication over multiple merges at a time. One duplication can enable another opportunity for optimization after duplication, which requires another iteration of the DBDS algorithm. We only run another iteration if the cumulative benefit of the previous one is above a certain threshold. This only applies for about 20% of all compilation units. Additionally, subsequent iterations of DBDS will consider new merges first and only expand to already visited ones if there is sufficient budget left. Duplication can be stopped for two reasons:

- Every compilation unit is given a code size increase budget, which is currently set to 50%. If the resulting IR size (computed by size estimations not IR node count) reaches this budget, we stop duplicating.
- There is an upper bound for the size of a compilation unit defined by the VM because of internal restrictions. If this bound is reached, we stop duplicating.

We explain the details of our trade-off heuristic in Section 5.4.

Applicability Checks in Graal Graal already supported a limited form of ACs through the *canonicalizable* interface, which implements simple optimizations like constant folding and strength reduction as operations on IR nodes. We extended those ACs with the ones presented in Section 2.

5.3 Node Cost Model

In order to be able to annotate IR nodes with meta information for the static performance estimation we came up with an *IR node cost model* that models abstract performance and code size costs for each IR node.

We express the meta information as Java Annotations[1] per node class with enumeration values for the estimated *cycles* and *code size*. The costs are designed to be platform agnostic. Listing 7 shows the `AbstractNewObjectNode` class with an annotation specifying the cycles and code size of this IR node. We implemented the node cost annotations for over 400 different nodes in the Graal compiler. Based on these annotations, we can compute the *costs* and *benefits* of (sub)graphs. Additionally, we use probability information at each control-flow split to determine relative basic block execution frequencies. The probability information is generated from HotSpot's [26] profiling information [32]. We use a basic block's execution frequency relative to the maximum frequency of a compilation unit to scale the benefit of a duplication candidate (see Section 5.4).

Figure 4 shows some IR *before* and *after* duplication together with the respective benefit computations. In this example the constant folding after duplication reduced the run-time of the program by 1.8 cycles.

5.4 Budget-based Duplication Heuristic

The most important part of our algorithm is the trade-off tier as it decides what to duplicate. We developed a function `shouldDuplicate`$(b_{pi}, b_m, benefit, cost)$ that decides for one duplication at a time, if it should be performed. During the development of the DBDS algorithm, 3 factors turned out to be most important for the trade-off decision:

1. *Compilation Unit Maximum Size:* HotSpot limits us on the size of the installed code per compilation unit.[2] Therefore, we cannot increase code size arbitrarily.
2. *Code Size Increase Budget:* In theory, code size increase is only limited by the number of duplication opportunities. However, we do not want to increase the code size beyond a certain threshold. More code always increases the workload for subsequent compiler phases.
3. *Probability:* We decided to use profiling information to guide duplication. We compute the product of the relative probability of an instruction with respect to the entire compilation unit and the estimated cycles saved. We sort duplication candidates based on these values and optimize the most likely and most beneficial ones first.

[1] https://docs.oracle.com/javase/tutorial/java/annotations/declaring.html
[2] This is configurable with the option `-XX:JVMCINMethodSizeLimit`, which is 655360 bytes by default.

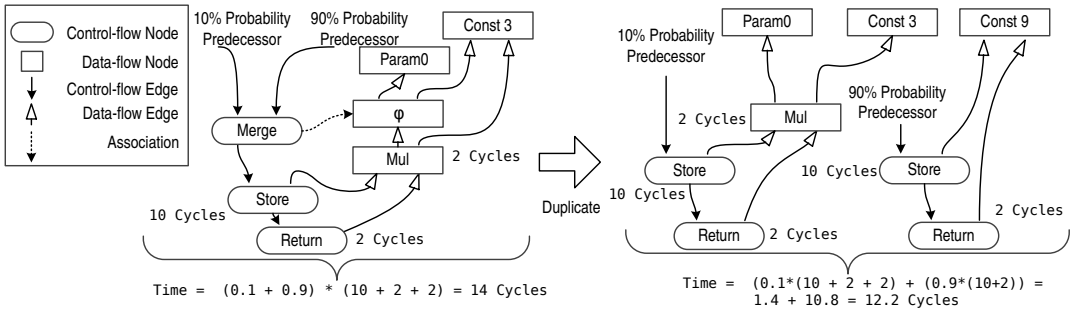


Figure 4. Node cost model example.

Based on the above observations we derived the following trade-off heuristic:

$$
\begin{aligned}
c &\ldots \text{Cost}\\
b &\ldots \text{Benefit}\\
p &\ldots b_{pi}\ \text{Probability}\\
cs &\ldots \text{Compilation Unit Size}\\
is &\ldots \text{Compilation Unit Initial Size}\\
IB &\ldots \text{Code Size Increase Budget} = 1.5\\
MS &\ldots \text{Max Compilation Unit Size}\\
BS &\ldots \text{Benefit Scale Factor} = 256\\
(b \times p \times BS) > c \wedge (cs < MS) \wedge (cs + c < is \times IB)& \qquad \text{(shouldDuplicate)}
\end{aligned}
$$

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. It turned out to generate the best results for the benchmarks presented in Section 6. The budget increase (currently set to 1.5 for 150%) represents the maximum code size increase.

6 Evaluation

We evaluated our implementation of the DBDS algorithm on top of the GraalVM,[3] by running and analyzing a number of industry-standard benchmarks.

6.1 Environment

All benchmarks were executed on a cluster of Sun X3-2 servers, all equipped with two Intel *Sandy Bridge* Xeon E5-2660 CPUs running at a clock speed of $2.2GHz$. Each CPU features 8 cores. The entire package is equipped with $256GB$ DDR3-1600 memory. GraalVM performs compilation in background threads concurrently to the interpreter executing the program. We ran each benchmark with three different configurations: `baseline` (DBDS disabled), `DBDS` (DBDS enabled) and `dupalot` (DBDS enabled but without cost/benefit trade-off). The `dupalot` configuration uses the simulation tier to find opportunities for duplications and performs them as long as there is any benefit, without caring about costs. We now give a short description of the benchmark suites.

Java Dacapo [3] is a well-known benchmark suite for Java containing 14 benchmarks[4] testing JIT compilation-, garbage collection-, file I/O- and networking performance.

Scala DaCapo [29] is the complementary DaCapo benchmark suite for the Scala[5] programming language, which compiles to JVM bytecode [21]. Scala workloads typically differ from Java workloads (as described by Stadler et al. [30]) in their type and class hierarchy behavior.

Micro Benchmarks: We use a micro-benchmark suite consisting of Java and Scala micro benchmarks addressing novel JVM features (since Java 8) like streams and lambdas.

JavaScript Octane [6] is a widely used JavaScript benchmark suite containing workloads ranging from 500 LOC to 300 000 LOC. The suite addresses JIT compilation, garbage collection, code loading and parsing of the executing JS VM. We measured Octane performance using Graal JS [34], the JavaScript implementation on top of Truffle, which is competitive in performance to Google's V8 [16]. Graal JS is on average 17% slower compared to the V8 JavaScript VM [34]. Truffle [34, 35] is a self-optimizing AST interpreter framework built on top of Graal. It uses partial evaluation [15] to produce compilation units for AST functions. Those compilation units are compiled by Graal to produce machine code.

We measured three different metrics in our experiments: *Peak performance*, *code size* and *compile time*.

Peak performance is reported by each benchmark suite itself. Java DaCapo, Scala Dacapo and the micro benchmarks report performance in milliseconds per iteration. JS Octane measures throughput performance and reports a throughput score after execution.

Compile time is measured programmatically in the compiler itself. Graal supports timing statements that are used throughout the compiler.

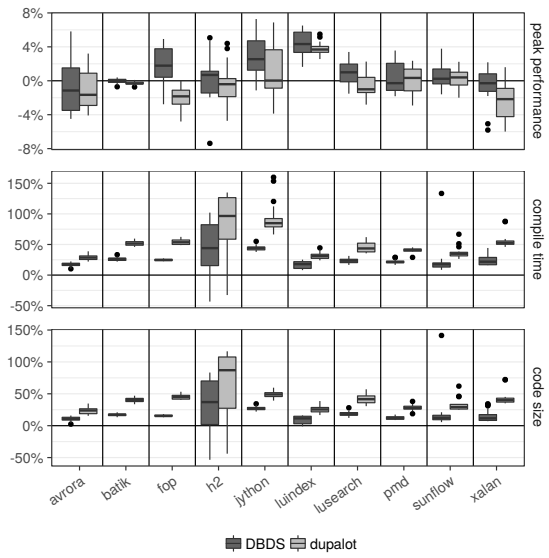
[3] Version 0.26 https://graalvm.github.io/

[4] We excluded the benchmarks *eclipse*, *tomcat*, *tradebeans* and *tradesoap* as they are no longer supported by JDK versions > $8u92$.

[5] https://www.scala-lang.org/

Code size is measured with a counter that tracks machine code size after code installation and constant patching.

For each configuration we executed the entire benchmark suite 10 times. We excluded the warm-up of each benchmark by only measuring performance after the point when compilation frequency stabilizes. We took the arithmetic mean of the last iterations after warm-up for each execution. For the visualization, we normalized the DBDS and the `dupalot` configuration to the *duplication-disabled* `baseline`.



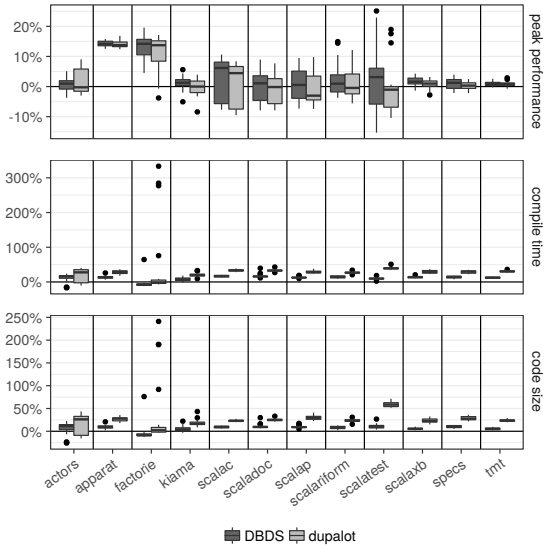
Configuration	Geometric Mean peak performance	compile time	code size
DBDS	0.99 %	24.92 %	15.90 %
dupalot	−0.14 %	50.08 %	38.22 %

Figure 5. Duplication Java DaCapo; peak performance (higher is better), compile time (lower is better), code size (lower is better).

6.2 Results

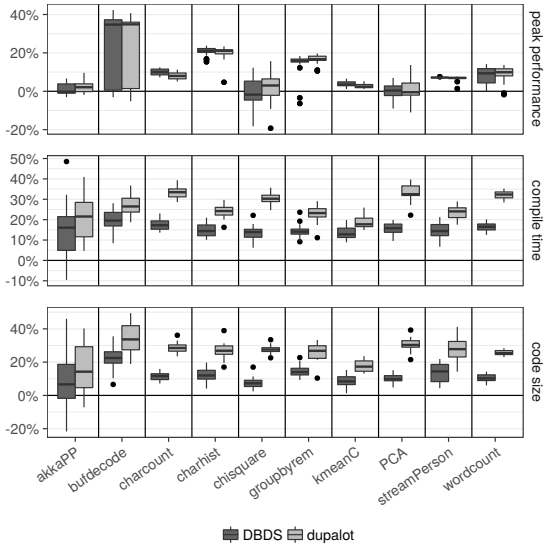
The results of our experiments can be seen in the boxplots [14] in Figures 5 to 8. The configuration of interest is the DBDS one, which implements the presented DBDS algorithm. The `dupalot` configuration allows us to prove that the proposed *trade-off tier* used by the DBDS configuration indeed reduces code size and compile time compared to the `dupalot` configuration. Ideally, the DBDS configuration performs significantly better in terms of code size and compile time than the `dupalot` configuration while producing the same peak performance improvements.

Peak Performance: The peak performance impact of the DBDS optimization varies over the different benchmark suites. In the mean the DBDS configuration increases peak performance by 5.89%. The Octane suite and the micro benchmarks show the highest peak performance increases with increases up to 40%, whereas benchmark suites such as Java DaCapo benefit less from duplication.



Configuration	Geometric Mean peak performance	compile time	code size
DBDS	3.15 %	11.32 %	6.88 %
dupalot	2.07 %	28.40 %	26.27 %

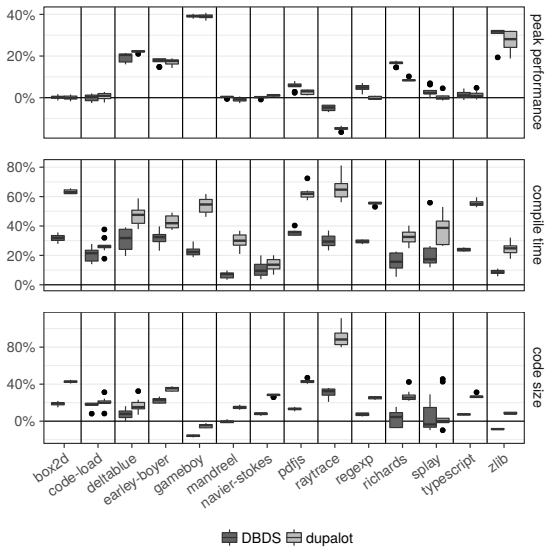
Figure 6. Duplication Scala DaCapo; peak performance (higher is better), compile time (lower is better), code size (lower is better).



Configuration	Geometric Mean peak performance	compile time	code size
DBDS	8.07 %	15.38 %	11.53 %
dupalot	8.57 %	26.41 %	25.78 %

Figure 7. Java/Scala Micro Benchmarks; peak performance (higher is better), compile time (lower is better), code size (lower is better).

Java Dacapo shows 3% improvements on *jython* and 4% on *luindex*. DBDS also performs considerably well on the Scala Dacapo workloads with improvements of up to 15%.



Configuration	Geometric Mean peak performance	compile time	code size
DBDS	8.81 %	22.48 %	7.31 %
dupalot	6.66 %	42.63 %	25.58 %

Figure 8. Duplication Graal JS Octane Benchmarks; peak performance (higher is better), compile time (lower is better), code size (lower is better).

The micro benchmark suite illustrates how the dynamic features of Scala and Java 8 create opportunities for duplication optimizations. Elimination of redundant type checks and opportunities for escape analysis are the reason for most peak performance increases that range from 5–40%.

Code Size & Compile Time: Compile time increases are very stable across all benchmarks ranging between 11–30% with a mean of 18.44% for the DBDS configuration. For the presented benchmarks, we can see that not performing all duplication opportunities always results in less code and shorter compilation times.

DBDS vs. dupalot: If we compare the compile times and the code size of the DBDS and dupalot configuration, DBDS is always better then the dupalot configuration (sometimes by a factor of 2), but achieves similar peak performance. For some benchmarks the DBDS configuration produces even faster machine code than the dupalot configuration. There are some benchmarks where the dupalot configuration is slightly faster than the DBDS configuration: e.g. the *akkaPP* benchmark from the micro benchmark suite. However, the compile time and code size metrics show that the dupalot configuration performs significantly worse than the DBDS configuration. Those benchmarks where the dupalot configuration generates faster code than the DBDS configuration are target for further optimizations and investigation.

The presented benchmarks show that duplicating code for every possible opportunity is not necessarily a good idea. There are even benchmarks where performing all duplication opportunities reduces peak performance: E.g. the *octane raytrace* benchmark is 15% slower in the dupalot configuration than in the baseline.

7 Related Work

Replication was proposed by Mueller and Whalley [23, 24] to optimize away *conditional* and *unconditional* branches. They perform duplication with the purpose of removing conditional and unconditional branches. Their approach is related to DBDS in that DBDS also duplicates code to remove conditional branches (see conditional elimination opportunity in Section 2).

Splitting [7] is an approach developed in the context of the *Self* compiler to tackle the problems of virtual dispatch [18]. *Splitting* is related to DBDS in many aspects. The self compiler performs splitting to specialize code after control-flow merges to the values used in predecessor branches. Chambers [7] describes Self's splitting heuristics which are based on the frequency of the code paths that are optimized (weight) and the cost of a duplication (code size). We extended their ideas (*reluctant splitting, eager splitting and the combination of both*) by using a fast duplication simulation algorithm in order to estimate the peak performance impact of the duplication before doing it. Additionally, we improved upon their idea of weight and cost by using a static performance estimator, to estimate the peak performance increase of a duplication transformation, and real profiling information from the VM to only optimize those parts of the program that are often executed.

Advanced Scheduling approaches like *Trace-*, *Superblock-* and *Hyperblock*-scheduling [13, 17, 22] apply code duplication in order to increase the instruction level parallelism (ILP) of a compilation unit. Those approaches differ from DBDS in what they want to achieve. They apply tail duplication [8] in order to extended basic blocks by forming *traces/-superblocks/hyperblocks*. Those structures increase ILP [33], which is needed to properly optimize code for VLIW processors which require elaborate compiler support to generate efficient instructions. Depending on the target structure (trace, superblock, hyperblock) the employed heuristics differ, but in general the optimization potential established by duplication is not analyzed in advance, as it is not needed for instruction selection and scheduling.

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. Those dependency sets collect all expressions influencing the computation of a variable. The effect of an optimization can then be estimated by propagating constant parameters through the dependent computations. The approach could be adapted to work on φ inputs instead of parameters to estimate the impact of φ inputs on subsequent code. DBDS

improves upon the approach by enabling a larger set of optimizations including control-flow dependent ones.

Bodík et al. [4] perform complete partial redundancy elimination (PRE) using duplication. They use duplication as a tool to enable PRE and also to compute in advance which duplications are necessary. We improved their ideas by computing a general set of optimizations that are enabled by duplication.

8 Conclusion and Future Work

In this paper we presented a novel approach that allows a compiler to decide which duplications should be performed in order to improve peak performance. The approach is based on an algorithm that combines simulation, trade-off and optimization into a three-tier process that enables optimizations after duplication. The simulation tier finds beneficial duplication candidates, which are then weighed up against each other in the trade-off tier, which tries to maximize peak performance while minimizing code size and compilation time.

We implemented our algorithm in the Graal compiler and showed that the approach can improve peak performance by up to 40% (with a mean of 5.89%) at a mean compilation time increase of 18.44% and code size increase of 9.93%.

In the future we want to tune our optimization tier. The current optimization tier implementation cannot duplicate over multiple merges along paths although the simulation tier can simulate along paths. We want to conduct experiments evaluating how complex a path-based implementation would be, and if we can increase peak performance even further.

Finally, we plan to validate the presented IR performance estimator. We are planning to conduct experiments validating a correlation between our benefit and cost estimations and the real performance and code size of an application.

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), 345–420. DOI: http://dx.doi.org/10.1145/197405.197406

[2] J. Eugene Ball. 1979. Predicting the Effects of Optimization on a Procedure Body. *SIGPLAN Not.* 14, 8 (Aug. 1979), 214–220. DOI: http://dx.doi.org/10.1145/872732.806972

[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 *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 169–190. DOI: http://dx.doi.org/10.1145/1167473.1167488

[4] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. 1998. Complete Removal of Redundant Expressions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 1–14. DOI: http://dx.doi.org/10.1145/277650.277653

[5] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. 1997. Value numbering. *Software-Practice and Experience* 27, 6 (1997), 701–724.

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

[7] Craig David Chambers. 1992. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-oriented Programming Languages*. Ph.D. Dissertation. Stanford, CA, USA. UMI Order No. GAX92-21602.

[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, 12 (Dec. 1991), 1301–1321. DOI: http://dx.doi.org/10.1002/spe.4380211204

[9] Cliff Click and Keith D. Cooper. 1995. Combining Analyses, Combining Optimizations. *ACM Trans. Program. Lang. Syst.* 17, 2 (March 1995), 16. http://doi.acm.org/10.1145/201059.201061

[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, 4 (Oct. 1991), 40. DOI: http://dx.doi.org/10.1145/115372.115320

[11] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*.

[12] 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 *Proceedings of the ACM Workshop on Virtual Machines and Intermediate Languages*. DOI: http://dx.doi.org/10.1145/2542142.2542143

[13] Joseph A. Fisher. 1995. Instruction-level Parallel Processors. IEEE Computer Society Press, Chapter Trace Scheduling: A Technique for Global Microcode Compaction, 186–198. http://dl.acm.org/citation.cfm?id=201749.201766

[14] Michael Frigge, David C. Hoaglin, and Boris Iglewicz. 1989. Some Implementations of the Boxplot. *The American Statistician* 43, 1 (1989), 50–54. http://www.jstor.org/stable/2685173

[15] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (01 Dec 1999), 381–391. DOI: http://dx.doi.org/10.1023/A:1010095604496

[16] Google. 2012. V8 JavaScript Engine. (2012). http://code.google.com/p/v8/

[17] 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, 234–253. http://dl.acm.org/citation.cfm?id=201749.201774

[18] 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 *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM, 294–310. DOI: http://dx.doi.org/10.1145/353171.353191

[19] David Leopoldseder, Lukas Stadler, Christian Wimmer, and Hanspeter Mössenböck. 2015. Java-to-JavaScript Translation via Structured Control Flow Reconstruction of Compiler IR. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS 2015)*. ACM, New York, NY, USA, 91–103. DOI: http://dx.doi.org/10.1145/2816707.2816715

[20] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. 1999. Performance Estimation of Embedded Software with Instruction Cache Modeling. *ACM Trans. Des. Autom. Electron. Syst.* 4, 3 (1999), 257–279. DOI: http://dx.doi.org/10.1145/315773.315778

[21] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2015. *The Java Virtual Machine Specification, Java SE 8 Edition*. http://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf

[22] 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, 161–170. http://dl.acm.org/citation.cfm?id=201749.201763

[23] Frank Mueller and David B. Whalley. 1992. Avoiding Unconditional Jumps by Code Replication. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 322–330. DOI: http://dx.doi.org/10.1145/143095.143144

[24] Frank Mueller and David B. Whalley. 1995. Avoiding Conditional Branches by Code Replication. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 56–66. DOI: http://dx.doi.org/10.1145/207110.207116

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

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

[27] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot™ Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*. USENIX, 1–12.

[28] Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. 2017. Making Collection Operations Optimal with Aggressive JIT Compilation. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala (SCALA 2017)*. ACM, New York, NY, USA, 29–40. DOI: http://dx.doi.org/10.1145/3136000.3136002

[29] 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 *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 657–676.

[30] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. 2013. An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance. In *Proceedings of the 4th Workshop on Scala (SCALA '13)*. ACM, New York, NY, USA, Article 9, 8 pages. DOI: http://dx.doi.org/10.1145/2489837.2489846

[31] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of the International Symposium on Code Generation and Optimization*. ACM Press, 165–174. DOI: http://dx.doi.org/10.1145/2544137.2544157

[32] April W. Wade, Prasad A. Kulkarni, and Michael R. Jantz. 2017. AOT vs. JIT: Impact of Profile Data on Code Quality. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2017)*. ACM, New York, NY, USA, 1–10. DOI: http://dx.doi.org/10.1145/3078633.3081037

[33] David W. Wall. 1991. Limits of Instruction-level Parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*. ACM, New York, NY, USA, 176–188. http://doi.acm.org/10.1145/106972.106991

[34] 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 *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 662–676. DOI: http://dx.doi.org/10.1145/3062341.3062381

[35] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. In *Proceedings of the 8th symposium on Dynamic languages (DLS '12)*. ACM Press, 73–82.