Fast-Path Loop Unrolling of Non-Counted Loops to Enable Subsequent Compiler Optimizations[*]

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

Roland Schatz
Oracle Labs
Linz, Austria
roland.schatz@oracle.com

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

Manuel Rigger
Johannes Kepler University Linz
Austria
manuel.rigger@jku.at

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

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

ABSTRACT

Java programs can contain non-counted loops, that is, loops for which the iteration count can neither be determined at compile time nor at run time. State-of-the-art compilers do not aggressively optimize them, since unrolling non-counted loops often involves duplicating also a loop's exit condition, which thus only improves run-time performance if subsequent compiler optimizations can optimize the unrolled code.

This paper presents an unrolling approach for non-counted loops that uses simulation at run time to determine whether unrolling such loops enables subsequent compiler optimizations. Simulating loop unrolling allows the compiler to determine performance and code size effects for each potential transformation prior to performing it.

We implemented our approach on top of the GraalVM, a high-performance virtual machine for Java, and evaluated it with a set of Java and JavaScript benchmarks in terms of peak performance, compilation time and code size increase. We show that our approach can improve performance by up to 150% while generating a median code size and compile-time increase of not more than 25%. Our results indicate that fast-path unrolling of non-counted loops can be used in practice to increase the performance of Java applications.

CCS CONCEPTS

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

KEYWORDS

Loop Unrolling, Non-Counted Loops, Loop-Carried Dependencies, Code Duplication, Compiler Optimizations, Just-In-Time Compilation, Virtual Machines

ACM Reference Format:
David Leopoldseder, Roland Schatz, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. 2018. Fast-Path Loop Unrolling of Non-Counted Loops to Enable Subsequent Compiler Optimizations. In *15th International Conference on Managed Languages & Runtimes (ManLang'18), September 12–14, 2018, Linz, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3237009.3237013

[*]This research project is partially funded by Oracle Labs.

1 INTRODUCTION

Generating fast machine code for loops depends on a selective application of different loop optimizations on the main optimizable parts of a loop: the loop's exit condition(s), the back edges and the loop body. All these parts must be optimized to generate optimal code for a loop. Therefore, a multitude of loop optimizations were proposed to reduce iteration counts, remove loop conditions [2], hoist invariant computations outside loop bodies [8], vectorize instructions [17], revert iteration spaces and schedule loop code to utilize pipelined architectures [1]. However, non-counted loops [10], that is, loops for which we cannot statically reason about induction variables and iteration count, are less optimizable than counted loops as their iteration count cannot be statically determined. Optimizing such loops could potentially lead to large performance gains, given that many programs contain non-counted loops.

As a generic way to optimize non-counted loops, we propose to unroll them without attempting to remove their exit conditions. However, this does not necessarily improve performance as the number of loop exit condition checks is not reduced and existing compiler optimizations often cannot optimize them away. Therefore, an approach is required that determines other optimization opportunities enabled by unrolling a non-counted loop. Such optimization opportunities have already been researched by Leopoldseder et al. [29] for code duplication of control-flow merges, showing significant increases in run-time performance if applied selectively. Loop unrolling is in fact a sequential code duplication transformation; the body of a loop is duplicated in front of itself. We developed an algorithm to duplicate the body of a non-counted loop, effectively unrolling the loop and enabling subsequent compiler optimizations. We developed a set of simulation-based unrolling strategies that analyze a loop for compiler optimizations enabled by loop unrolling. In summary, this paper contributes the following:

- We present an optimization, called *fast-path loop creation*, that can be used to unroll general loops. Fast-path loop creation is based on splitting loops with multiple back edges into hot-path and slow-path loops, enabling code motion to optimize may-aliasing memory accesses.

Benchmark	# Counted Loops	# Non-Counted Loops	% Non-Counted Loops
lusearch	219	449	67%
jython	1779	3333	64%
h2	538	4473	89%
pmd	1456	3379	69%
avrora	169	593	77%
luindex	364	660	64%
fop	653	1289	66%
xalan	510	1022	66%
batik	723	1857	71%
sunflow	249	389	59%
Arithmetic Mean			69.8%

Table 1: Number of counted and non-counted loops in the Java DaCapo Benchmark suite.

- We present an algorithm to partially unroll the hot path of a non-counted loop based upon the proposed fast-path loop creation.
- We present a set of simulation-based unrolling strategies to selectively apply unrolling of non-counted loops to improve peak performance.
- We implemented our approach in an SSA-based Java bytecode-to-native code just-in-time (JIT) compiler and evaluated the implementation in terms of performance, code size, and compile time using a set of Java, Scala, and JavaScript benchmarks.

2 UNROLLING NON-COUNTED LOOPS

In this section, we explain why unrolling non-counted loops is more challenging than unrolling counted loops. We present an initial idea to tackle this issue based on simulation, on which we will expand in the subsequent sections.

Non-counted loops are loops for which neither at compile time nor at run time the number of loop iterations can be determined. This can be due to one out of several reasons; the most common ones are shown in Listing 1: the loop exit conditions are not numerical (see Figure 1a), the loop body consists of side-effecting instructions[1] aliasing with the loop condition (see Figure 1b), or loop induction variable analysis [48] cannot guarantee that a loop contains induction variables leading to termination of the loop (see Figure 1c).

We informally observed that a significant amount of Java code contains non-counted loops. To further test this hypothesis, we instrumented the Graal [36] just-in-time (JIT) compiler to count the number of counted and non-counted loops [2] in the Java DaCapo benchmark suite. The results in Table 1 demonstrate that non-counted loops are more frequent than counted loops, which provides high incentives to compiler developers to optimize them.

```
while(mem[x]!=NUL){
 mem[y] = ...
}
```

Listing 2: Non-counted loop with side effects.

[1] The term *side-effect* describes operations that *can* have an observable effect on the execution system / environment. Such operations include function calls, memory writes, `sun.misc.unsafe` [34] usage, native calls and object locking [23].

[2] We counted the loops after inlining, because the number of loop iterations in counted loops is often only inferrable after this optimization.

Loop Unrolling [10–12, 14, 25, 41] non-counted loops [10] is only possible in a general way if a loop's exit conditions are unrolled together with the body of a loop. Previous work proposed avoiding to unroll non-counted loops together with their exit conditions like the one proposed by Huang and Leng [25] who used the weakest pre-condition calculus [13] to insert a loop body's weakest pre-condition into a given loop's initial condition to maintain a loops semantic after unrolling it. However, side-effects incur problems for optimizing compilers when trying to unroll non-counted loops. If the body of a loop contains side-effecting instructions, deriving the weakest pre-condition requires modeling the state of the virtual machine (VM), which we consider impractical for just-in-time compilation. As the compiler cannot generally infer the exact number of loop iterations for non-counted loops, it cannot speculatively unroll them and generate code containing side-effects that are unknown. This is a general problem, as side-effects and multi-threaded applications prevent compilers in managed execution systems like the Java virtual machine (JVM) from statically reasoning about the state of the memory, thus preventing the implementation of generic loop unrolling approaches. Consider the code in Listing 2. The loop's condition is the value in memory at index x. However, the body of the loop contains a memory write at index y. If alias analysis [15, 31] fails to prove that x and y do not alias, there is no way, without duplicating the exit conditions as well, to unroll the loop without violating the read after write dependency of the read in iteration n on the write in iteration $n-1$. Figure 2 shows the memory dependencies of the loop from Listing 2 in a graph illustrating memory reads and write over two loop iterations. Unrolling the loop requires the compiler to respect the memory constraints of the loop. That means that the condition cannot be re-ordered with the loop's body without violating the memory constraints of the loop. Therefore, a compiler cannot derive a weakest pre-condition with respect to the body of the loop as the post-condition of the loop body is unknown and the effect of the body of the loop on the state of the VM is unknown.

We propose to unroll non-counted loops with code duplication [29]. Duplicating the loop body together with its loop condition enables unrolling of general loops. We can represent every loop as a `while(true)` loop with the exit condition moved into the loop

```
while(true) {
 if(...) {
  body
 } else {
  break
 }
}
```

Listing 3: General Loop Construct.

body (see Listing 3). A loop can then by unrolled by duplicating the body of the loop and inserting it before the initial loop body. Listing 4 shows the example from Listing 3 after unrolling one iteration. When applied to the source example from Listing 2, the unrolled source loop looks like Listing 5. In contrast to general loop unrolling [2], we cannot remove the second exit path (lines 3, 5, 6) of the loop as it is non-counted. Unrolling loops via duplication does not create traditional unrolling optimization opportunities, because loops are unrolled with their loop exit conditions. Therefore,

```
while(mem[...]!=NUL){
  ...
}
```

(a) Non-Numerical Exit Condition.

```
while(mem[x]>a){
  ...
  // x and y might alias
  mem[y] = ...
  ...
}
```

(b) Loop Body Aliasing with Condition.

```
while(a>0){
  if(...){
    a++;
  }else{
    a-=b;
  }
}
```

(c) Incompatible Induction Variables.

Listing 1: Non-Counted Loops

```
while(true) {
  if(...) {
    body unrolled
    if(...) {
      original body
    } else {
      break
    }
  } else {
    break
  }
}
```

Listing 4: General Loop Construct unrolled.

```
while(mem[x]!=NUL){
  mem[y] = ...
  if(mem[x]!=NUL){
    mem[y] = ...
  }else{
    break;
  }
}
```

Listing 5: Side-effect-full Loop after Unrolling.

there is no reduction in execution time due to a reduced number of loop exit condition checks. We propose to tackle this issue with a simulation-based scheme enabling other optimizations via unrolling, based on the work of Leopoldseder et al. [29]. Loop unrolling can enable subsequent compiler optimizations in the same way as dominance-based duplication simulation [29]. This enables optimizations such as constant folding, strength reduction [2], conditional elimination [43], and so forth. To determine the effects of unrolling a loop on the optimization potential of the compilation unit, we simulate the unrolling prior to the actual transformation; therefore we can precisely find all profitable unrolling candidates and decide which ones are profitable enough to be unrolled.

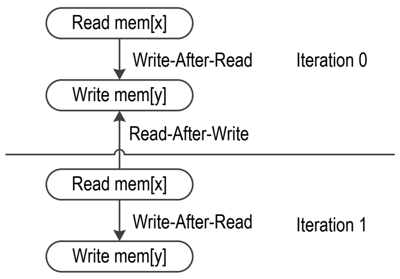


Figure 2: Side-effect Loop Memory Graph: The directed arrows show memory dependencies between read and write operations in two iterations of the loop.

3 FAST-PATH UNROLLING OF NON-COUNTED LOOPS

In this section, we present *Fast-Path Loop Unrolling*, our approach to simulation-based loop unrolling of non-counted loops. First, we propose a novel algorithm to perform loop unrolling via duplication and peeling, called *Fast-Path Loop Creation*, which simulates the effects of unrolling a loop and analyzes it for optimization opportunities after unrolling. We do this prior to the actual code transformation and unroll loops based on their optimization potential with respect to the optimizations implemented in the compiler. We use a static performance estimator [30] to estimate the entire run time of the fast-path of a loop and compare it with the version computed during simulation. If the optimization potential is sufficient, we perform the unrolling transformation (see Section 5).

3.1 Fast-Path Loop Creation

Before exploring the unrolling of non-counted loops, we started to experiment with a novel transformation that we call *fast-path loop creation*, an optimization for multi-back-edge loops. Multi-back-edge loops are very common in many applications; for example, they can be generated from conditional statements at the end of a loop's body or they can result from `continue` statements in loops. Additionally, a compiler often creates them as a result of inlining function calls into loops. We devised this optimization for loops with multiple back edges because they often do not have an equally-distributed probability for each control flow path leading to a back edge. If profiling information [47] indicates that a set of back edges is taken with a very high likelihood, the other back edges may hinder the (optimal) optimization of the loop due to two main reasons:

- A side-effect can influence the scheduling of anti-dependencies in the loop [20].
- A back-edge value flowing into a loop φ [9] can make a loop non-counted.

Consider the code in Listing 6. The loop invariant read [8] in line 4 cannot be hoisted outside of the loop as the memory location may be aliasing with the write in line 7. If alias analysis [15, 31] cannot prove that the loop variable i is different than k for all possible values of i and k, then the compiler has to assume that there could be a write-after-read dependency from the read to the write in one iteration. This results in a read-after-write dependency from iteration n to iteration $n + 1$. Considering that, for example, the `false` branch in the loop has a low probability in comparison to the `true` branch, the read has to be performed every iteration although the write is rarely taken. However, one possibility to still hoist the read out of the loop is to use *fast-path loop creation.*

The general transformation idea of fast-path loop creation is to create an outer, *slow-path* loop for all back edges of the loop that have a low probability. The inner, *fast-path* loop then only consists of frequently taken back edges. The outer loop created is a `while(true)` loop that is exited under the same conditions as the inner loop. Listing 7 shows the problematically-aliasing loop from Listing 6 after fast-path loop creation in pseudo code. Once we created the outer loop, we can hoist the read from the inner loop to the outer loop. We promoted the write-after-read dependency from the inner, fast-path, loop to the outer, slow-path, loop. The slow-path loop is only taken if the `else` branch in the inner loop is taken, therefore its probability is equal to the one of the `else` branch.

```
double result = 0;
for (int i = 0;i < arr.length; i++) {
 if (/*...*/) {
  result += arr[k];
 } else {
  // write that can alias
  arr[i] = result;
 }
}
```

Listing 6: Multi-back-edge Loop.

```
double result = 0;
int i = 0;
outer:while(true){ /*1*/
 // initial read
 tmp = arr[k];
 inner:for (;i < arr.length; i++) { /*2*/
  if (/*...*/) { /*3*/
   result += tmp;
   continue inner;
   /*3*/ } else { /*4*/
   // write that can alias
   arr[i] = result;
   // may aliasing write happened
   // continue outer, perform read again
   continue outer;
  /*4*/}
 /*2*/}
 break outer;
/*1*/}
```

Listing 7: Multi-back-edge Loop After Fast-Path Loop Creation.

Algorithm. Below, we present the algorithm for fast-path loop creation. To be consistent with the implementation section, we use a pseudo-graph-based intermediate representation (IR) based on our implementation platform's IR [16]. Graal IR is in *static-single-assignment* [9] (SSA) form. It is a superposition of two graphs, the data- and the control flow graph. Loop back edges and loop exits are modeled explicitly as instructions in the IR. The pseudo code for the algorithm can be seen in Algorithm 1. We create the outer loop header and re-route all loop ends of the inner loop that are considered to be part of the slow-path to the outer loop.

```
Data: Loop loop
Result: Loop slowPathLoop
LoopHeader fastPathLoopHeader ← loop.loopHeader();
/* Create new loop header for the outer loop and place it before
   the fast-path loop                                            */
LoopHeader slowPathLoopHeader ← new LoopHeader();
fastPathLoopHeader.insertInstructionBefore (slowPathLoopHeader);
/* Create phis for the outer loop based on the types of the inner
   loop.                                                         */
for PhiNode innerPhi in fastPathLoopHeader.phis () do
    PhiNode emptyPhi ← new PhiNode(innerPhi.type());
    slowPathLoopHeader.addPhi (emptyPhi);
end
/* All loop exit paths of inner loop will also exit outer loop */
for LoopExitNode exit in fastPathLoopHeader.exits () do
    LoopExitNode outerExit ← new LoopExitNode();
    slowPathLoopHeader.addLoopExit (exit);
    exit.insertInstructionAfter (outerExit);
end
/* Determine all fast-path ends, e.g. the n highest probable loop
   ends                                                          */
Set<LoopEndNodes> fastPathEnds ← computeFastPathEnds (loop);
   /* Update phis of inner and outer loop                        */
for LoopEndNode end in fastPathLoopHeader.ends () do
    if fastPathEnds.contains (end) then
        /* Fast-path back edges will jmp back to fast-path
           header                                                */
        continue;
    else
        end.setLoopHeader (slowPathLoopHeader);
        /* Add                                                   */
        int outerPhiIndex ← 0;
        for PhiNode phi in fastPathLoopHeader.phis () do
            slowPathLoopHeader.phis ().get (outerPhiIndex++).addInput
              (phi.removeInputAtLoopEnd (end));
        end
    end
end
/* update predecessor at [0] phi inputs                          */
int outerPhiIndex ← 0;
for PhiNode phi in fastPathLoopHeader.phis () do
    phi.replaceInputAt (0, slowPathLoopHeader.phis ().get
      (outerPhiIndex++));
end
return new Loop(slowPathLoopHeader);
```

Algorithm 1: Fast-Path Loop Creation Algorithm.

Figure 3 shows the IR of a generic loop during the transformation. First the compiler creates the outer loop header, which becomes the slow-path loop header. Then, the compiler re-routes the slow-path loop back edges to the slow-path (outer) loop header.

Discussion. Fast-path loop creation has the advantage that the compiler does not need to duplicate code in order to optimize may-aliasing memory accesses. Although code duplication can often improve performance, it inherently increases code size, which a compiler tries to minimize. Fast-path loop creation is a useful optimization on its own. Although we did not evaluate the approach in detail, we determined peak performance improvements of up to 50% for micro-benchmarks. However, in our work, we focused on the unrolling of non-counted loops, and used this optimization as a starting point for a fast-path loop unrolling algorithm described in Section 3.1.1.

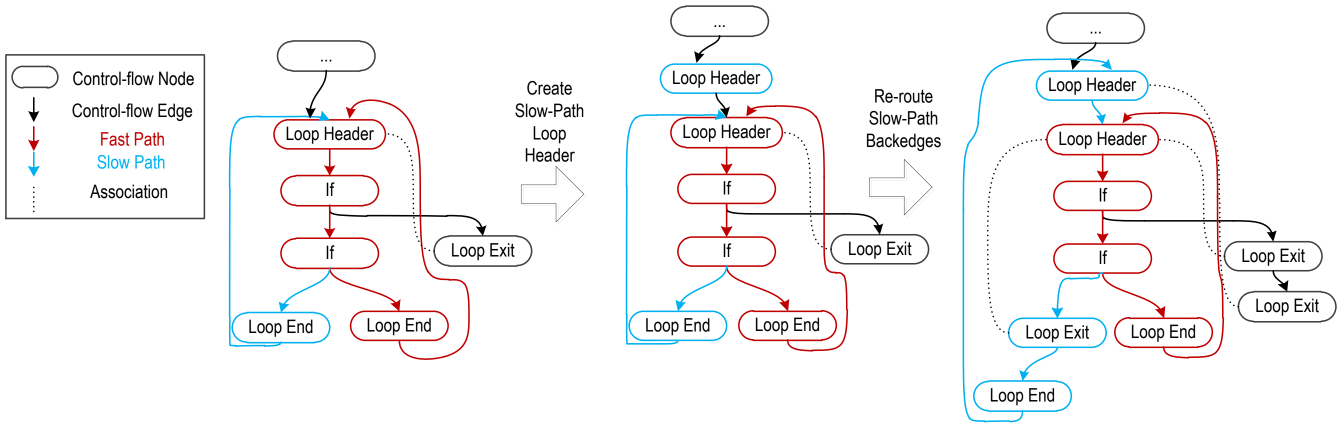


Figure 3: Fast-Path Loop Creation Example.

3.1.1 Non-counted Loop Unrolling via fast-path loop creation. We can extend the algorithm for fast-path loop creation to perform *fast-path* unrolling of non-counted loops. First, we identify those loops that should be unrolled. We then create the slow-path, outer loop for these loops.[3] After the slow-path loop is created, we use a loop peeling transformation [2] to duplicate the body of the inner (including its loop exit conditions), fast-path loop u times, where u denotes the number of desired unrollings. After the peeling step, we remove the inner, fast-path loop by removing inner loop exits and re-routing inner loop back edges to the outer loop. The loop φs of the inner loop are replaced with inputs coming from the single non-back-edge predecessor. In a final step, we remove the inner loop header. Figure 4 shows the example from Figure 3 after fast-path loop creation, during peeling and inner loop removal. Unrolling loops with the proposed technique has one major advantage over general, and partial unrolling of loops. As we re-route all back edges that we consider being part of the slow-path to the outer loop header, only the fast-path loop back edges remain connected to the inner, fast-path loop. Peeling the fast-path loop and removing it afterwards effectively unrolls *only* the fast-path of the original code and *not* its slow-path. Therefore, we keep the code size increase at a minimum, assuming the correctness of the profiling information.

We perform fast-path loop unrolling independently of the fast-path loop creation optimization. If a loop should be unrolled, we perform the unrolling transformation. Later in the compilation pipeline, we perform a dedicated analysis to detect optimizable patterns like the one from Listing 6, for which we create the fast-path loop independently of any prior unrollings. The fast-path loop creation optimization and the fast-path loop unrolling share IR transformations, but semantically they are two separate optimizations.

[3] This also works for loops with only one back edge. The inner loop temporarily then has no back edge and is degenerated until the transformation is finished.

3.2 Fast-Path Loop Unrolling Algorithm

We now explain the fast-path unrolling algorithm in detail. Our implementation platform Graal only performs a limited set of optimizations on non-counted loops[4]. We do not want to interfere with them, that is, we want to prevent creating IR patterns that are not optimizable for other transformations. Therefore, we ignore counted loops and loops for which the compiler can infer the iteration count after hoisting loop-invariant instructions out of the loop's body. We use the notion of unrolling *opportunities*. Opportunities model optimization-opportunities enabled by unrolling a loop.

We implemented a set of analysis algorithms determining optimization opportunities that can be enabled by loop unrolling (see Section 4). We follow the approach from Leopoldseder et al. [29], who proposed to separate a code duplication optimization into a simulation and transformation part. We use this scheme and group our unrolling approach into three major steps: identifying the optimization potential (1), deciding which loop should be unrolled (2) and then performing the final transformations (3). We first identify a loop's optimization potential after unrolling, by simulating the transformation, and then decide whether we want to unroll it. Algorithm 2 shows the non-counted loop unrolling optimization. First, we filter out all loops that are counted and therefore ignored by our approach. Then, we check a set of implemented opportunities on a loop. Each opportunity *op* performs a simulation of the loop after unrolling it (*op.shouldUnroll(loop)*) and analyzes the loop for optimizable patterns in unrolled iterations. If such an optimization is found, we use a static performance estimator to compute the overall run time of the loop's body and the overall run time of the loop's first (unrolled) iteration after unrolling. If the estimator predicts that the unrolled iteration has a lower run time than the original loop's body, we judge whether the expected performance increase justifies the expected code size increase (see Section 5). We derived the upper bound of unrolling iterations via empirical evaluation. We performed our experiments from Section 6 with an upper limit of 2 to 64 unrollings. Although sometimes special code patterns are sensitive to a high number of unrollings, in general we could not

[4] Graal performs loop unswitching and inversion on non-counted loops.

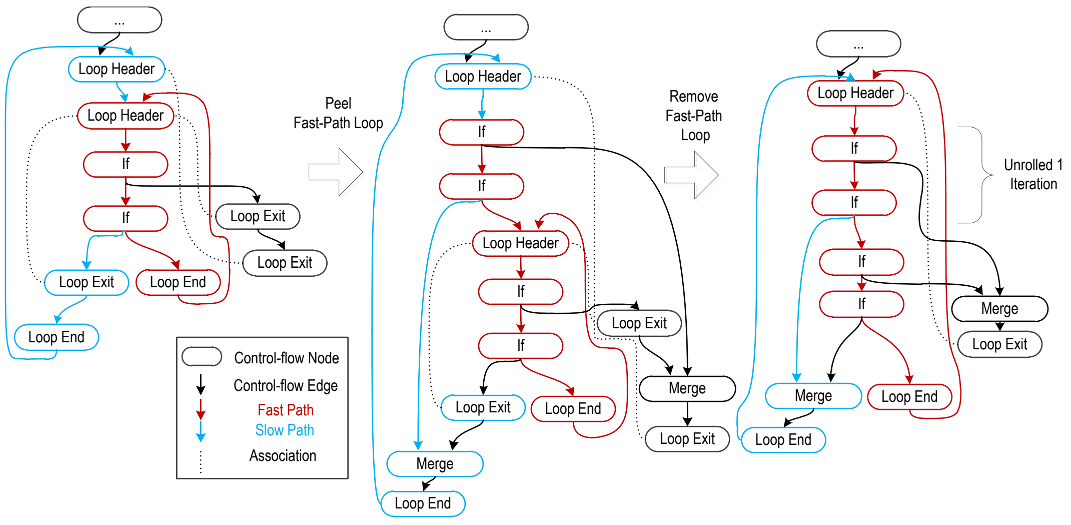


Figure 4: Path-Based Unrolling via Fast-Path Loop Creation and Peeling.

measure noticeable differences with an upper limit higher than 4. Thus, we unroll a loop by a maximum unrolling factor of 4. That means, we first create the fast-path loop (Section 3.1), and then peel the inner, slow-path loop at most four times. Finally, we remove the inner, fast-path loop. Only unrolling loops for which we know that

```
Data: ControlFlowGraph cfg
Result: Optimized ControlFlowGraph cfg
outer: for Loop loop in cfg.loops () do
    if isCounted (loop) then
        continue outer;
    for Opportunity op in UnrollingOpportunities do
        int unrollings ← op.shouldUnroll (loop);
        if unrollings > 0 then
            createFastPathLoop (loop);
            for i in 0 ... unrollings do
                peelIteration (loop);
            end
            removeFastPathLoop (loop);
    end
end
runCleanUpOptimizations (cfg);
```

Algorithm 2: Path-Based Unrolling Algorithm.

there is sufficient optimization potential after unrolling allows us to keep code size increase and compile time increase at a moderate level (See Section 6).

4 OPTIMIZATION OPPORTUNITIES AFTER UNROLLING

In the previous sections, we described our approach for unrolling the fast-path of a non-counted loop (see Section 3) based on splitting loops with multiple back edges into slow-path and fast-path loops. In this section we present some of the most important optimization opportunities that we consider for non-counted loop unrolling, namely safepoint poll reduction, canonicalization, and loop-carried dependencies.

Safepoint Poll Reduction. HotSpot's [24] execution system relies on *safepoints* [32] to perform operations that require all *mutator* threads to be in a well-defined state with respect to heap accesses. If the JVM requests a safepoint, it marks a dedicated memory page as non-readable. Mutators periodically poll this safepoint page and segfault in case the VM requested a safepoint. The JVM handles the segfault using a segfault handler registered for that purpose, which then invokes the safepoint handler that performs the desired operation. Typical safepoint operations are garbage collections [28], class redefinitions [51], lock unbiasing, and monitor deflation. As the safepoint protocol is collaborative, it forces the code generated by the JIT compilers to periodically poll the safepoint page (the interpreter polls it after every interpreted bytecode). Generated code typically performs these polls at specific program locations:

- Method returns: Safepoint polls are performed at every method return.
- Loop back edges: Safepoint polls are performed at every loop back edge.

Safepoint polls inside loops typically impose many problems on optimizing compilers. They require compilers to balance run-time performance and latency of safepoint operations. Optimizing compilers in the JVM aim to generate fast code; a safepoint poll in a hot loop is an additional memory read at every iteration, which leaves space for further optimization, for example, the safepoint poll can be optimized away if the iteration count of a loop is known and low enough to not have a negative effect on application latency. However, if a loop is long-running and the safepoint poll on the back edge is optimized away, the entire VM may be forced to wait for the loop to finish until the next safepoint is hit. In the worst case a full GC cannot be performed because a mutator loop does not poll the safepoint page on its back edge. Stalling a full GC can crash the VM as it can, for example, run out of memory. There are multiple solutions to this dilemma; for example, the Graal compiler currently removes safepoint polls on a loop's back edge if it can prove that the loop is counted and the loop's iteration count is in the range of a 32 bit integer. For non-counted loops and for loops in the range of 64 bit integers (Java `long`) Graal only removes safepoint polls on back edges if the path leading to the back edge is guaranteed to perform a safepoint poll, for example, via a method call.

Graal does not remove safepoint polls for non-counted loops, which leaves further optimization opportunities. Consider the code in Listing 8. The non-counted loop has two back edges, the `true` and the `false` branch. The compiler removes the safepoint poll on the `false` branch as there is a call inside which will poll on return. However, for the `true` branch the compiler does not remove the safepoint poll. Thus, if the `true` branch is the fast-path, one additional memory read is performed in every iteration of the loop. There are multiple solutions to optimize this safepoint poll without violating the implications of the safepoint protocol. The simplest solution is to unroll the loop u times, which will reduce the number of safepoint polls to n/u.

```
while (...;/*..condition..*/;...) {
 if (/*...*/) {
  // fast-path
  // no call
  ....
  safepointPoll;
 } else {
   // safepoint poll on call return
   call();
 }
}
```

Listing 8: Unrolling Opportunity: Safepoint Poll Reduction.

Canonicalization. [5] Instructions having loop φs [9] as inputs are potentially optimizable [29]. They can often be optimized by replacing their inputs with an input to the φ instead of the original φ. To optimize loop phis, we simulate the unrolling of one iteration by checking if an instruction is optimizable under the assumption that it has the back edge input of a loop φ as input instead of the φ. Listing 9 shows a simple loop that follows this pattern. The loop φ `loopPhi` has three inputs, the unknown value `a` on the forward predecessor, the constant 0 on the first back edge, and the unknown value `b` on the second back edge. In the body of the loop, we check `if(loopPhi == 0)`, which is generally not `true`. After unrolling the loop once, it has four back edges instead of three. One back edge was added by the unrolled loop body. Line 13 in Listing 10 was the back edge 1 in original loop's (Listing 9) body. However, peeling the fast-path of the loop in front of the original loop replaces the loop φ with the constant `0`, which was the φ's value on the original back edge 1. Therefore, the check `if(loopPhi == 0)` in the original iteration of the loop becomes `if(0 == 0)`. The `if` instruction can be eliminated and the call to `doSth` can be unconditionally executed.

Loop Carried Dependency. Loop carried dependencies [7, 46] are dependencies between different iterations of a loop. Typically, they appear in array access patterns inside loops. Consider the example loop in Listing 11. In every iteration of the loop, we read two array locations `a[i]` and `a[i+1]`. Unrolling one iteration of this non-counted loop allows us to avoid re-reading `a[i+1]` in the next iteration. In the first iteration, we read `a[0]` and `a[1]`. In the second iteration, we read `a[1]` and `a[2]`. If we unroll this loop once and there is no *aliasing* side-effect in the body of the loop, we can eliminate one redundant load.

We implemented a simulation-based unrolling analysis that estimates the effects of a loop unrolling. We combined it with a read

```
while (c)
 // loop phi: 3 inputs
 // forward predecessor: a
 // back edge 1: 0
 // back edge 2: b
 loopPhi = φ(a,0,b)
{
 if(loopPhi==0){
  doSth();
 }
 if(...){
  // back edge 1
  continue;
 }else{
  // back edge 2
  continue;
 }
}
```

Listing 9: Unrolling Opportunity: Canonicalization Loop.

```
1  while (c)
2   // loop phi: 4 inputs
3   // forward predecessor: a
4   // back edge 1: 0
5   // back edge 2: b
6   // back edge 3: b
7   loopPhi = φ(a,0,b,b)
8  {
9   if(loopPhi==0){
10   doSth();
11  }
12  if(...){
13   //original back edge 1: 0 replaces loopPhi
14   if(c){
15    if(0==0)
16     doSth();
17    }
18    if(...){
19     // back edge 1
20     continue;
21    }else{
22     // back edge 2
23     continue;
24    }
25   } else {
26    break;
27   }
28  }else{
29   // back edge 3
30   continue;
31  }
32 }
```

Listing 10: Unrolling Opportunity: Canonicalization Loop After Unrolling. Grey-shaded lines highlight the original iteration of the loop. The gray line with white text (line 15) shows the enabled optimization opportunity. The black line with white text (line 13) shows the replacement of a loop φ with the back edge value along the fast-path.

[5] We group different simpler optimizations like global value numbering [8], constant folding, strength reduction and conditional elimination [2, 43] under the term *canonicalization.*

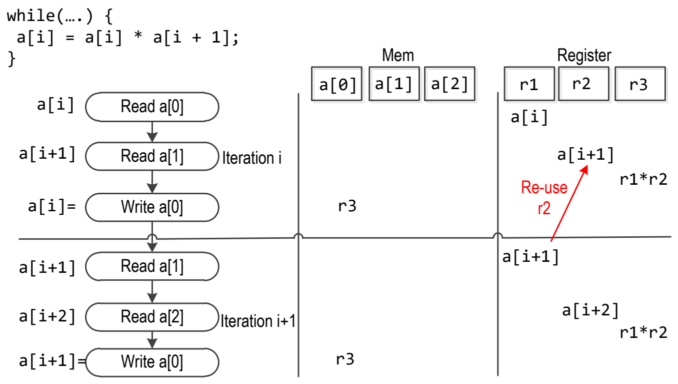


Figure 5: Loop-Carried Dependency Read Elimination Simulation.

elimination optimization to determine if there are loop-carried dependencies in the original loop body that can be optimized.

In order to find loop carried dependencies, we propose to compute a superblock [26][6] through the loop. For each memory location written and read in the superblock, the compiler can determine the associated instruction. We map memory reads to virtual registers and track their values over all instructions in the superblock. We first iterate over the superblock and record all memory reads and writes. Then, we iterate over all loop φ instructions of the loop and establish a mapping from loop φ values to their back edge values of the superblock[7]. Finally, we iterate over the superblock a second time replacing the φ node inputs in the loop header with the established back edge value mapping. For every instruction in the superblock, we try to eliminate memory accesses by re-using an already computed value. We repeat this process until there are no further eliminations possible. Figure 5 illustrates our read elimination simulation. We track the values read from memory in registers and update them (as well as the memory locations) according to the instruction seen. We then replace i with $i + 1$, which is the value of the φ on the back edge, and repeat the simulation for the loop body. If we find a read or a write that is redundant, we use a static performance estimator and record the estimated run-time cycles saved for this instruction. We then trade-off the cycles saved (via unrolling) with the overall size of the superblock and perform the unrolling if the benefit is sufficiently high (see Section 5 for details).

```
while (...) {
 a[i] = a[i] * a[i+1]
}
```

Listing 11: Unrolling Opportunity: Loop Carried Dependency Removal.

5 IMPLEMENTATION

We implemented the unrolling of non-counted loops in the Graal compiler [35, 36, 40], a dynamically-optimizing compiler for the HotSpotVM [24]. For evaluation, we used GraalVM [36], a HotSpot version in which Graal replaces C2 [38] as the top-tier compiler. Graal is a Java bytecode-to-machine-code just-in-time (JIT) compiler written in Java itself. Graal-generated code is comparable in performance to C2-generated code [40].

5.1 System Overview

Graal's compilation pipeline consists of a front end and a back end. The front end consists of bytecode parsing, creation of Graal IR [16], and high-level platform-independent optimizations such as inlining and partial escape analysis [44]. The platform-independent IR is then lowered to a platform-dependent one in the back end. On this low-level IR, additional optimizations and register allocation are performed.

5.2 Implementation Details

We implemented the unrolling optimization in the front end of the Graal compiler. The front end applies it iteratively, together with partial escape analysis [44] and scalar replacement, conditional elimination and read elimination. This way, optimization opportunities enabled by unrolling are immediately seized by the respective optimization.

5.2.1 Unrolling Heuristic. The final decision whether to unroll a loop is performed by a trade-off function, which takes several variables into account that are relevant for the final unrolling decision:

- Maximum compilation unit size: The virtual machine imposes an upper bound for code size per compilation unit.
- Initial graph size: The heuristic must be dynamically based on the initial size of the compilation unit.
- Cycles saved: The number of estimated cycles saved inside the loop body by unrolling it. This value is calculated during the simulation of the unrolling of the loop. The compiler estimates the overall cycles of the loop and the cycles of the loop after unrolling and computes the difference.
- Code Size Increase Budget: Prior work [29] in the Graal compiler derived constants for maximal code size increase of single optimizations. Therefore, all non-inlining based optimzizations inside the Graal compiler are limited in their code size increase by 50%. The reason for that is that code size increases also increase compile time as the workload for subsequent compiler optimizations is increased.
- Byte per cycle spending: In order to relate an estimated benefit with its cost we compute the ratio between code size and run-time improvement. Thus, we need to specify how much code size increase (in bytes) we are willing to accept for every saved cycle. This is currently configured to be 512 bytes per saved cycle. We derived this value via a structured experiment running our benchmarks from Section 6 with all powers of 2 between 2 and 1024. The value 512 generated the best results for those benchmarks.

[6] We include structured control-flow diamonds if profiling information indicates that a split's successor paths are taken with equal probability.
[7] The superblock ends in a loop back edge.

All operands are computed by the optimization opportunities (see Section 3). The final unrolling decision is done by the algorithm given by the pseudo code in Algorithm 3 and is based on the following trade-off heuristic:

l	. . .Loop
u	. . .Unrollings
$l.s$	. . .Loop Size
$l.cp$	. . .Cycles saved per loop iteration
$l.oc$	. . .Overall cycles loop including condition
cs	. . .Compilation Unit Size
is	. . .Compilation Unit Initial Size
IB	. . .Code Size Increase Budget = 1.5
MS	. . .Max Compilation Unit Size
MU	. . .Max Unrollings = 4
BS	. . .Bytes/Cycle Spending = 512

$$\texttt{canUnroll(loop)} \mapsto is * IB < MS \ \wedge \ cs + u * l.s < MS$$
$$\texttt{shouldUnroll(loop)} \mapsto l.cp * BS > l.s$$
$$\texttt{nrOfUnrollings(loop)} \mapsto u \equiv min(MU, l.cp/l.oc * MU * 10)$$

The compiler unrolls a loop `nrOfUnrollings` times if it has

```
/* Size Restriction */
if canUnroll (loop) then
    /* Trade-off Heuristic */
    if shouldUnroll (loop) then
        /* Compute Final Unrolling Factor */
        return nrOfUnrollings (loop);
return 0;
```

Algorithm 3: Unrolling Decision Algorithm.

sufficient optimization potential. It derives this by comparing the saved cycles per iteration with the size of the loop, multiplied by a constant factor that we can set to express how much code size we are willing to spend for one cycle run time reduction.

6 EVALUATION

In our evaluation we seek to determine whether fast-path loop unrolling of non-counted loops can improve the performance of Java applications. We evaluated our implementation on top of the GraalVM[8] by running and analyzing a set of industry-standard benchmarks.

6.1 Environment

All benchmarks were executed on an Intel© Core™ i7-6820HQ machine running at 2.7GHz, equipped with 32 GB DDR-4 RAM clocking at 2133 MHz. The CPU provides 8 hardware threads. We disabled turbo boost [9] for more stable results.

We evaluated our non-counted loop unrolling approach with three industry-standard benchmarks suites (Java DaCapo [3], Scala DaCapo [42], jetstream JavaScript benchmark [39] [10]) and a set of Java micro-benchmarks, that stress the usage of Java 8 language features such as streams and lambdas.

The Java and Scala benchmarks (Java DaCapo, Scala DaCapo, micro-benchmarks) can be executed directly on the GraalVM as they are compiled to JVM bytecode [33]. The jetstream JavaScript benchmark suite was executed on GraalJS [50], an ECMAscript [11] compliant JavaScript implementation on top of Truffle [53]. Truffle [49, 50, 52, 53] is a self-optimizing abstract syntax tree (AST) interpreter framework on top of the GraalVM. Truffle language implementations can utilize the Graal compiler to perform *Partial Evaluation* [19] of AST programs and reach near-native performance. GraalJS is on average 17% slower compared to the V8 JavaScript VM [21]. Truffle AST interpreters are themselves implemented in Java. During compilation, their ASTs are combined to one compilation unit via partial evaluation, which effectively inlines the logic for each AST operation into the root AST node. This is performed on the Java bytecode level, therefore dynamic languages executed on top of Truffle produce Java compilation units themselves. However, Truffle compilation units are typically larger and more dynamic than classical Java workloads, requiring more ellaborate compiler optimizations to reach near-native performance [29].

We tested 2 configurations: the `baseline`, without unrolling enabled, and a configuration pbu (for **p**ath-**b**ased **u**nrolling) with the optimization enabled. We accounted for warm-up of each benchmark by only measuring performance after the compilation frequency stabilized. For each benchmark we performed 20 out-of-process runs and computed the mean of the last iterations after warm-up. The number of warm-up and measurement iterations depend on the time needed by the compiler to stabilize. Therefore, we computed the number of measurement iterations based on the number of warm-up iterations. This varied for each benchmark ranging from 5 to 10 iterations. For plotting, we normalized the results of the pbu configuration to an arithmetic mean computed from the `baseline` for each benchmark. Some benchmarks in our experiments, especially DaCapo and ScalaDacapo, suffer from stability problems showing irregular outliers for all metrics. Some of those outliers can be partially attributed to HotSpot's multi-tiered execution system with an interpreter and two JIT compilers (C1 and Graal). The number of compilations done by the JIT compilers depends on profiling information gathered during interpreting the code, therefore code size and compile time metrics, which are accumulated over the entire run of a benchmark, are not stable across several out-of-process iterations. We also measured irregular outliers with the baseline configuration, however we used the baseline to calculate the mean for normalization. Note that our own micro-benchmarks and the asm.js jetstream benchmarks are mostly stable.

Our main hypothesis is that fast-path unrolling of non-counted loops can significantly increase the run-time performance of Java applications. Therefore, our primary metric of interest is performance (average run time or throughput). However, in order to validate that our optimization can be used in practice, we also measured code size and compile time.

[8]Version 0.32 https://graalvm.github.io/

[9]https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

[10] We used only the asm.js [22] benchmarks from the jetstream suite, which are C programs compiled to JavaScript via emscripten [54]. A large part of the jetstream suite is based on the retired Octane [6] benchmark suite. We measured fast-path loop unrolling performance on the octane suite but did not get any significant changes in performance thus we do not report Octane numbers in this paper.

[11]https://www.ecma-international.org/publications/standards/Ecma-262.htm

6.2 Results

Figure 6 shows the results of our experiments for the benchmarks (Figures 6a to 6d) as boxplots [18]. For each benchmark, we plotted compile time, code size and performance (runtime or throughput). We normalized the results to the `baseline` configuration, without unrolling enabled.

Performance. The performance impact of fast-path unrolling of non-counted loops varies over the different benchmark suites. The impact on Java DaCapo is mixed. We see slight improvements and regressions of up to 2% (e.g. xalan). The performance impact on the Scala DaCapo suite is similar. However, several benchmarks showed larger improvements (*factorie*) and regressions (*apparat*). The *factorie* benchmark offers much optimization potential for unrolling. It contains many loops with carried dependencies that can be optimized by our approach. *appart* reacts sensitive to the removal of safe-points as it has a small working set which suffers caching issues if garbgabe collection is delayed. Therefore our safepoint poll reduction can also have negative side-effects.

Performance improvements on the micro-benchmarks are significant with run-time reductions of up to 35%. The most significant performance improvements are reached on our subset of the jetstream benchmark suite with improvements of up to 150% (*jetstream:dry.c*). Our subset of the jetstream suite only contains asm.js benchmarks. Those benchmarks offer much optimization potential (mostly loop-carried dependencies), for example, array accesses that can be optimized if a loop is unrolled once.

Compile Time. We consider the compile-time increase to be acceptable for the benchmarks. We measured the highest compile-time increase in the *Scala DaCapo:factorie* and *Scala DaCapo:specs* benchmarks, with a median increase of about 25%. We analyzed the *factorie* benchmark in detail; several hot compilation units in this benchmark are very large. Unrolling them, although beneficial (see run-time reduction by 5%), creates additional code that slows down subsequent compiler phases. For this benchmark, the unrolling overhead is about 12%; however, the additional compile-time overhead for subsequent optimizations is also increased by about 10%, causing a general compile-time overhead of 25% for the entire benchmark.

Code Size. The code size impact of the entire optimization is negligible in nearly all cases. This is due to our trade-off function that is configured to only optimize loops for which we know that there is a sufficient optimization potential. Unrolling all loops that carry optimization potential would have a severe impact on code size. However, those loops for which the run-time impact justifies a certain code-size increase are much less frequent, thus the overall increase in code size is low. There are two outliers with a code size increase of up to 20%. They are the same outliers as for compile time: the *Scala DaCapo factorie* and *specs* benchmarks. We see that compile time increase and code size increase correlate on those benchmarks. The current parameterization of the optimization will never allow a code size increase that is larger than 50%. However, those two benchmarks can be optimized further. As part of our future work, we want to experiment with different parameters for the bytes-per-cycle ratio in our trade-off function to reduce the compile-time overhead.

Discussion. The performance increases shown in our experiments seems to confirm our initial hypothesis, namely that unrolling of non-counted loops is beneficial. We have shown that we can increase the run-time performance of Java programs by using simulation-based unrolling of non-counted loops.

It is important to note that Truffle AST interpreters themselves are programmed in Java. This means that after partial evaluation of a JavaScript AST (for e.g. the jetstream benchmarks) Graal still compiles Java code. However, partial evaluation often creates IR patterns that are more dynamic than statically typed Java code. Thus, the compiler has more potential for optimizations.

Since many benchmarks contain more non-counted loops than counted ones, the optimization potential for our fast-path loop unrolling approach would be high in theory (as shown in the initial experiment, see Section 2). However, in our implementation we only optimize loops for which we can determine one of the presented optimization opportunities.

7 RELATED WORK

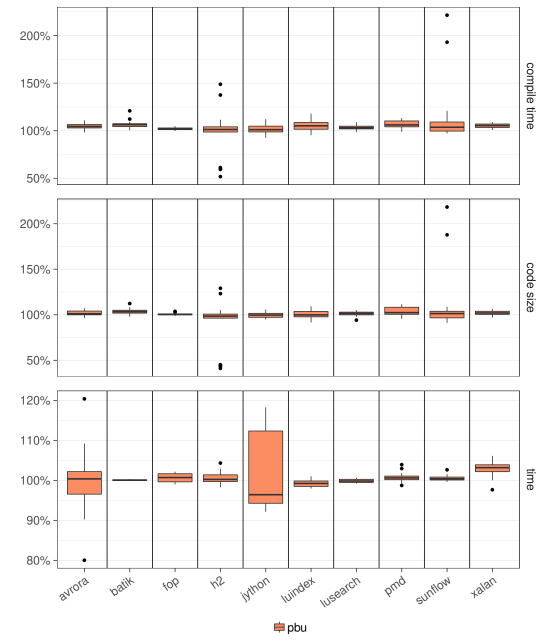
Our work builds upon the idea of simulation-based code duplication presented by Leopoldseder et al. [29]. We extended this work by simulating not only general code duplications at control flow merges but simulating the effects of unrolling entire non-counted loops.

Numerous approaches to generate faster code for loops have been proposed over the last decades. Most related to our approach is the work of Huang and Leng [25]. They proposed a generalized loop unrolling of `while(...)` loops, based on adding the *weakest precondition* [13] of a loop to its condition. We take a different approach than their work by selectively unrolling loops with arbitrary, side-effecting instructions together with their loop conditions. We do not use the concept of *weakest preconditions* as the automatic derivation of them is not possible in the presence of arbitrary side-effects. Therefore, we are searching for different optimization opportunities that can still be enabled via unrolling a loop even if its loop conditions are unrolled with the body.

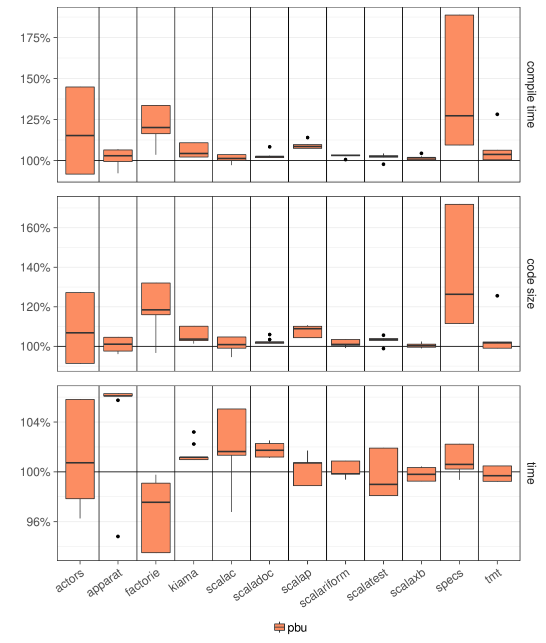
Loop unrolling and software pipelining [1] combined with loop jamming was researched by Carr et al. [5]. Their approach performs unroll-and-jam to increase the instruction-level parallelism of nested loops in order to generate better scheduling on pipelined architectures. Our approach is different in that we explicitly unroll non-counted loops to enable other compiler optimizations. We did not implement an instruction-level-parallelism-based optimization.

The HotSpot server compiler [38] applies different optimizations on counted and non-counted loops including unrolling, tiling and iteration splitting [37]. Graal implements similar loop optimizations as the server compiler. Our work extends Graal to also unroll non-counted loops.

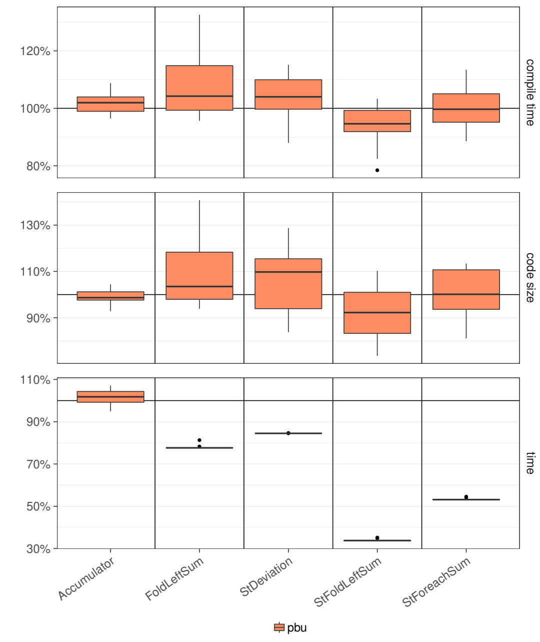
Carminati et al. [4] studied a loop unrolling approach to reduce worst-case execution time of real-time software. They use different unrolling strategies for counted and non-counted loops, and use if-conversion to create branch-less code. Our approach to non-counted loop unrolling allows to effectively unroll only the fast-path of a loop. Additionally, we improve upon their work by removing all restrictions (general side-effects) on the body of the target loop to be unrolled.



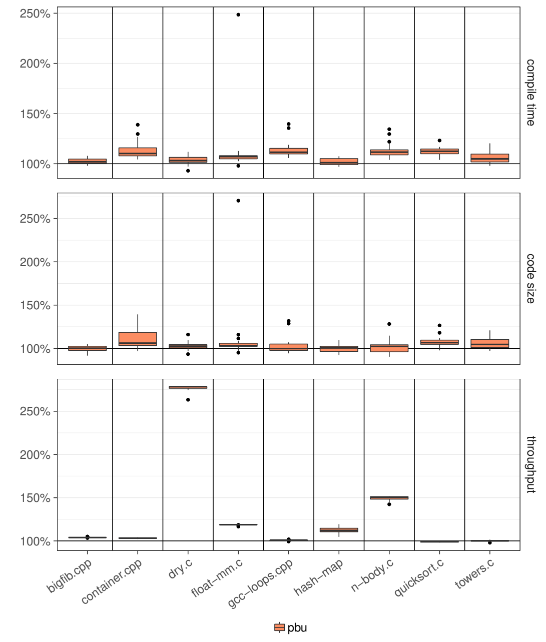
(a) Java DaCapo



(b) Scala DaCapo



(c) Micro Benchmarks



(d) JavaScript JetStream

Figure 6: Fast-Path Unrolling Performance relative to Baseline; compile time(lower is better), code size(lower is better), time, (lower is better), throughput (higher is better).

Krall and Lelait [27] proposed loop unrolling for SIMD-based auto-vectorization in an optimizing compiler. Counted loops are unrolled *n* times where *n* equals the vector length. A scalar-to-SIMD transformation then transforms scalar code to its vectorized equivalent. This work is partially related to ours in that they unroll to enable subsequent optimizations, in their case scalar-to-SIMD transformation. However, their work only considers counted loops.

8 CONCLUSION & FUTURE WORK

In this paper, we have presented a novel approach to unroll non-counted loops based on their optimization potential. We have found that a large number of Java programs contain non-counted loops. To optimize them, we have presented a novel approach to use simulation-based code duplication for the unrolling of non-counted loops. We implemented our approach on top of the GraalVM and showed in our evaluation that optimizing non-counted loops with simulation-based loop unrolling can significantly increase the run-time performance of Java programs, thus suggesting that modern Java compilers could consider implementing such optimizations.

As part of future work, we want to add more optimizations based on analyzing more non-counted loops in Java programs (e.g. SPECjvm2008 [45]). Additionally, we are currently working on our implementation to reduce compile time further.

REFERENCES

[1] Vicki H Allan, Reese B Jones, Randall M Lee, and Stephen J Allan. 1995. Software pipelining. *ACM Computing Surveys (CSUR)* 27, 3 (1995), 367–432.

[2] 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. https://doi.org/10.1145/197405.197406

[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. https://doi.org/10.1145/1167473.1167488

[4] Andreu Carminati, Renan Augusto Starke, and Rômulo Silva de Oliveira. 2017. Combining loop unrolling strategies and code predication to reduce the worst-case execution time of real-time software. *Applied Computing and Informatics* 13, 2 (2017), 184–193.

[5] Steve Carr, Chen Ding, and Philip Sweany. 1996. Improving Software Pipelining With Unroll-and-Jam. In *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on,*, Vol. 1. IEEE, 183–192.

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

[7] Alan E Charlesworth. 1981. An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family. *Computer* 14, 9 (1981), 18–27.

[8] Cliff Click. 1995. Global Code Motion/Global Value Numbering. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95).* ACM, New York, NY, USA, 246–257. https://doi.org/10.1145/207110.207154

[9] 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. https://doi.org/10.1145/115372.115320

[10] Jack W Davidson and Sanjay Jinturkar. 1995. *An aggressive approach to loop unrolling.* Technical Report. Technical Report CS-95-26, Department of Computer Science, University of Virginia, Charlottesville.

[11] Jack W Davidson and Sanjay Jinturkar. 1995. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. In *Microarchitecture, 1995., Proceedings of the 28th Annual International Symposium on.* IEEE, 125–132.

[12] Jack W Davidson and Sanjay Jinturkar. 1996. Aggressive loop unrolling in a retargetable, optimizing compiler. In *International Conference on Compiler Construction.* Springer, 59–73.

[13] Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Etats-Unis Informaticien, and Edsger Wybe Dijkstra. 1976. *A discipline of programming.* Vol. 1. prentice-hall Englewood Cliffs.

[14] Thomas J. Watson IBM Research Center. Research Division, FE Allen, and J Cocke. 1971. *A catalogue of optimizing transformations.*

[15] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. 1998. Type-based Alias Analysis. (1998), 106–117. https://doi.org/10.1145/277650.277670

[16] 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.*

[17] Sara El-Shobaky, Ahmed El-Mahdy, and Ahmed El-Nahas. 2009. Automatic Vectorization Using Dynamic Compilation and Tree Pattern Matching Technique in Jikes RVM. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems.* ACM, 63–69.

[18] 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

[19] 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. https://doi.org/10.1023/A:1010095604496

[20] Philip B. Gibbons and Steven S. Muchnick. 1986. Efficient Instruction Scheduling for a Pipelined Architecture. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction (SIGPLAN '86).* ACM, New York, NY, USA, 11–16. https://doi.org/10.1145/12276.13312

[21] Google. 2018. V8 JavaScript Engine. (2018). https://chromium.googlesource.com/v8/v8.git

[22] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 185–200.

[23] Peter Hofer, David Gnedt, Andreas Schörgenhumer, and Hanspeter Mössenböck. 2016. Efficient tracing and versatile analysis of lock contention in Java applications on the virtual machine level. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering.* ACM, 263–274.

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

[25] Jung-Chang Huang and Tau Leng. 1999. Generalized loop-unrolling: a method for program speedup. In *Application-Specific Systems and Software Engineering and Technology, 1999. ASSET'99. Proceedings. 1999 IEEE Symposium on.* IEEE, 244–248.

[26] 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

[27] Andreas Krall and Sylvain Lelait. 2000. Compilation techniques for multimedia processors. *International Journal of Parallel Programming* 28, 4 (2000), 347–361.

[28] Philipp Lengauer and Hanspeter Mössenböck. 2014. The taming of the shrew: increasing performance by automatic parameter tuning for java garbage collectors. In *Proceedings of the 5th ACM/SPEC international conference on Performance engineering.* ACM, 111–122.

[29] 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 the 2018 International Symposium on Code Generation and Optimization (CGO 2018).* ACM, New York, NY, USA, 126–137. https://doi.org/10.1145/3168811

[30] 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. https://doi.org/10.1145/315773.315778

[31] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. 2003. A Compiler Framework for Speculative Analysis and Optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03).* ACM, New York, NY, USA, 289–299. https://doi.org/10.1145/781131.781164

[32] Yi Lin, Kunshan Wang, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 2015. Stop and Go: Understanding Yieldpoint Behavior. (2015), 70–80. https://doi.org/10.1145/2754169.2754187

[33] 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

[34] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. 2015. Use at Your Own Risk: The Java Unsafe API in the Wild. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015).* ACM, New York, NY, USA, 695–710. https://doi.org/10.1145/2814270.2814313

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

[36] 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

[37] Oracle. 2015. Loop Optimizations in the Hotspot Server VM Compiler (C2). (2015). https://wiki.openjdk.java.net/pages/viewpage.action?pageId=20415918

[38] 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.

[39] Filip Pizlo. 2014. JetStream Benchmark Suite. (2014). http://browserbench.org/JetStream/

[40] 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. https://doi.org/10.1145/3136000.3136002

[41] Vivek Sarkar. 2000. Optimized unrolling of nested loops. In *Proceedings of the 14th international conference on Supercomputing*. ACM, 153–166.

[42] 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.

[43] 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. https://doi.org/10.1145/2489837.2489846

[44] 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. https://doi.org/10.1145/2544137.2544157

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

[46] Bogong Su and Jing Wang. 1991. Loop-carried dependence and the general URPR software pipelining approach (unrolling, pipelining and rerolling). In *System Sciences, 1991. Proceedings of the Twenty-Fourth Annual Hawaii International Conference on*, Vol. 2. IEEE, 366–372.

[47] 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. https://doi.org/10.1145/3078633.3081037

[48] Michael Wolfe. 1992. Beyond Induction Variables. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. ACM, New York, NY, USA, 162–174. https://doi.org/10.1145/143095.143131

[49] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 2014. An object storage model for the truffle language implementation framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*. ACM, 133–144.

[50] 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. https://doi.org/10.1145/3062341.3062381

[51] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. 2010. Dynamic code evolution for Java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*. ACM, 10–19.

[52] 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 *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, 187–204.

[53] 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.

[54] Alon Zakai. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 301–312.