Lazy Sparse Conditional Constant Propagation in the Sea of Nodes

Christoph Aigner
Johannes Kepler University
Linz, Austria
christoph.aigner@jku.at

Gergö Barany
Oracle Labs
Vienna, Austria
gergo.barany@oracle.com

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

Abstract

Conditional constant propagation is a compiler optimization that detects and propagates constant values for expressions in the input program taking unreachable branches into account. It uses a data flow analysis that traverses the program's control flow graph to discover instructions that produce constant values.

In this paper we document our work to adapt conditional constant propagation to the Sea of Nodes program representation of GraalVM. In the Sea of Nodes, the program is represented as a graph in which most nodes 'float' and are only restricted by data flow edges. Classical data flow analysis is not possible in this setting because most operations are not ordered and not assigned to basic blocks.

We present a novel approach to data flow analysis optimized for the Sea of Nodes. The analysis starts from known constant nodes in the graph and propagates information directly along data flow edges. Most nodes in the graph can never contribute new constants and are therefore never visited, a property we call lazy iteration. Dependences on control flow are taken into account by evaluating SSA ϕ nodes in a particular order according to a carefully defined priority metric.

Our analysis is implemented in the GraalVM compiler. Experiments on the Renaissance benchmark suite show that lazy iteration only visits 20.5 % of all nodes in the graph. With the constants and unreachable branches found by our analysis, and previously undetected by the GraalVM compiler, we achieve an average speedup of 1.4 % over GraalVM's optimized baseline.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; *Dynamic compilers*; Correctness.

MPLR '24, September 19, 2024, Vienna, Austria

ACM ISBN 979-8-4007-1118-3/24/09
https://doi.org/10.1145/3679007.3685059

Keywords: data flow analysis, constant propagation, compilers, optimization, Sea of Nodes

ACM Reference Format:
Christoph Aigner, Gergö Barany, and Hanspeter Mössenböck. 2024. Lazy Sparse Conditional Constant Propagation in the Sea of Nodes. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3679007.3685059

1 Introduction

Constant Propagation is a compiler optimization that tries to perform as many calculations as possible at compile time, minimizing execution time by not needing to calculate those values at run time. Because detecting every compile time constant is generally an undecidable problem [5], the best one can do is to employ an algorithm with reasonable time complexity that, although not finding every possible constant, finds most constants and does so without reporting non-constant values as constant.

The state of the art in constant propagation is Wegman and Zadeck's *Sparse Conditional Constant* (SCC) algorithm [7]. It uses a data flow analysis on a program represented as a control flow graph (CFG) consisting of basic blocks of instructions in Static Single Assignment (SSA) form [3]. The analysis is *sparse* as it exploits SSA form to associate constants found with SSA variables, as opposed to earlier non-SSA algorithms which associated analysis results with pairs of variables and program points. SCC is *conditional* in that the found constants are also taken into account when evaluating branch conditions and to mark unreachable program paths which do not need to be analyzed. Skipping unreachable paths, in turn, allows finding more constants, so that this composition of simple constant propagation and unreachable code elimination is more powerful than arbitrary iterations of the separate analyses [1].

In this work we adapt sparse conditional constant propagation to the Sea of Nodes program representation in the GraalVM compiler (see Section 2.3). Graal IR is in SSA form, but most instructions 'float' rather than being assigned to specific basic blocks [4]. While SCC propagates values across the SSA data flow graph, it also needs the CFG for marking control flow edges as reachable or unreachable, and for visiting all instructions in a block. Our algorithm also propagates

values over the data flow edges of the Graal IR graph, but control flow reachability is propagated differently.

The main contributions of this work are:

- Data flow analysis using *lazy iteration*: Most nodes in the IR graph never need to be visited because they will never produce a constant, even if they can be executed. This is in contrast to Wegman and Zadeck's algorithm, which needs to visit every reachable instruction in the program.
- The *worklist priority ordering*: Optimistic data flow analysis using lazy iteration is only correct if SSA ϕ nodes are visited in a particular order, for which we developed a priority ordering (Section 3.5.2).

We present experimental data that shows that lazy iteration is very effective, visiting only 20.5 % of all nodes in the IR on average. It finds new constants not previously detected by the GraalVM compiler, which only features a non-optimistic version of constant propagation that does not compute optimistic fixed points over loops. The detected constants and unreachable CFG edges lead to an average speedup of 1.4 % on the standard Renaissance benchmark suite.

2 Background

2.1 Sparse Conditional Constant Propagation

The following summary of sparse conditional constant propagation is based on Wegman and Zadeck [7].

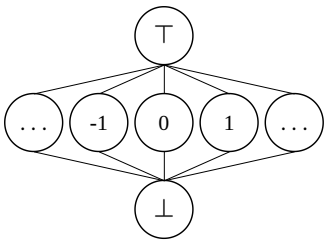

Figure 1. Three-tier value lattice.

SCC uses a three-tier data flow lattice shown in Figure 1. The lattice element for each variable is initially ⊤, representing a value that is yet unknown but may be determined to be a constant by the analysis. The constant elements in the middle tier represent values that will always evaluate to the given constant at runtime. The ⊥ element denotes values for which the analysis cannot guarantee a constant value. Lattice values are combined using the meet (greatest lower bound, ⊓) operation. Thus values can only be 'lowered' until a fixed point is reached.

The analysis tracks reachability of CFG edges using a two-tiered lattice, initially assuming that all edges are unreachable. At a control flow split, the analysis evaluates the split condition using its current information. Depending on whether the result is constant or not, only one or all outgoing control flow edges are marked as reachable and enqueued in a worklist for further iteration. Once a CFG edge is marked as reachable, it cannot become unreachable again.

Listing 1. Example of optimistic conditional constant propagation [1]. The if-branch inside the loop is never taken, and the function always returns 1.

```
public static int exampleCC(int a) {
    int x = 1;
    do {
        if (x != 1) {
            x = 2;
        }
    } while (a-- >= 1);
    return x;
}
```

Sparse conditional constant propagation is an optimistic analysis: When a control flow join point is reached but analysis information for some of the control flow predecessors is not yet available, the analysis can optimistically assume that the corresponding ϕ inputs are ⊤ and continue propagating information under this assumption. If the corresponding control flow paths are reachable, the algorithm guarantees that they will be traversed by the analysis at some point. If at that point the analysis provides a new value for a ϕ input, the optimistic assumption is invalidated, and program parts are re-analyzed with the new, lowered, information.

Optimistic analyses can discover more information than pessimistic ones, but they can only guarantee correctness if the analysis runs to completion, while pessimistic analysis can be interrupted at any time and still provide correct information [1]. Sparse conditional constant propagation is optimistic and integrates constant propagation with the analysis of unreachable code: Unreachable code paths are never analyzed and, by being associated with ⊤ data flow information, do not affect the analysis at all.

Listing 1 shows a small example adapted from [1] for a constant that cannot be found by any sequence of dead code elimination and simple constant propagation but can be found by conditional constant propagation. This is because of a cyclic dependency between data flow and control flow analysis. To detect `x` as constant, it must be known that the assignment statement `x = 2` is unreachable. To detect this statement as unreachable, it must be known that `x` is constant. This constant can only be found by first optimistically assuming `x` to be constant and subsequently verifying that the assumption was correct.

2.2 GraalVM

GraalVM[1] is a high-performance polyglot virtual machine. It executes programs written in Java, other languages that

[1] https://www.graalvm.org/

compile to JVM bytecode, and any other programming language implemented using GraalVM's Truffle language implementation framework. All input languages are transformed into a uniform internal representation and compiled to high-performance native code by the GraalVM compiler. GraalVM supports both just-in-time (JIT) and ahead-of-time (AOT) compilation of JVM languages.

2.3 Graal IR

The GraalVM compiler's intermediate representation (Graal IR) [4] is based on the *Sea of Nodes* concept [1]. This IR represents a program as a directed graph. Each node in the graph represents an operation. Edges between the nodes represent data and control dependences. Nodes are doubly-linked, i. e., from each node we can iterate efficiently both over its inputs and its usages.

Control flow edges only exist between so-called *fixed* nodes that must be strictly ordered because they have side effects (e. g., memory writes or method calls) or because they represent control flow transfers (branches, operations raising exceptions, or control flow merge points). All other nodes are *floating* nodes that are only constrained by data flow dependences. Floating nodes that are not connected via dependences are not ordered with respect to each other. Floating nodes are not assigned to any particular basic block. For final code generation and for certain optimizations, the GraalVM compiler computes a full schedule of the graph which assigns all nodes to blocks and imposes a strict order on them. Scheduling is an expensive operation, therefore most optimizations should work without a schedule.

Loops in the input program must be *reducible*, i. e., have a single loop entry. Graal IR uses a `LoopBegin` node to represent this entry point. Exits from the loop are represented as `LoopExit` nodes, and backedges are represented by `LoopEnd` nodes. Two-way control flow splits are represented as `If` nodes (multi-way `Switch` nodes are also available). The merging of control flow paths except loop backedges is represented as `Merge` nodes.

Graal IR uses SSA form, with ϕ nodes at loop begin and merge nodes, with one input per control flow predecessor. A ϕ node is a floating node but is connected to its fixed merge point by a control dependence edge. At a loop begin, the ϕ's first input is always the initial value on loop entry. At the point in the compilation pipeline when our constant propagation runs, the graph is in *loop-closed SSA form*: Values defined inside a loop must not be used directly outside the loop. Instead, special *proxy nodes* at loop exits mark the points where a value flows out of the loop.

Figure 2 shows a slightly simplified Graal IR generated for the method from Listing 1. In calculations, a constant input is directly shown in the node instead of connecting the node to the appropriate constant node in order to reduce the number of edges in the figure.

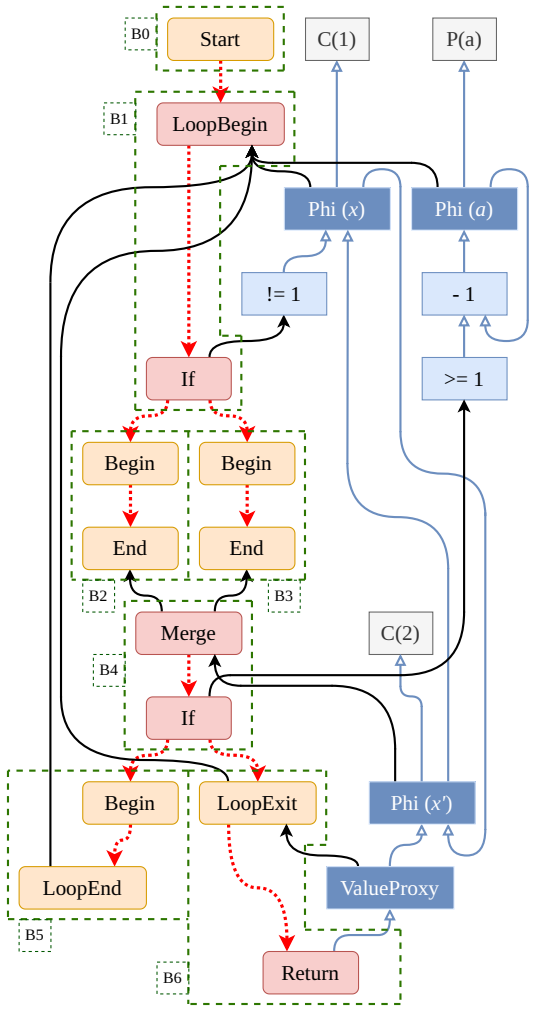

Figure 2. IR for the example presented in Listing 1.

A red dotted downward arrow represents a control flow successor, a black upward arrow depicts a control dependence, while a blue upward arrow with an empty head represents a data dependence. All nodes with rounded corners are fixed nodes and can therefore be directly attributed to a basic block, denoted by a dashed green outline enclosing multiple nodes. All nodes with sharp corners are floating nodes. Light gray nodes denote constants (`C`) or method parameters (`P`).

The `LoopEnd` node causes a jump back up to the `LoopBegin` node with which it is connected via a control dependence edge. The `LoopExit` node denotes control flow leaving the associated loop. `ValueProxy` nodes, which are connected with a loop exit, are inserted for values used inside the loop before any usage outside the loop.

The GraalVM compiler already performs constant propagation, but does not compute optimistic fixed points for loops. It also includes a general data flow analysis framework, but only for fixed nodes. This is sufficient for implementing its

Partial Escape Analysis [6] but is not appropriate for analyses that reason about floating arithmetic nodes, as would be needed for constant propagation. The GraalVM compiler currently has no data flow analysis framework that would take floating nodes into account. This work is the first step in a project to formulate such a general framework without needing to compute a schedule of the graph.

3 Approach

The approach presented in this paper aims to minimize analysis time of the graph by only lazily iterating over the parts of the graph relevant for finding constants while maintaining the power of *Conditional Constant Propagation* (CCP). This is in stark contrast to previous algorithms like *Sparse Conditional Constant Propagation* (SCCP) and *Sparse Simple Constant Propagation* (SSCP) which evaluate the entire (reachable portion of) the graph [7]. Additionally, while SCCP in the form described in the paper by Wegman and Zadeck [7] would require us to calculate a full schedule and to subsequently iterate over the entire reachable portion of the graph, our *Lazy Sparse Conditional Constant Propagation* (LSCCP) algorithm works directly on an unscheduled version of the Sea of Nodes representation.

Leaving parts of the graph unevaluated means that LSCCP needs to deal with unevaluated values as inputs for nodes and cannot assume that such nodes will be analyzed again later when complete information about all reachable inputs is available. Thus it needs to discern whether an input is unevaluated because the analysis has not reached it yet, or because this input will never generate a constant value and will therefore never be evaluated throughout the analysis.

The idea to mitigate this problem is to generally evaluate the program graph in a forward traversal order consistent with the order of execution which we refer to as a 'top-down' order (Section 3.5). This allows assumptions substituting missing information about inputs to be made safely.

Since LSCCP operates on an unscheduled Sea of Nodes representation where no global order of operations is known, we designed a priority metric for the work list based only on the available information. The priority metric enforces an order at critical points of the analysis where value flow analysis and reachability analysis influence each other (ϕ nodes and control split nodes). This allows value flow analysis and reachability analysis to be kept up to date with each other to provide each other with the best information possible.

Unlike classical data flow analysis where the order of evaluation is only relevant for the speed of the analysis, in our algorithm the order in which nodes are evaluated is essential for correctness.

3.1 Evaluation Domains

LSCCP uses two separate lattices to represent information in its value flow and reachability analysis respectively.

Value lattice. Values are represented by a lattice as in SCC. To avoid confusion with the values of the reachability lattice, we refer to the value lattice's $\top$ element as `UNSEEN` (no value known yet), the middle tier values as `CONSTANT`, and the $\bot$ element as `UNRESTRICTED` (no constant can be guaranteed). We use the name `UNSEEN` as a shorthand: It denotes both nodes that have never been visited by the analysis as well as nodes that have been visited but that have `UNSEEN` inputs.

Reachability lattice. In LSCCP all CFG edges are annotated with a reachability. Unlike SCC's two-tiered CFG reachability lattice, our reachability lattice contains three elements: `UNKNOWN` ($\top$), `UNREACHABLE` and `REACHABLE` ($\bot$). Using a lattice we can use logic in the reachability analysis that is similar to the value flow analysis when dealing with unevaluated inputs.

3.2 General Value Propagation

Initially all nodes in the graph are marked as `UNSEEN`. Due to the fact that value propagation starts at constants and not the CFG entry, evaluation may hit cases where a node's input is `UNSEEN` while it would be `UNRESTRICTED` if the input had been evaluated.

For example, consider evaluating the inequality check node representing the comparison $Phi(x) \neq 1$ in the graph in Figure 2. The first time we encounter this node in the analysis, one input is a constant 1 while the other input ($Phi(x)$) is still `UNSEEN`. We do not want to prematurely lower this to `UNRESTRICTED` since the `UNSEEN` input will become a constant later on, in which case we will want to produce a constant value for this node.

When visiting such a node, we treat all `UNSEEN` inputs as `UNRESTRICTED`. This still allows us to correctly treat an expression like `a * b` as 0 if `a` has been evaluated to 0 while `b` is still marked as `UNSEEN`. However, if any input was `UNSEEN` and the result of the visit would be `UNRESTRICTED`, we still propagate an `UNSEEN` result to signal that the result may be lowered to a constant later.

This allows us the flexibility to revisit nodes while preserving the overall invariant that a node's lattice value may only change to a lower value. In contrast to SCCP, an `UNSEEN` value for a node does not imply that the node has not been visited by the analysis yet.

3.3 Handling of ϕ nodes

The evaluation of ϕ nodes depends on both the value lattice elements for the ϕ's inputs and the reachability lattice element of the control flow edge associated with each input. For inputs coming from edges marked `UNKNOWN`, an assumption needs to be taken to evaluate the ϕ node. In general, a ϕ node can be evaluated using a *pessimistic* or an *optimistic* assumption regarding reachability. This means that incoming control flow edges with `UNKNOWN` reachability can be either

pessimistically interpreted as REACHABLE or optimistically interpreted as UNREACHABLE. A pessimistic assumption only retains maximum precision when we are sure that UNKNOWN edges will not be lowered to UNREACHABLE in the future.

3.3.1 Straight-Line ϕ nodes. Generally non-loop ϕ nodes are evaluated pessimistically. The top-down order of evaluating the graph ensures that the reachability information has already been calculated when evaluating the ϕ node, given that this ϕ node does not depend on backedges for which reachability can not yet be known and is subject to change. To show why we need pessimistic evaluation here, consider the following structure inside a loop:

```
if (condition)
  if (true)
    x1 = 1;
  else
    x2 = 2;
else
  x3 = 3;
x4 = φ(x1, x2, x3);
```

Here condition is not constant and will never be visited by our analysis because it cannot be evaluated at compile time, thereby leaving the reachability of the x3 input of the ϕ instruction UNKNOWN. The inner condition can be evaluated, leaving x1 as REACHABLE and x2 as UNREACHABLE. If we now optimistically assumed x3 to be UNREACHABLE, we would propagate the constant 1 into x4 which is an incorrect result. We would not recover from this mistake because condition will stay unevaluated throughout the analysis, thereby not triggering a reevaluation.

For a pessimistic evaluation to be admissible, it is required that UNKNOWN inputs are stable, which is the case for straight-line ϕ nodes as described in Section 4.2.

3.3.2 Loop ϕ nodes. Our handling of loop ϕ nodes combines both optimistic and pessimistic evaluation: We treat them optimistically when first entering a loop, but pessimistically after visiting the loop body.

In contrast to non-loop ϕ nodes, final reachability information for loop ϕ nodes is not yet available when a loop ϕ is evaluated for the first time when reaching a loop begin node. As the running example from Listing 1 demonstrates, a pessimistic assumption at this point would lose precision: Evaluating $Phi(x)$ pessimistically would never allow the CONSTANT 1 to enter the loop, thereby inhibiting the path through B2 to be found unreachable. This discovery is needed, however, to conclude that x is not modified inside the loop. Therefore, the loop ϕ must be evaluated optimistically to find that x is constant.

On the other hand, SCCP's optimistic analysis relies on the fact that every reachable block in the program will be visited, and every reachable control flow edge will be explicitly marked as reachable during the analysis. This allows the analysis to find the final value for a loop ϕ once the reachability information has stabilized. We cannot do the same kind of optimistic analysis since our analysis does not visit and mark all reachable control flow edges.

Therefore, we handle loop ϕ nodes as follows: When a loop is first entered, the loop begin's ϕ nodes are evaluated optimistically. This means that we assume that any UNKNOWN backedge may in fact be unreachable, and we ignore the associated input values. This allows us to propagate any constants entering the loop into the first loop iteration. At the same time, we schedule any loop ϕ with a not-UNSEEN lattice value for reevaluation after the entire loop (see Section 3.5.2). The organization of the worklist guarantees that the entire body is evaluated as far as possible before revisiting the loop ϕ nodes. At this point, the reachability of any UNKNOWN loop ends is guaranteed to be final (Section 4.2), and the loop ϕ node can safely be evaluated pessimistically, i. e., assuming that any still UNKNOWN backedge is now *reachable*. This may replace optimistically assumed constant values with the correct UNRESTRICTED value.

3.4 Reachability Propagation

To detect conditional constants, in addition to value flow, reachability needs to be taken into account. This information is tracked on a per edge basis because tracking it on a per block basis can lead to imprecisions, inhibiting detection of constants as shown by Click [1]. Processing, however, is done on blocks instead of edges by taking the reachabilities of the block's predecessor edges and its input values into account to calculate the reachability of its successor edges.

For a block to be considered reachable, it either has to be the start block of the CFG, or it has to have at least one predecessor edge that is not marked as UNREACHABLE. We can ignore backedges in this case because they can only occur on loop begins which have to be traversed to reach these backedges in the first place.

If a block has more than one successor, it must end with a control split node. In this case, the control split node is evaluated given its inputs and the successor edges are marked with the appropriate reachability lattice element. Similar to value flow explained in Section 3.2, immediately lowering successor edges from UNKNOWN to REACHABLE while predecessor edges are still UNKNOWN may inhibit future discovery of UNREACHABLE edges later on. Therefore, if an edge is considered reachable while predecessor edges are still UNKNOWN, we propagate UNKNOWN to signal that this edge might still be lowered later on.

To ensure that the control split nodes to be evaluated do in fact have up-to-date information from the value flow analysis, instead of immediately propagating reachability through them, they are scheduled using the worklist. While propagating reachability along the CFG, the reachability of

input edges of ϕ nodes may change. This new information triggers a reevaluation of all affected ϕ nodes.

3.5 Top-Down Analysis of the Graph

LSCCP evaluates nodes directly involved or closely related to control flow (such as ϕ nodes) in a forward traversal order consistent with the order of execution. Loops are analyzed to completion before information is propagated out of the loop, and predecessor blocks of control flow merges are evaluated before the merge. It creates conditions suitable for making assumptions about unevaluated inputs and to ensure that the best correct result can be calculated. This is achieved by the use of a priority-ordered worklist.

In the presentation that follows, lower numeric values denote higher priorities.

3.5.1 Base Priority Metric. In a quick pass over the CFG, a base priority is calculated for each block. This base priority is based on the minimal visit depth of a CFG block in a reverse postorder traversal [2]. A block's depth is calculated by incrementing the maximum of the depths of its predecessors by 1, ignoring loop backedges. The start block has a depth of 0.

The LSCCP base priority metric follows the same structure but extends it by adding one extra condition: If a block is a loop exit, its base priority does not only depend on its immediate predecessors but also all loop ends of the associated loop. All these loop ends are therefore regarded as predecessors of the loop exit while calculating the base priority. This effectively moves the loop exit below the entire loop in the priority.

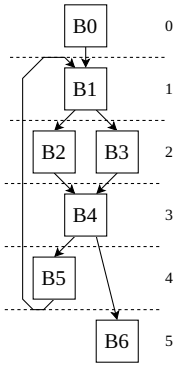

Figure 3. CFG for the IR presented in Figure 2.

Recalling the example introduced in Section 2.1, Figure 3 depicts the CFG from Figure 2. To the right of the CFG, the base priority of the blocks in the given line is given. Blocks that do not affect each other (in this example B2 and B3) can have the same base priority since the order in which they are processed does not affect the result of the analysis. The loop exit block B6 has a lower priority than any block in the loop, including the backedge block B5. This ensures that the loop is fully evaluated before evaluating any usages of values which depend on the given loop.

3.5.2 Priority-Ordered worklist. LSCCP uses a single priority-ordered worklist for both value flow and reachability analysis. This worklist internally consists of two queues. The first one is an unordered queue used for scheduling pure value flow nodes, such as arithmetic nodes. Elements are first removed from the unordered queue. Only if this queue is empty are elements from the second queue taken. Whenever a node from either worklist is visited, its usages are enqueued in the appropriate worklists if the analysis information associated with the current node changed.

This second queue is a priority queue. Elements are visited highest priority first (lowerst numeric priority value first).

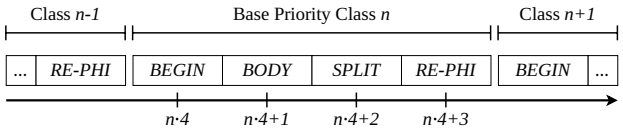

Figure 4. Internal layout of a block's base priority class.

The actual priorities used in the second queue are laid out as presented in figure 4. To maintain an approximation of the order of nodes within a basic block, the block's base priority n is quadrupled to allow us four *priority classes* per block. A priority class is a set (an equivalence class) of nodes with the same numeric priority value. We are not interested in the ordering of nodes within a priority class, and we do not need to represent the classes as explicit data structures. We only need nodes in higher-priority classes to be processed before nodes in lower-priority classes, which is ensured by the queue.

Nodes at the start of a CFG block and ϕ nodes are scheduled in the *BEGIN* class of their block's base priority class to ensure they are processed before any nodes in the current block that might use the value produced by this node.

Normal fixed nodes that can produce constant values (e. g., fixed division nodes that may raise an exception) are then scheduled using the *BODY* class. Finally, control split nodes that terminate blocks are scheduled after all fixed nodes in a block in the *SPLIT* class to ensure that all values that originate in the current block, in particular the condition controlling the split, are evaluated before the split itself.

In addition to scheduling a node given its base priority and position within a block, the worklist offers a second scheduling mode for ϕ nodes on loop begins (loop ϕ nodes for short). As will be discussed below (Section 3.3.2), loop ϕ nodes must be re-evaluated after the loop body has been fully evaluated. Therefore every loop ϕ is scheduled again in the `RE-PHI` class in the loop's last loop end (backedge) block in the priority queue.

Recalling the IR graph from Figure 2, *Phi*(x) is initially scheduled with priority 4, which is the *BEGIN* class of the base priority class 1 of its associated loop begin. Scheduling the `If` node with the condition `x==1` shows why maintaining an order within a basic block is necessary. The `If` node uses a value which depends on *Phi*(x) in the same basic block. Therefore the `If` needs to be evaluated *after* the given ϕ node in the *SPLIT* class, resulting in a scheduling priority of 6. To reschedule *Phi*(x), we first obtain the maximum base priority class of any associated loop end (which is 4 for B5). Then, to ensure we capture all values produced by this block, we schedule it using the *RE-PHI* class resulting in an effective priority of 19.

If the node *Phi*(a) were ever to be scheduled, it would receive the same priority as *Phi*(x) because their relative positions in the graph are the same. This collision does not matter: Control split nodes and ϕ nodes that receive the same priority are guaranteed to be independent and can therefore be evaluated in any order. In practice, the data structure used is a priority queue, therefore nodes with the same priority are evaluated in a first-in-first-out manner.

The worklist keeps track of currently scheduled nodes, ensuring each node only exists in the list once. If a node is trying to be scheduled a second time with a different priority, the original priority is kept.

Overall our priority-ordered worklist captures the same relative ordering information between nodes that we would need from a schedule of the graph. However, as we only need ordering information between certain fixed nodes and ϕ nodes, and as nodes are only added to the priority queue on demand as required by the analysis, the computation of this ordering information is much cheaper than the computation of a full schedule.

3.6 Putting Everything Together

Finally, we present the full Lazy Sparse Conditional Constant Propagation in Algorithm 1.

We start by setting the value lattice elements of all nodes (denoted by $\lambda(node)$) to `UNSEEN` and the reachability lattice elements of all CFG edges (denoted by $\Lambda(edge)$) to `UNKNOWN`. Then we initialize the worklist with all constant nodes in the graph. LSCCP processes nodes until the worklist is empty. Finally, we replace all nodes for which we found new constants.

3.7 Example of LSCCP analysis

In this section we present a full run of LSCCP on the running example program with its graph shown in Figure 2. Table 1 shows the states of the worklist throughout the example run. The worklist is separated into the value queue holding floating arithmetic nodes, and the priority queue holding fixed nodes as well as ϕ and proxy nodes.

Algorithm 1 LSCCP

```
1:  procedure LSCCP
2:      initialize all nodes with UNSEEN
3:      initialize all CFG edges with UNKNOWN
4:      initialize worklist with all constants
5:      while worklist has items do
6:          c ← worklist.next()
7:          if c is a φ-node then
8:              ProcessPhi(c)
9:          else if c is a control flow node then
10:             ProcessControlFlowNode(c)
11:         else
12:             ProcessValueFlowNode(c)
13:     replace all nodes found to be constant

14: procedure ProcessPhi(phi)
15:     if phi is a loop φ node ∧ λ(phi) = UNSEEN then
16:         new ← λ(first input of phi)
17:         reschedule phi if lowered
18:     else
19:         new ← meet(λ(reachable inputs of phi))
20:     UpdateValueLatticeElement(phi, new)
21: procedure ProcessControlFlowNode(flow)
22:     for all e in successor edges of flow do
23:         if CFG block of flow is reachable then
24:             new ← true if flow is no control split or e
                      is reachable according to
                      λ(flow.condition) else false
25:         else
26:             new ← false
27:         UpdateReachability(e, new)
28: procedure ProcessValueFlowNode(val)
29:     new ← evaluation of val given its inputs
30:     UpdateValueLatticeElement(val, new)

31: procedure UpdateValueLatticeElement(node, elem)
32:     if elem < λ(node) then
33:         λ(node) ← elem
34:         schedule usages of node
35: procedure UpdateReachability(edge, reachable)
36:     target ← target of e
37:     if reachable is false then
38:         new ← UNREACHABLE
39:     else if Λ(e) = UNREACHABLE then
40:         new ← REACHABLE
41:     else return
42:     Λ(edge) ← new
43:     schedule φ-nodes at the start of target
44:     schedule target
```

Table 1. Worklist states throughout the example run

Step	Current Node	Evaluation result	Value Queue (after)	Priority Queue (after)
0	(initial state)		C(1), C(2)	
1	C(1)	1	C(2), ≠1, -1, ≥1	4: $Phi(x)$
2	C(2)	2	≠1, -1, ≥1	4: $Phi(x)$, 12: $Phi(x')$
3	≠1	UNSEEN	-1, ≥1	4: $Phi(x)$, 12: $Phi(x')$
4	-1	UNSEEN	≥1	4: $Phi(x)$, 12: $Phi(x')$
5	≥1	UNSEEN		4: $Phi(x)$, 12: $Phi(x')$
6	$Phi(x)$	1	≠1	12: $Phi(x')$, 19: $Phi(x)$
7	≠1	true		6: If(B1), 12: $Phi(x')$, 19: $Phi(x)$
8	If(B1)	Λ(B1 → B2) := UNREACHABLE		8: Begin(B2), 12: $Phi(x')$, 19: $Phi(x)$
9	Begin(B2)	Λ(B2 → B4) := UNREACHABLE		12: $Phi(x')$, 15: If(B4), 19: $Phi(x)$
10	$Phi(x')$	1		15: If(B4), 19: $Phi(x)$, 20: ValueProxy
11	If(B4)			19: $Phi(x)$, 20: ValueProxy
12	$Phi(x)$	1		20: ValueProxy
13	ValueProxy	1		21: Return
14	Return			

Steps 0–2. First we start by adding the two constant nodes `C(1)` and `C(2)` to the worklist. As these nodes are neither fixed nodes nor ϕ nodes, they are inserted into the value queue of the worklist. Now we remove `C(1)` from the worklist, set its value to a `CONSTANT 1` and schedule all its usages. The inequality, subtract and greater-equal nodes are scheduled in the value queue of the worklist (recall that for brevity the usage of C(1) in these nodes was not indicated by edges), while $Phi(x)$ is scheduled in the priority queue with priority 4 (= $1 \cdot 4 + 0$, see Section 3.5.2). Evaluating `C(2)` similarly causes its value to be set to a `CONSTANT 2`, and its usage $Phi(x')$ is scheduled with priority 12 (= $3 \cdot 4 + 0$).

Steps 3–5. Evaluating the previously scheduled inequality, subtract and greater-equal nodes yields `UNSEEN`: Each of these nodes has one `UNSEEN` and one constant input. Evaluation results in an unknown value which is propagated as `UNSEEN` to signal the possibility that the value may still become a constant in the future (see Section 3.2). Because this does not change the lattice values for these nodes, their usages are not scheduled.

Step 6. Now the value queue of the worklist is empty, therefore we remove the first element of the priority queue, which is $Phi(x)$. As explained in Section 3.3.2, we optimistically assume the second input of $Phi(x)$ to be `UNREACHABLE` and propagate the `CONSTANT 1` through this node. To check this assumption later on, we reschedule $Phi(x)$ with priority 19 (= $4 \cdot 4 + 3$). Because we lowered the value of $Phi(x)$, all its usages will be scheduled, causing the inequality and $Phi(x')$ to be scheduled. Since $Phi(x')$ already exists in the worklist, it is not inserted a second time (see Section 3.5.2).

Step 7. The next node to be evaluated is the inequality node. This node now has two constant inputs and can be evaluated to a `CONSTANT true`, scheduling the associated `If` node with priority 6 (= $1 \cdot 4 + 2$).

Steps 8–9. Evaluating the `If` node scheduled right before, we see that the false branch of this condition is unreachable because of the `CONSTANT true` input. Therefore, we set the edge from B1 to B2 to `UNREACHABLE` and schedule the begin node of B2 with priority 8 (= $2 \cdot 4 + 0$) to represent the entire block for reachability analysis, see Section 3.6. This node is immediately removed from the worklist and since the only predecessor edge is `UNREACHABLE`, the successor edge from B2 to B4 is marked `UNREACHABLE` and the `If` node at the end of B4 is scheduled (see Section 3.6) with priority 15 (= $3 \cdot 4 + 3$) to represent B4 for reachability analysis. Additionally, $Phi(x')$ at the start of B4 would be scheduled if it were not already in the worklist.

Step 10. Now $Phi(x')$ is the next node to be evaluated. Its inputs are a `CONSTANT 2` on the first input and a `CONSTANT 1` on the second input. However, the first input value is considered unreachable because it corresponds to the CFG edge from B2 to B4, therefore the `CONSTANT 1` from the second input can be propagated. We would schedule its usage (i.e. the loop $Phi(x)$ with priority 4), but because it is already in the worklist with priority 19, we do not re-schedule it. The second usage of $Phi(x')$ is the `ValueProxy` node which gets scheduled in its loop exit block with priority 20 (= $5 \cdot 4 + 0$, see Section 5).

Step 11. The next node to be evaluated is the `If` node at the end of the basic block B4. B4 is considered reachable because it does not exclusively have `UNREACHABLE` edges as predecessor edges (the edge from B2 to B4 is `UNKNOWN` which is interpreted as `REACHABLE`, see Section 3.4). This in combination with the `UNSEEN` input coming from the greater-equal

node results in both successor edges being considered reachable, resulting in no change to their associated reachability lattice level.

Step 12. Then $Phi(x)$ is reevaluated to check if the previous assumption for this node still holds. Both incoming control flow edges are UNKNOWN and therefore considered reachable, but both inputs to the ϕ are also associated with a CONSTANT 1. Therefore, this node is evaluated to CONSTANT 1 which confirms our optimistic assumption made earlier and the value of $Phi(x)$ is not lowered further.

Steps 13–14. The next node to process is the ValueProxy which propagates its input CONSTANT 1, resulting in the Return node to be scheduled with priority 21 ($= 5 \cdot 4 + 1$, see Section 3.5.2). Finally, the Return node is processed. Since it does not produce a value it is not analyzed further, thereby leaving the worklist empty.

End. This concludes the analysis for conditional constants. We found four nodes ($Phi(x)$, inequality, $Phi(x')$ and ValueProxy) that can be replaced with constants at their usages. Additionally we found two CFG edges (B1 to B2, B2 to B4) and one basic block (B2) to be unreachable. These can be eliminated from the CFG.

4 Correctness and Precision

This section provides reasoning for the assumptions made in Section 3 that are needed to ensure that LSCCP finds at least as many constants as SCCP while not erroneously reporting values as constant.

4.1 Isolated Analyses

Looking at the general value flow analysis presented in Section 3.2 in isolation, we can conclude that for any given input, this analysis produces a correct result that is at least as high as the one generated by SCCP when interpreting UNSEEN as UNRESTRICTED, because we propagate constants as soon as possible while never blocking future analysis. All evaluation functions are designed to lower a value from UNSEEN as soon as possible without blocking future discovery of constants. If at any point in the analysis all inputs of a general value flow node were not UNSEEN and no CONSTANT value can be calculated, the evaluation function is required to result in UNRESTRICTED for all future queries, to uphold the assumption that during the evaluation of the first loop iteration we have the maximum amount of values assumed to be constant for Section 4.2.

Inspecting the pure reachability analysis (without control splits), it is easy to conclude that propagating the blocks reachability onto the successor edge if the block itself is considered unreachable, generates a correct result for these edges when interpreting UNKNOWN as REACHABLE. Since any unreachable edge must transitively depend on a constant value which in turn again depends on a constant, our analysis will find all of these edges.

4.2 Combination of value and reachability analyses

Reachability analysis and value flow analysis interact at control split nodes and ϕ nodes. Control splits depend on the reachability of their predecessor edges as well as the condition or value on which they split to produce results in the reachability analysis. The analysis of ϕ nodes depends on the reachability of the predecessor edges of their connected Merge as well as their input values to produce a result in the value flow analysis. To analyze such nodes without generating incorrect results while still finding at least as many constants as SCCP, we need to be able to draw reliable conclusions about the true values of UNSEEN and UNKNOWN inputs.

Control splits. These nodes are handled very similarly to pure reachability analysis (recall Section 3.4). Given the value which the control split depends on is correct, it is easy to see that evaluation of these nodes yield the best correct result. The ordering in the worklist in combination with the initial optimistic evaluation of the loop ϕ nodes (recall Section 3.3.2) ensure that, any of the split's successor edges that is not UNREACHABLE on the first evaluation, will always be reachable throughout the analysis.

Straight-line ϕ nodes. Straight-line ϕ nodes can safely assume that during their first evaluation all inputs associated with $\top$ lattice elements will stay that way throughout the analysis due to the worklist ordering, the initial optimistic evaluation of loop ϕ nodes and the evaluation functions of general value flow analysis as shown in Section 4.1. Such inputs can therefore be safely assumed as their $\bot$ counterparts resulting in a correct result that is at least as good as the one produced by SCCP.

Loop ϕ nodes. When re-evaluating a loop ϕ node we rely on the fact that during the first evaluation of the loop, we work with the maximum amount of values that are assumed to be constant (which is ensured by the initial optimistic evaluation of said loop ϕ node), which would cause all CFG edges that might at some time be UNREACHABLE to be lowered to this reachability lattice element. This allows us to safely conclude that all edges that are UNSEEN after the first evaluation of the loop, are sure to stay that way throughout the analysis, which we rely on as mentioned in Section 3.3.2. This creates the necessary preconditions to treat the loop ϕ node as a straight-line ϕ node upon re-evaluation, meaning it can be evaluated pessimistically, guaranteeing correctness upon convergence without lowering precision below SCCP.

4.3 Conclusions

Because all parts of the analysis end up generating correct results we can conclude that the analysis as a whole generates a correct result. Termination is guaranteed because

the transfer functions for all nodes are monotone and the lattices have finite height.

Because all lattice elements that are UNSEEN or UNKNOWN throughout the analysis (which are the only lattice elements in the analysis for which this assumption is taken) can be safely assumed as their $\bot$ counterparts, and because the value lattice tracks at least as many elements as SCCP, the result generated by LSCCP is at least as good as the one generated by SCCP.

5 Implementation

We developed a prototype implementation of the LSCCP algorithm in the GraalVM compiler.

In our implementation, we assign priorities to loop exit nodes and the associated proxies (see Section 2.3) so that these are only visited once the loop's fixed point has been reached. Otherwise, if optimistic constant values or reachability information were to leak out of loops, we would waste effort analyzing code after the loop which would have to be re-analyzed once the fixed point is reached.

Additionally, in the procedures UpdateValueLatticeElement and UpdateReachability, checks were inserted to guarantee that monotonicity is upheld. In UpdateValueLatticeElement $elem \leq \lambda(node)$ must hold, otherwise monotonicity would be violated while in UpdateReachability the condition $reachable \vee \Lambda(edge) \neq$ REACHABLE must hold for monotonicity to be upheld.

5.1 Four-tier Value Lattice

In our implementation, the data representation of the value lattice uses the GraalVM compiler's existing 'stamp' infrastructure for representing ranges of values. While we do not propagate ranges in general, we use the fact that integer stamps provide a 'known to be nonzero' flag. Thus, in contrast to previous constant propagation algorithms which usually operate on a value lattice with three tiers (as presented in Section 2.1), our value lattice for integers actually has four levels: An extra level below all nonzero constants is added expressing that a value is not known to be constant but known not to be zero. Similarly, we track floating point values with a four-level lattice with a 'not-NaN' level, as well as object references with a 'not-`null`' level. We refer to these not-zero, not-NaN, and not-`null` values as the 'non-special' tier in the lattice.

The additional non-special tier is useful for evaluating common conditions such as `value != 0` (e. g. before a division) or `reference != null` (before a memory access that would otherwise raise an exception). The non-special tier is also useful for tracking Boolean values. In the JVM, Booleans are internally treated as 32-bit integer values where `true` is defined as any `value != 0`.[2] Therefore, `true` values are harder to track because they may not have a constant value associated with them internally, but the non-special tier in the value lattice allows for easy reasoning.

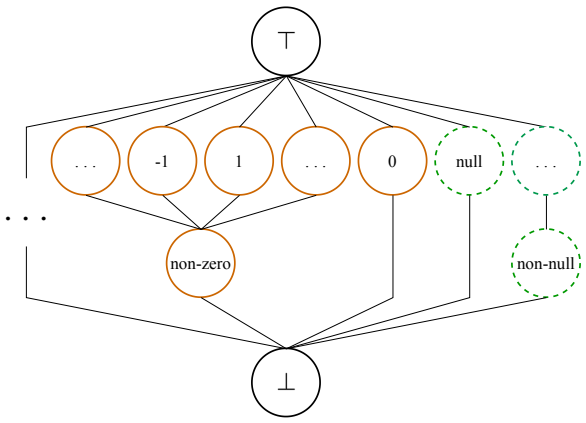

Figure 5. Four-tiered value lattice used by LSCCP.

The four levels described above are illustrated in Figure 5. The lattice elements between $\top$ and $\bot$ are color-coded to represent types. Integer values are outlined in orange (solid) while object references are outlined in green (dashed). The three dots on the left-hand side indicate that there are more types (e.g. floating point values) in this lattice than shown in the figure.

5.2 Conditional Nodes

The GraalVM compiler has conditional nodes representing the computation `condition ? trueValue : falseValue`, which are a fusion of a control split followed by an immediate merge and a phi node. This floating node poses an interesting problem because it has no ties into the control flow portion of the graph and is therefore hard to schedule with priority using the worklist.

While in the case of ϕ nodes, the condition and its associated control flow are guaranteed to be evaluated before the ϕ node, this is not the case with conditional nodes. Consider the case of a conditional `condition ? 1 : 2`, where the condition is still associated with the UNSEEN state. We might want to propagate the non-zero value for the result of this expression. However, if the condition became a known constant later, we would have to change this result from non-zero to a constant. This would violate monotonicity, as the non-zero tier is below the constants in the lattice, and values must only change to lower lattice elements.

To resolve this issue, we prevent the evaluation of a conditional node to non-zero if its condition is associated with UNSEEN. While this may inhibit further discovery of values, tests showed that this case rarely shows up in practice. It would be possible to handle this case precisely by adding another kind of 'non-zero' tier *above* the constant layer in the value lattice. We decided that this case was not worth

[2] https://docs.oracle.com/javase/specs/jvms/se22/jvms22.pdf section 2.11.1

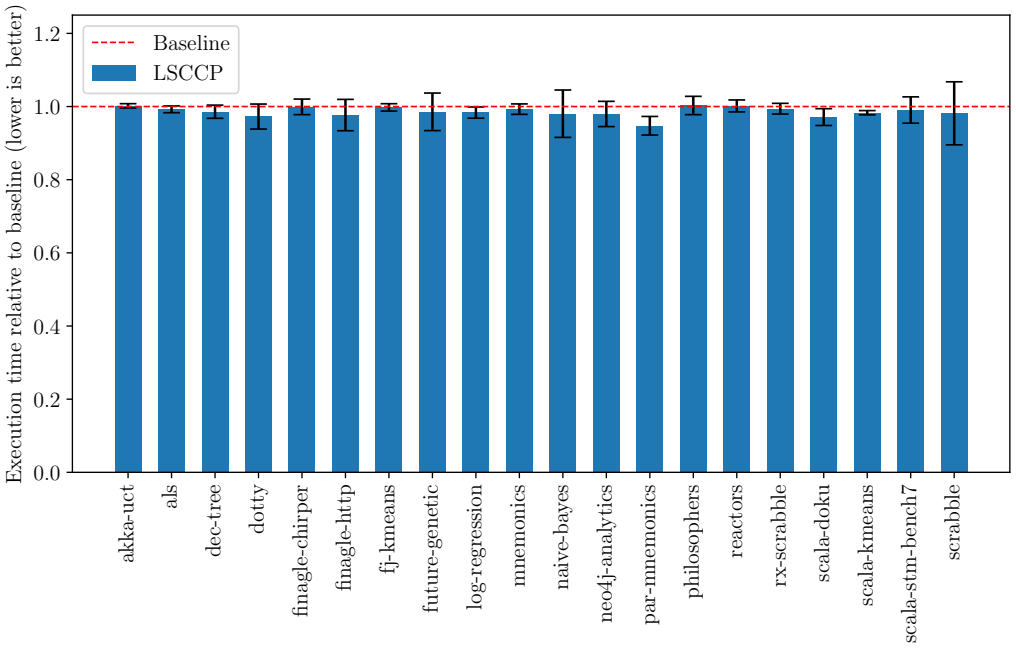

Figure 6. Results obtained for the Renaissance Benchmark Suite.

the extra complexity and prefer the slight imprecision from propagating UNSEEN instead.

6 Evaluation

Our implementation passes the unit test suite of GraalVM and the compilation of the entire Java Standard Library.

6.1 Benchmarking setup

We evaluated our implementation of LSCCP on the Renaissance benchmark suite[3]. The system used for testing runs Ubuntu 22.04 on an Intel 11th Gen Intel Core i7-1165G7 processor equipped with 64GB of RAM. A command line switch was built into GraalVM to disable the newly implemented conditional constant propagation phase to test baseline and the implementation of LSCCP on the same build. For each benchmark, warm-up runs are executed, followed by timed runs which contribute to the final result. The per-benchmark default number of warm-up and timed runs specified by the Renaissance benchmarking harness were used in this evaluation. In an effort to reduce the effect of noise, the entire benchmark suite was executed 15 times for both baseline and LSCCP. Baseline and LSCCP runs were carried out alternately to increase fairness.

[3] https://renaissance.dev/

6.2 Results

Figure 6 shows average results over the 15 benchmark runs for the 20 benchmarks of the Renaissance benchmark suite that are supported on the platform used for testing. The error bar indicates the standard deviation encountered over all timed iterations of the 15 LSCCP runs. As expected, LSCCP performs slightly better (though still mostly within margin of error) than baseline for most cases. No benchmark has seen any reliably measurable performance regression. The mean increase in performance measured over all 20 benchmarks in this suite was 1.4 % (minimum -0.28 %, maximum 5.25 %, median 1.43 %). LSCCP found on average 0.15 % of the values in the graphs to be constant, excluding constants found earlier through non-optimistic constant propagation.

The 'par-mnemonics' benchmark has seen the largest performance improvement of 5.25 %. In this benchmark, 0.29 % of all nodes were newly evaluated to be constant by LSCCP. The worst performance regression was measured for 'philosophers' with 0.28 %, for which only 0.03 % of all nodes were newly evaluated to constant by LSCCP. Since the conditional constant propagation phase does not change the graph if no new constants were found, the difference produced by LSCCP was negligible for this benchmark. We conclude that this result is within margin of error to baseline.

Our evaluation also showed that lazy iteration is very effective, as LSCCP only evaluates 20.5 % of the nodes in the graph on average. This is a significant improvement over the

previous state of the art, which would evaluate the entire reachable portion of the graph, which makes up 99.3 % of all basic blocks in the evaluated benchmarks. For the nodes that are visited by the analysis, the average number of visits per node is 1.1, indicating that the analysis quickly converges towards a fixed point.

Overall our optimization improves peak performance by an average of 1.4 % over GraalVM's optimized baseline on the Renaissance benchmarks, although this is mostly within the measurement noise on these benchmarks.

6.3 Discussion

As discussed before, GraalVM already performs conditional constant propagation, but only computing pessimistic fixed points for loops. Therefore any difference in our results vs. GraalVM's baseline analysis can only come from certain degenerate loop patterns, where our optimistic LSCCP analysis can prove that the loop will be exited on its first iteration, or that a variable used in the loop is a loop-invariant constant.

We do not provide a more detailed comparison of the two approaches: As GraalVM performs its non-optimistic constant propagation on the fly while simplifying nodes during other phases, there is no distinct non-optimistic constant propagation phase. Therefore, a more direct comparison against GraalVM's existing constant propagation would not be possible to do fairly since constant folding and propagation is deeply intertwined with the 'canonicalization' cleanups that run many times during compilation. Most compiler phases expect the program to be in canonical shape before they process them. Removing constant folding from canonicalization would have a very large detrimental impact on most of the compiler. Running a specialized baseline constant propagation pass at one or a few points in the compilation pipeline would be possible, but it would not make up for optimization opportunities lost from compiler phases that were unable to do their work on non-canonical inputs. Any such restructuring of the compiler would produce entirely artificial results.

7 Conclusions and Future Work

We presented a formulation of conditional constant propagation in a Sea of Nodes, exploiting the properties of its floating nodes and carefully organizing the iteration order of the analysis to connect data flow to control flow. Our approach features *lazy iteration* to reduce the portion of the graph necessary to be evaluated for finding all conditional constants. The evaluation of our prototype showed that lazy iteration is very effective, only evaluating 20.5 % of the graph.

In a next step, this analysis could be extended to allow `if` and `switch` statements to generate new data flow facts in the respective branches for the usages of the values they depend on. To achieve this, an approximation of a schedule must be calculated for values those values to find the correct usages to inject the data flow facts into.

This work is part of a larger project aimed at implementing a general data flow analysis framework in the GraalVM compiler. To our knowledge, this is the first general data flow analysis on the data flow component of a Sea of Nodes graph. Click's thesis [1] presents a powerful combined analysis that identifies constants, unreachable code, and congruences between values in the Sea of Nodes. However, this is a custom analysis that is not formulated in terms of a general data flow analysis framework.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Tomoki Nakamaru for their feedback that has helped us improve an earlier version of this paper.

This research project was partially funded by Oracle Labs. We thank all members of the Virtual Machine Research Group at Oracle Labs. Oracle, Java, GraalVM, and HotSpot are trademarks or registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. We also thank all researchers at the Johannes Kepler University's Institute for System Software for their support of and feedback on our work.

References

[1] Cliff Click. 1995. *Combining Analyses, Combining Optimizations.* Ph. D. Dissertation. Rice University. https://hdl.handle.net/1911/96451

[2] Keith D. Cooper and Linda Torczon. 2004. *Engineering a Compiler.* Morgan Kaufmann.

[3] 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 (1991), 451–490. https://doi.org/10.1145/115372.115320

[4] 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.* http://ssw.jku.at/General/Staff/GD/APPLC-2013-paper_12.pdf

[5] John B. Kam and Jeffrey D. Ullman. 1977. Monotone data flow analysis frameworks. *Acta Informatica* 7 (1977), 305–317. https://doi.org/10.1007/BF00290339

[6] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2018. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14).* Association for Computing Machinery, New York, NY, USA, 165–174. https://doi.org/10.1145/2544137.2544157

[7] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (1991), 181–210. https://doi.org/10.1145/103135.103136

Received 2024-05-25; accepted 2024-06-24