Efficient Method Extraction for Automatic Elimination of Type-3 Clones

Ran Ettinger*, Shmuel Tyszberowicz†, Shay Menaia‡

*Ben-Gurion University of the Negev, Israel
†The Academic College of Tel Aviv-Yaffo, Israel

Abstract—A semantics-preserving transformation by Komondoor and Horwitz has been shown to be most effective in the elimination of type-3 clones. The two original algorithms for realizing this transformation, however, are not as efficient as the related (slice-based) transformations. We present an asymptotically-faster algorithm that implements the same transformation via bidirectional reachability on a program dependence graph, and we prove its equivalence to the original formulation.

I. INTRODUCTION

With the target of improving the maintainability of existing software, clone refactoring is a process of identification and subsequent elimination of duplication in a given code base [16]. In the case of clones that were created by copying, pasting, and then modifying existing code, the duplication may involve non-contiguous sets of statements. The elimination of such so-called type-3 clones [22] involves the matching of identical statements among the clone instances (or similar statements that can be made identical by the introduction of a parameter) followed by the motion of unmatched interleaved statements, such that each clone instance would become contiguous [25]. This way, the transformed matched code in one clone instance would be ready for extraction into a new—reusable—method, whereas the matched code in all other clone instances could be replaced by a reuse of that newly-extracted method. The automation of such a process depends on the ability to find a solution to an optimization problem [16]: find the largest possible extractable sub-clone, while introducing the shortest possible parameter list to the extracted method. In the context of binary code, where the goal of clone elimination would be the compaction of executable code [3], only the former requirement—of eliminating the largest sub-clone possible—would be relevant. However, in the refactoring of source code, readability is definitely of high importance [9], [12]; thus, it is imperative that the code would be as similar to the original as possible, and hence readable and recognizable to the human programmer.

The challenge of preparing non-contiguous sets of statements for semantics-preserving extraction has been addressed in the literature, e.g. [17], [14], [18], [15], [13], [7], [4], [23], [24], [1], [5]. Of those, the approach of Komondoor and Horwitz [15], [13] has been demonstrated as most effective in the elimination of type-3 clones [13], [6]. Their solution employs a combination of transformation techniques, including statement reordering and the duplication of predicates and jumps. The extraction process takes the form of an automatic semantics-preserving source-code transformation. Given a set of (not-necessarily contiguous) marked statements, the transformation prepares them for extraction into a reusable method. This is achieved by attempting to move the interleaved unmarked statements to either before or after the marked code. Some measures are taken to ensure semantics preservation, such as the duplication of conditionals.

An algorithm to implement this transformation is expected to compute three code fragments to substitute an existing code fragment, by assigning each of that fragment’s statements into one of three buckets: before, marked, or after; we therefore refer to the transformation as 3-bucketing, or simply bucketing. Consider, for example, the pair of clones in Fig. 1, with the matched (identical) statements \{4, 6, 7\} (top part of the figure) and \{11, 13, 14\} (bottom) marked by “++”. This is a shortened and simplified version of a real clone pair found in the Tiarks benchmark of type-3 clones [22] and previously used in its original form for the illustration of bucketing [6]. For expository reasons, we assume that the called methods—\(f1-f7, g2, g3, g5\)—incur no side effects and their results depend on the value of the sent parameters only; hence the dataflow can be trivially inferred from the included text. The bucketing algorithm assigns statement 2 to the before bucket and statement 5 to the after bucket. A copy of the predicate 1 and the enclosing conditional statement is added to the resulting before fragment (Fig. 2), to ensure semantics preservation.

When the original bucketing transformation is not successful in correctly moving an unmarked statement, it assigns it into the marked bucket; this is known as promotion [15]. In our example, as can be seen in Fig. 2, predicate 1 and statement 3 end-up being promoted; they are both needed in the marked bucket (and hence in the body of the extracted method) to preserve semantics; the promoted predicate is essential to preserve control flow, whereas the promoted statement is needed to preserve dataflow; indeed, moving statement 3 to the before bucket could change the result of the promoted predicate (due to the modified value of \(x\)), whereas moving it to the after bucket (with no additional compensating code modifications) is not possible as its result, in the modified \(x\), is used in the last marked statement.

Komondoor and Horwitz (KH) have shown [15] that their solution compares favorably with previously proposed trans-
public void f() {
    ...
    1: if (f1(x1)) {
        2: x2 = f2();
        3: x1 = f3(x2, x3++);
        4: ++ x4 = f4(x2);
    }
    5: x5 = f5(x3);
    6: ++ x4 = f6(x4);
    7: ++ x4 = f7(x3, x4);
    ...
}

public void g() {
    ...
    8: if (f1(x1)) {
        9: x2 = g2();
        10: x1 = g3(x2, x3++);
        11: ++ x4 = f4(x2);
    }
    12: x5 = g5(x3);
    13: ++ x4 = f6(x4);
    14: ++ x4 = f7(x3, x4);
    ...
}

Fig. 1: Example code, ahead of bucketing on a pair of type-3 clones, marked by ++.

Fig. 2: Example code after bucketing, before method extraction, with promoted predicates and statements marked by +++.

The statement numbers highlight the involved transformations of statement reordering and predicate duplication.

formations, most notably in comparison to the Tuck transformation by Lakhotia and Deprez [17]. This effectiveness is partly thanks to the provision for dataflow, to and from the extracted clone. In terms of time complexity, however, the original bucketing algorithm is not as attractive: unlike the related transformations—e.g. [17], [18], [5]—performing a reachability algorithm on a dependence graph, requiring time linear in the size of that graph (and hence quadratic in the number of nodes in the worst case), this algorithm takes time cubic in the number of nodes.

According to a clone refactorability definition by Tsantalis et al. [25], a clone-elimination transformation is considered unacceptable whenever the given code fragments contain any statement that cannot be moved (either before or after the extracted code). Accordingly, an application of the 3-buckets transformation of Komondoor and Horwitz is deemed unsuccessful whenever any non-marked statement is promoted (and must therefore be extracted along with the marked code). We have recently proposed [6] an adaptation of the original bucketing method-extraction approach that yields refactorable results according to the definition of Tsantalis. We achieved this through a search for the largest subset of the initially marked statements whose extraction requires no promotion. An evaluation of this approach on real-world type-3 clones from the Java portion of the Tiarks benchmark [22] has produced encouraging results [6]. This investigation has reconfirmed the appropriateness of bucketing for clone elimination, with effective movement of interleaved non-identical statements in 59 of the 110 clone pairs, comprising 18 distinct cases. The bucketing duplication of predicates was needed in 10 of these 18 cases, demonstrating the value of duplication as a step towards the removal of (larger portions of) code duplication. Moreover, our proposed refactorability improvement, searching for the largest non-promoting (and hence refactorable) sub-clone, was indeed needed in 11 of the 18 distinct cases. In the example of Fig. 1, the largest such subset of the cloned statements \{1, 4, 6, 7\} is \{1, 4, 6\}, requiring the removal of a single statement. Clone C1 of Fig. 6 is an example in which the largest refactorable subset of the initially-marked statements is substantially smaller than the original set \{6\}. The largest non-promoting subset of the initially-marked C1 is C2; this subset involves 11 statements (including 6 predicates), out of the 26 statements (12 predicates) in the full C1. In searching for C2, an automatic tool would need to employ the bucketing algorithm as many as 14,914 times (for all subsets of size 14 to 5, of the set of non-predicate statements in C1). When such large numbers of runs of a bucketing algorithm are needed—in the automation of clone elimination—to find an optimally extractable sub-clone, the cubic time-complexity of the original algorithm might not be acceptable.

Other optimization goals could be, for example, to find the subset of \(M\) yielding the smallest number of duplicated predicates or the smallest number of parameters required by the extracted method. Furthermore, the process of automatic identification of method-extraction opportunities with the goal of improving the quality of an existing code base may require
a massive number of runs of the algorithm on each existing method body. This has been proposed, for example, by Tsantalis and Chatzigeorgiou [24]; they have applied and extended, in this context, Maruyama’s algorithm for the extraction of block-based slices [18]. According to that proposal, a large number of decompositions of any method is automatically considered, based on the code slices of each variable or object whose value may be modified when that method is invoked. An attempt to formulate such a mechanism based on the bucketing transformation is a potential target for future work. A bucketing-based decomposition into three fragments rather than two, with potential dataflow (to and from the selected variable’s slice), is likely to yield more attractive results. The automatic identification of such opportunities on any method, for the extraction of various sets of marked statements, would require a large number of runs of the algorithm on a single method.

This paper attempts to improve the appropriateness of the latest version of bucketing, the one from Raghavan Komondoor’s doctoral thesis [13], for solving such optimization problems in the context of automatic type-3 clone elimination. We propose a new algorithm that implements this transformation through two carefully crafted linear-time traversals of the program dependence graph. The contributions of the paper include (1) a new method-extraction algorithm performing the same code transformation as the original 3-buckets algorithm, asymptotically faster; and (2) a proof that the two algorithms produce the same results. The reduced time complexity turns out to be essential for optimal and effective elimination of subclones and clone groups, even when the number of involved statements is not so high. And the importance of the proven equivalence of results can be highlighted by recalling that Komondoor has provided a proof of semantics preservation of his original algorithm [13]. It is also worth noting that in line with his original formulation, the code transformation considered in this paper focuses on statement reordering with predicate duplication, in preparation for method extraction. The actual extraction of the marked (and promoted) code fragments into reusable procedures/methods remains outside the scope of our attention—these fragments will have become contiguous by bucketing and therefore ready for extraction in the usual way [10], [19], [9], [26], [21].

II. A PROGRAM REPRESENTATION FOR THE MOTION OF NON-CONTIGUOUS CODE

Before we turn to introduce our new algorithm, some definitions on the representation of control and data flow are due. These will help us first in presenting our approach in a precise manner, and will be used later in the paper for proving equivalence with previous work.

The version of a control flow graph (CFG) representation of a code fragment needed for our work takes the form of a directed graph defined through a set \( N \) of nodes and a set \( E \) of directed edges. The set \( N \) includes a node for representing each program statement, one of which is the entry node, and there is a single exit node too, such that each node is both reachable from the entry and can reach the exit, through directed paths. Each edge \((m, n) \in E\) represents the direct flow of control from its source \( m \) to its target \( n \). Each node is the source of at most two edges. The exit node has no successors; normal nodes have one successor and a predicate node—corresponding to a conditional or to a loop statement’s condition—has two successors. A CFG of the example code fragment comprises the set of nodes \( N = \{1, 2, 3, 4, 5, 6, 7\} \) with 1 and 7 as its entry and exit nodes, respectively, and the set of directed edges \( E = \{(1, 2), (1, 5), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7)\} \).

In addition to the CFG, the bucketing algorithms work on a representation of the program that includes control and data dependences [8]. Intuitively, a data or control dependence of a node \( n \) on another node \( m \) means that their order of execution is significant. Reversing that order such that \( n \) would be reached ahead of \( m \) may cause an unwanted change in the program’s results. The remainder of this section provides a formal definition of the relevant dependence relations.

**Definition 1** (Postdominance). A node \( n \) postdominates a node \( m \) in a program’s CFG iff every path from \( m \) to the exit includes \( n \).

In the example, node 4 postdominates nodes 2-4. It does not, however, postdominate node 1, due to the CFG path \((1, 5, 6, 7)\). Node 6, in turn, postdominates all nodes except 7.

**Definition 2** (Control Dependence). A CFG node \( n \) is control dependent on a CFG node \( p \) iff \( n \) postdominates a successor of \( p \), but \( n \) does not postdominate \( p \) itself.

In the example, nodes 2, 3, and 4 are control dependent on node 1 because they all postdominate 2 but not 1 itself.

The set of variables each CFG node \( n \) may modify is denoted \( \text{Def}(n) \), and the set of variables it refers to is \( \text{Use}(n) \). In our example code these sets can be easily inferred from the program text thanks to our assumption that the called methods incur no side effects and use only the parameters sent to them.

**Definition 3** (Flow Dependence). A CFG node \( n \) is flow dependent on a CFG node \( m \) iff \( m \) defines a variable \( v \) that is used in \( n \), i.e. \( v \in \text{Def}(m) \cap \text{Use}(n) \), and there exists a non-empty path from \( m \) to \( n \) in the CFG with no further definition of \( v \).

The variable \( x_4 \), for example, is defined in node 4 and used in nodes 6 and 7; as there is a (re-)definition of \( x_4 \) in node 6 but not in node 5, node 6 is flow dependent on node 4, but node 7 is not; instead, 7 is flow dependent on node 6.

**Definition 4** (Anti Dependence). A CFG node \( n \) is anti dependent on a CFG node \( m \) iff \( m \) uses a variable \( v \) that is defined in \( n \), i.e. \( v \in \text{Use}(m) \cap \text{Def}(n) \), and there exists a non-empty path from \( m \) to \( n \) in the CFG with no other definition of \( v \).

The value of variable \( x_1 \) is used in node 1 and defined in node 3; the existence of the control-flow path \((1, 2, 3)\) with no
the flow dependence of nodes is control dependent on gray represent cloned statements—designated for extraction.

Definition 5 (Output Dependence). A CFG node \( n \) is output dependent on a CFG node \( m \) iff the set of variables defined in \( m \) includes at least one variable \( v \) that is also included in the set of variables defined in \( n \), and there is a non-empty path from \( m \) to \( n \) in the CFG with no other definition of \( v \).

Regardless of the flow dependence of node 6 on node 4, as stated above, we note that node 6 is also output dependent on node 4, again due to variable \( x4 \).

Definition 6. A Program dependence graph (PDG) corresponding to a given program’s CFG is a directed graph having the same set of nodes as in the CFG and having two sets of edges \( E_C, E_D \). There exists an edge \((p, n) ∈ E_C\) iff \( n \) is control dependent on \( p \). Similarly, there exists an edge \((m, n) ∈ E_D\) iff \( n \) is data dependent on \( m \).

The PDG of the example code fragment is depicted in Fig. 3. The data-dependence edge (curved arrow) directed from node 1 to node 3 reflects the anti dependence of 3 on 1. Similarly, the edges from node 2 to nodes 3 and 4 stand for the flow dependence of nodes 3 and 4 on node 2, whereas the edge from 6 to 7 is due to each of the three kinds of data dependence: flow, anti, and output. Finally, the edges from 1 to nodes 2-4 (in straight arrows) are due to the control dependence of those nodes on the predicate of node 1.

Definition 7. A control-dependence leaf, or simply control leaf, is a PDG node with no control-dependence successors.

The control leaves in the example PDG (Fig. 3) are nodes 2-7. In bucketing, each control-leaf node is designated into a single bucket, whereas non-leaf nodes may be duplicated in multiple buckets.

Definition 8. The slide associated with a PDG node \( n \), denoted slide(\( n \)), is a set of nodes containing \( n \) and all its control-dependence ancestors.

The slide of node 3 in the example of Fig. 1 includes the nodes 1 and 3. It is important to note that when a control leaf is assigned into a bucket, it is guaranteed according to KH that (at least a copy of) all members of its slide will be included in the code fragment generated for that bucket. Accordingly, in our approach to bucketing, we consider the slide of each control leaf as an atomic entity. (A comprehensive treatment of slides as executable subprograms—rather than sets of statements—in a program representation of non-contiguous code can be found in the first author’s doctoral thesis [4, Chapter 8].)

Definition 9. There is a slide dependence between two nodes \( m, n \) of a PDG with nodes \( N \) and edges \( E_C, E_D \), if both \( m \) and \( n \) are control-leaf nodes and there exists a data-dependence edge \((m′, n′) ∈ E_D\) with \( m′ \in \text{slide}(m) \) and \( n′ \in \text{slide}(n) \). Node \( n \) is said to be slide dependent on node \( m \).

Node 3 in the example PDG (Fig. 3) is, for instance, slide-dependent on node 4, since node 1 ∈ slide(4), 3 ∈ slide(3), and \((1, 3) ∈ E_D \).

Definition 10. A slide dependence graph (SlideDG) corresponding to a PDG with nodes \( N \) is a directed graph that includes all control-leaf nodes in \( N \) and a slide-dependence edge directed from \( m \) to \( n \) iff \( n \) is slide dependent on \( m \).

The SlideDG representation of our example code is given in Fig. 4. (As in the PDG, all nodes of the clone are marked in gray in Fig. 4.) Note that node 1 is not included in the graph, as it is not a control leaf. The significance of a non-empty directed path from control-leaf node \( m \) to another control-leaf node, \( n \), is that semantics might not be preserved by reordering their slides. That is, assigning \( m \) to a later bucket than that of \( n \) may lead to a change in dataflow, and therefore to an unwanted change in the semantics of the program.

Slide node 2 will be forced, by bucketing, into the before bucket, as it reaches (for example) node 4 through directed.
path of length 1 in this graph; slide node 5 will be forced into the after bucket, being reachable from the marked node 4; and 3 will be promoted as it is both reaching (7) and reachable (4). The removal of 7 from the set of marked nodes, and hence its removal from the scope of the bucketing transformation, would result in sending node 3 into the after bucket.

III. SLIDE-BASED BUCKETING

A set $M$ of CFG nodes corresponds to code that can be extracted into a method if it contains a single entry node and all exits from $M$ to nodes outside $M$ end in the same node. Such a set, known as a hammock, corresponds to a single-entry-single-outside-exit region of the CFG [13].

For the transformed program to yield the same results as the original on any input, the goal of a 3-bucketing algorithm is to identify three subsets of the original program corresponding to three subsets of the PDG nodes, such that all data and control dependences are preserved. The preservation of control dependences is expected inside each of the three subsets, whereas data dependences should be preserved both in and among the three parts.

Preservation of control dependences inside each bucket is enabled through duplication of conditionals and jumps. The placement of a node into a bucket implies the inclusion of (at least a copy of) all its control-dependence ancestors in the corresponding subprogram—before, marked, or after. Preservation of data dependences is achieved by the generation of ordering constraints on the placement of pairs of nodes and by requiring the algorithm to identify a node partitioning that does not violate any such constraint [15], [13].

Given a set of marked nodes $M$ and a SlideDG representation of the tightest hammock containing $M$ in the original program, the sets of nodes that reach any member of $M$ and the set of nodes reachable from $M$—to which we refer as reaching-$M$ and $M$-reachable, respectively—could be collected through two linear-time traversals, each starting from nodes $M$. However, this would require the construction of the complete SlideDG. It turns out, unfortunately, that this construction takes cubic time, in the worst case, in the number of PDG nodes. This is so since for each of the $O(|N|)$ control-leaf nodes, we need to traverse $O(|N|)$ control-dependence ancestors, and for each such ancestor we should traverse $O(|N|)$ data-dependence successors (or predecessors). Instead, we compute the two required closures using a search of the PDG nodes. This traversal of the complete SlideDG. It turns out, unfortunately, that this construction takes cubic time, in the worst case, in the number of PDG nodes. This is so since for each of the $O(|N|)$ control-leaf nodes, we need to traverse $O(|N|)$ control-dependence ancestors, and for each such ancestor we should traverse $O(|N|)$ data-dependence successors (or predecessors). Instead, we compute the two required closures using a search of the PDG, in which each edge is traversed at most once in each direction. This approach yields a linear-time algorithm, in the size of the PDG, and hence quadratic-time in the number of its nodes in the worst case.

Our proposed algorithm takes as input a PDG with nodes $N$ and edges $E_C,E_D$ and a set of nodes marked for extraction, $M \subseteq N$. It computes three subsets of the nodes $N$ corresponding to control leaves forced into the before, marked, and after buckets. The remaining control-leaf nodes, taken together, can be arbitrarily added to either the before or the after bucket.

The algorithm, shown in Fig. 5, consists of a sequence of five steps. Following our observation on the significance of slide-dependence paths, the first two steps collect two sets of control-leaf nodes: $M$-reachable and reaching-$M$. The former set includes all leaf nodes in the reflexive-transitive closure of $M$ with respect to the slide-dependence relation, whereas the latter set involves the same closure in the inverse relation. Placing any member of $M$-reachable in the before bucket, or any member of reaching-$M$ in the after bucket, would violate at least one ordering constraint. In our example, $M = \{4,6,7\}$, the resulting sets $M$-reachable and reaching-$M$ are $M \cup \{3,5\}$ and $\{2,3\} \cup M$, respectively.

The remaining three steps (lines 3-5 of the algorithm) assign slides to buckets. Any node that both reaches $M$ and is reachable from $M$ is designated for extraction by being included in the set marked (line 3). In Komondoor’s terminology, the nodes in this bucket that were not initially marked, i.e. marked \ $M$, are being promoted. Nodes that reach $M$ and were not designated for extraction are forced into the before bucket (line 4), while non-reaching nodes that are reachable from $M$ are forced into the after bucket (line 5). Returning to our example, the expected results are: before= $\{2\}$, marked= $\{3,4,6,7\}$, and after= $\{5\}$.

The remaining slides, if any, were not forced into any bucket. Komondoor’s original approach is to force such nodes non-deterministically, one at a time, into either the before or after buckets, and then identify all other nodes that must be placed in that same bucket. The same approach could be taken here too, if desired, by following slide-dependence paths starting at (ending in) nodes placed in the after (before) bucket.

The PDG-based search is essentially a demand-driven algorithm that visits relevant slide-dependence successors at each step.$^1$ Following the definition of slide dependence (Definition 9 above), there are three steps in a visit of a leaf-node $m$: (a) collecting all newly-visited nodes in its slide (line 15 on Fig. 5), (b) for each such node, $m'$, search all its data-dependence successors (line 18) or predecessors (line 21, for the backward search), and finally (c) from each such successor $n$ or predecessor $l$, complete the search by finding all leaf nodes whose slide contains $n$ (line 19) or $l$ (line 22).

To keep the search linear in the size of the PDG, steps (a) and (c) collect only newly visited nodes. As step (b) will be reached at most once for any node, the search will traverse each data-dependence edge at most once. Each control-dependence edge $(p,n) \in E_C$, in turn, may be traversed at most once in each direction: on the way up, in step (a), for finding newly-visited PDG nodes in the slide of $n$ (lines 15 and 24-29 in the recursive visit-slide-of function), and on the way down, in step (c), to search for slides that $p$ is included in (lines 19/22 and 30-37 in the recursive search-slides-with).

The search algorithm—lines 7-13 on Fig. 5—maintains a worklist of slides (i.e. control-leaf PDG nodes) still to be visited, a global set of reached slides, and two global sets of visited PDG nodes, visited-up and visited-down, for those nodes whose control-dependence predecessors or successors were considered, respectively. This traversal of the PDG

$^1$We owe this observation to Aharon Abadi.
slide-based-bucketing(N,E,C,D,M)
1: M-reachable := slides-first-search(N,E,C,D,M,”forward")
2: reaching-M := slides-first-search(N,E,C,D,M,”backward")
3: marked := reaching-M \ M-reachable
4: before := reaching-M \ M-reachable
5: after := M-reachable \ reaching-M
6: return (before,marked,after)

slides-first-search(N,E,C,D,M,direction)
7: initialize visited-up and visited-down to Ø
8: initialize worklist and reached to M
9: while worklist is not empty do
10: take the first node m out of the worklist
11: newly-visited-slides := slide-dependence-search(m)
12: add all newly-visited-slides to reached and to the worklist
13: return reached

slide-dependence-search(m)
14: initialize newly-reached-slides to Ø
15: newly-visited-slide-nodes := visit-slide-of(m)
16: forall m’ ∈ newly-visited-slide-nodes do
17: if the direction is "forward" then
18: forall n ∈ N such that (m’,n) ∈ E_D do
19: newly-reached-slides := newly-reached-slides ∪ search-slides-with(n)
20: else (i.e. the direction is "backward")
21: forall l ∈ N such that (l,m’) ∈ E_D do
22: newly-reached-slides := newly-reached-slides ∪ search-slides-with(l)
23: return newly-reached-slides

visit-slide-of(n)
24: initialize the local set of newly-visited nodes to Ø
25: if n ∉ visited-up then
26: add n to visited-up and to newly-visited
27: forall p ∈ N such that (p,n) ∈ E_C do
28: newly-visited := newly-visited ∪ visit-slide-of(p)
29: return newly-visited

search-slides-with(m)
30: initialize the local set of newly-reached nodes to Ø
31: if m ∉ visited-down then
32: add m to visited-down
33: if there exists no n ∈ N such that (m,n) ∈ E_C then
34: if m ∉ reached add m to newly-reached
35: else forall n ∈ N such that (m,n) ∈ E_C do
36: newly-reached := newly-reached ∪ search-slides-with(n)
37: return newly-reached

IV. EQUIVALENCE WITH NODE-BASED BUCKETING

In this section we provide the details of Komondoor’s original algorithm [13], to which we refer as node-based bucketing. We then prove that the results of our slide-based re-formulation are equivalent to those of the node-based approach—thus providing the same correctness guarantees (recalling that Komondoor has proved semantics preservation of his algorithm [13, Appendix B]).

The node-based bucketing algorithm takes as input a set of CFG nodes designated for extraction (the marked nodes), a subgraph of the program’s CFG (the tightest hammock containing all marked nodes), the sets of used and defined variables in each CFG node, and the control dependences corresponding to the given CFG. The algorithm’s output is a partition of the hammock’s nodes into three sets (buckets): before, marked, and after. The algorithm involves the following three steps: (1) generate ordering constraints on the hammock’s nodes; (2) promote unmovable nodes, adding them to the set of marked nodes; and (3) partition the hammock nodes into three buckets—before, marked, and after.

A post processing step in a refactoring tool would then transform the program fragment corresponding to the selected hammock. It would replace the fragment with a sequence of three sub-fragments, corresponding to the before, marked, and after buckets. In preparation of those fragments, the tool must first complete the before and after buckets by adding copies of all control-dependence ancestors of nodes in those buckets, for the predicates and jumps that must be duplicated. The marked bucket does not require this post-processing step, as such nodes are added to it in the promotion step.

Base Ordering Constraints. There exist two kinds of base ordering constraints, as follows. (i) Data-dependence constraints: For each pair of nodes m,n in the hammock such that n is data (flow, anti, or output) dependent on m, i.e. there is a data-dependence edge (m,n) in the hammock’s PDG, generate the constraint m ≤ n. This means that (a copy of) m must not be placed in any bucket that follows that bucket that contains (a copy of) n. Recall that the order of the buckets is before < marked < after. (ii) Control-dependence constraints: For each node n in the hammock, and for each predicate or jump p in the hammock such that n is (directly or transitively) control dependent on p in the CFG, generate a constraint n ⇒ p. This means that (a copy of) p must be present in each bucket that contains (a copy of) n.

The base data-dependence constraints of the example code correspond to the curved arrows in Fig. 3. Base control-dependence constraints, in turn, correspond to non-empty paths of straight arrows in this diagram. The path from node 1 to nodes 2, 3, and 4 causes generation of the following three constraints: 2 ⇒ 1, 3 ⇒ 1, and 4 ⇒ 1, as shown in Table I.

Extended Ordering Constraints. The following three rules are applied repeatedly, until no more extended constraints can be generated.

Fig. 5: Slide-based bucketing algorithm
TABLE I: Base and extended ordering constraints.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>⇒</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
</tr>
<tr>
<td>2</td>
<td>⇒</td>
<td>≤1 (3)</td>
<td>≤1 (3)</td>
<td>≤1 (3)</td>
<td>≤1 (3)</td>
<td>≤1 (3)</td>
<td>≤1 (3)</td>
</tr>
<tr>
<td>3</td>
<td>⇒</td>
<td>≤1 (3)</td>
<td>≤1 (3)</td>
<td>≤1 (3)</td>
<td>≤1 (3)</td>
<td>≤1 (3)</td>
<td>≤1 (3)</td>
</tr>
<tr>
<td>4</td>
<td>⇒</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
</tr>
<tr>
<td>5</td>
<td>⇒</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
</tr>
<tr>
<td>6</td>
<td>⇒</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
</tr>
<tr>
<td>7</td>
<td>⇒</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
<td>≤0</td>
</tr>
</tbody>
</table>

Extension Rule 1 (Transitivity):

\[ a \leq b, b \leq c \Rightarrow a \leq c. \]

Extension Rule 2 (Left Predicate Substitution):

\[ a \Rightarrow p, p \leq b \Leftrightarrow a \leq b. \]

Extension Rule 3 (Right Predicate Substitution):

\[ a \leq p, b \Rightarrow p \Leftrightarrow a \leq b. \]

We define the degree of an ordering constraint as follows.

Definition 11. The smallest number of applications of Extension Rule 1 (transitivity) for forming an ordering constraint \( m \leq n \) is denoted \( m \leq_k n \) and referred to as the degree of the constraint.

The degree of constraints will help us in illustrating the meaning of the ordering constraints. (It will also be used later in this section for proving the equivalence of our new slide-based algorithm to the node-based one.)

The following two rules are applied repeatedly, in any order, until no more nodes can be promoted. (i) If there exist constraints \( m_1 \leq n \) and \( n \leq m_2 \), where \( n \) is unmarked and \( m_1, m_2 \) are marked, promote \( n \). (ii) If there exists a constraint \( m \Rightarrow p \) such that \( m \) is marked and \( p \) is unmarked, promote \( p \).

Step 2. A Procedure for Promoting Nodes.

The following two rules are applied repeatedly, in any order, until no more nodes can be promoted. (i) If there exist constraints \( m_1 \leq n \) and \( n \leq m_2 \), where \( n \) is unmarked and \( m_1, m_2 \) are marked, promote \( n \). (ii) If there exists a constraint \( m \Rightarrow p \) such that \( m \) is marked and \( p \) is unmarked, promote \( p \).

A promoted node is regarded as marked as soon as it is promoted. In the extraction of the clone in Fig. 1, node 3 is promoted: as can be seen in Table I, choosing \( m_1 \) to be node 4 and \( m_2 \) to be node 7, with both 4 and 7 marked, we get the promotion of node 3 due to the ordering constraints \( 4 \leq 3 \leq 7 \). Note that had we not marked node 7, node 3 would not have been promoted; instead it would have correctly been placed in the \( \text{after} \) bucket—and therefore moved after the statement of node 6, yet still before the statement of node 7, thus preserving the data dependence of 7 on 3.

Step 3. Assignment of Nodes to Buckets.

A node \( r \) is forced into the \( \text{before} \) bucket if \( r \) is not in the \( \text{marked} \) bucket and any of the following two conditions hold. (i) There exists a constraint \( r \leq b \), where \( b \) is a node in the \( \text{before} \) bucket. (ii) There exists a constraint \( r \leq m \), where \( m \) is a node in the \( \text{marked} \) bucket. Node 2 in Fig. 1 is forced into the \( \text{before} \) bucket due to the base data-dependence constraint \( 2 \leq 4 \), resulting from the definition of variable \( \times 2 \) in node 2 and its subsequent use in the marked node 4. Note that the placement of node 2 in the \( \text{before} \) bucket causes a duplication of the predicate from node 1 (Fig. 2).

A node \( s \) is forced into the \( \text{after} \) bucket if any of the following two conditions hold. (i) There exists a constraint \( a \leq s \), where \( a \) is a node in the \( \text{after} \) bucket. (ii) There exists a constraint \( m \leq s \), where \( m \) is a node in the \( \text{marked} \) bucket. In our example, node 5 is forced to the \( \text{after} \) bucket due to the constraint \( 4 \leq 5 \), where \( 4 \) is a marked node.

The cubic time complexity of that original algorithm is caused by Step 1, in the generation of extended constraints. Had those extended constraints been amenable for trivial expression as paths on the PDG, we would have no longer needed to express them as an explicit data structure; in such a case we would be able to avoid this data structure’s cubic-time construction, while potentially reducing the associated space too. (This is a key strength of graph-based program representations, making them so useful for slicing [8], chopping [11], and other analyses [20].) Unfortunately, it is not the case that there exists a directed path from a node \( m \) to a node \( n \) in the PDG iff \( m \leq n \). For example, whereas there exists a path from node 1 to node 2 (Fig. 3), there is no constraint \( 1 \leq 2 \) (Table I); and although \( 4 \leq 3 \), there exists no directed path from node 4 to 3. The reduced complexity of our proposed approach has been achieved by employing for the problem in hand a different kind of a directed graph—the slide dependence graph. As will be proved shortly, the relation underlying this graph supports the desired property: there is a directed path in that graph iff there exists a constraint.

The following lemma relates slide dependence with ordering constraints of degree-0 (Definition 11). Recall that such degree-0 constraints are generated by base data-dependence constraints followed by Extension Rules 2 and 3, but with no application of Extension Rule 1 (transitivity).

Lemma 12. Let \( N \) and \( E_C, E_D \) be the nodes and edges of a given PDG, respectively, and let \( m, n \in N \) be two control-leaf nodes. An ordering constraint \( m \leq_0 n \) relates the two nodes if and only if \( n \) is slide dependent on \( m \).

Proof. (LHS \( \Rightarrow \) RHS) There are essentially three ways to form an ordering constraint \( m \leq_0 n \) between two control-dependence leaves \( m, n \in N \). In the first, \( n \) is directly data
dependent on $m$ and therefore also slide dependent on it. The second way is to apply Extension Rule 2 on a base constraint $p \leq n$, provided $m \Rightarrow p$. In this case, since $p$ is a control-dependence ancestor of $m$, we have $p \in \text{slide}(m)$ and therefore $n$ is slide dependent on $m$. The third way is to apply Extension Rule 3 on a base constraint $m \leq p$, when $n \Rightarrow p$. In this case, the slide dependence of $n$ on $m$ is due to $p$ being a control-dependence ancestor of $n$, such that $p \in \text{slide}(n)$.

Note that further applications of these two Extension Rules would yield no further constraints. This is due to the definition of $\Rightarrow$ relating a node to all its (direct or transitive) control-dependence ancestors. Note also that an application of the two Extension Rules on both sides of a base constraint $p \leq q$ would require both sides to be predicates, and such base constraints cannot exist in the absence of side effects in predicates. (The absence of side effects in predicates is assumed by Komondoor in order to ensure duplication of predicates preserves the original semantics.)

(LHS $\Leftarrow$ RHS) The slide dependence of $n$ on $m$ is caused by a data dependence of node $m'$ on node $m'$, with $m' \in \text{slide}(m)$ and $n' \in \text{slide}(n)$. In terms of ordering constraints, this directly translates to a base constraint $m' \leq n'$. Being in $\text{slide}(m)$, $m'$ must be either $m$ itself or one of its control-dependence ancestors, such that $m \Rightarrow m'$. In both cases we get $m \leq n$ from $m' \leq n$, by substitution in the former case, or as an extended constraint through Extension Rule 2. Similarly, since $n' \in \text{slide}(n)$, we get the required $m \leq n$ from $m \leq n'$ either by substitution, when $n = n'$, or by Extension Rule 3, as in this case $n \Rightarrow n'$.

This lemma is next used in proving the equivalence of the two bucketing approaches. The following theorem relates paths in the SlideDG to Komondoor’s ordering constraints, with no restriction on the degree of a constraint.

Theorem 13. Let $N$ and $E_C, E_D$ be the nodes and edges of a given PDG, let $E_S$ be the corresponding set of slide-dependence edges, and let $m, n \in N$ be two control-leaf nodes. Provided each predicate is the control-dependence ancestor of at least one control leaf, a (possibly extended) ordering constraint $m \leq n$ exists if and only if there exists a directed path of slide-dependence edges (in $E_S$) from $m$ to $n$.

Proof. (LHS $\Rightarrow$ RHS) For this side of the proof we use induction on the degree of the ordering constraint. In the base case, $m \leq n$, there exists a direct slide-dependence edge from $m$ to $n$, as shown by Lemma 12. For the inductive step, assuming the existence of a slide-dependence path of any two nodes related through an ordering constraint of degree $k$, we show the existence of such a path between any two nodes related through an ordering constraint of degree $k+1$. Focusing on the first application of transitivity (i.e., Extension Rule 1), one that contributes to the degree being $k+1$, we note that $m \leq k+1$ is either due to (a) $m \leq m' \leq n$, for some control leaf $m' \in N$, or to (b) $m \leq p \leq n$, for some non-control-leaf predicate node $p \in N$.

Being a control leaf, $m'$ in (a) above is slide dependent on $m$ due to Lemma 12. According to the induction hypothesis, a slide-dependence path exists from $m'$ to $n$. This path, prefixed by the direct edge from $m$ to $m'$, forms the expected path from $m$ to $n$.

For (b) above, we note that since any control-dependence descendant $m'' \in N$ of $p$ forms the constraint $m'' \Rightarrow p$, such $m''$ can substitute $p$ in the extended ordering constraint, yielding $m \leq m'' \leq n$. This is due to a combination of Extension Rule 2 on the $k$-degree ordering constraint and of Extension Rule 3 on the 0-degree side. Such $m''$ could be any control-leaf descendant of $p$, guaranteeing the existence of a slide-dependence path from $m$ to $n$ as in the case of $m$ in (a) above. The proviso, that each predicate node is the ancestor of at least one control leaf, completes this part of the proof.

(LHS $\Leftarrow$ RHS) The induction this time is on the length of the slide-dependence path from $m$ to $n$. Assuming each two control leaves $m, n \in N$ with a directed path of $k$ slide-dependence edges from $m$ to $n$ form the ordering constraint $m \leq n$, we show any two such nodes with a $k+1$ long path form such an ordering constraint too.

Again, the base case (for $k = 1$ this time) follows from Lemma 12, in the reverse direction. For the inductive step ($k+1$), we consider a decomposition of the given slide-dependence path as a first edge from $m$ to a control-leaf node $m' \in N$, followed by a $k$-long path from $m'$ to $n$. Again, $m \leq m'$ is given by Lemma 12, whereas the induction hypothesis ensures $m' \leq n$. Hence, the transitivity of the ordering constraints (i.e., Extension Rule 1) provides the required $m \leq n$.

The proviso in the theorem above is added since in the presence of a control-dependence cycle with no control leaf being control dependent on any node in the cycle, the nodes on that cycle will not contribute to any slide dependence, as they will not be members of any slide. In contrast, ordering constraints involving the cycle may exist. Such constraints may cause the generation of extended constraints with no corresponding path of slide-dependence edges. This potential weakness of the SlideDG can be easily avoided by considering strongly connected components in the control-dependence subgraph of the PDG, effectively collapsing such cycles, in the PDG-based definition of slides and control leaves. E.g., “while(f1()) { if (f2()) { break; } }” is a code leading to such a cycle. Beyond the evident control-dependence edges from the $f1()$ predicate to the $f2()$ predicate, and from the latter to the break, the cycle is completed by the control dependence of the $f1()$ on the break.\footnote{Komondoor’s bucketing treats jump statements as pseudo predicates, following Ball and Horwitz [2]—with the jump’s target as its true control-flow successor and the node to which the jump would fall through, if it were a “no-operation”, as the dummy (non-executable) false successor.}

The equivalence of results, for the promotion and partition steps of bucketing, is shown in the following.

Theorem 14. Let $N$ and $E_C, E_D$ be the nodes and edges of a given PDG and let $M \subseteq N$ be any subset of control-
leaf nodes marked for extraction. Provided each predicate is the control-dependence ancestor of at least one control leaf, any control-leaf node is assigned to the same bucket by both algorithms.

Proof. Let \( (before, marked, after) \) be the result of the slide-based bucketing algorithm and let \( (before', marked', after') \) be the result of the original node-based algorithm. Recall that a control-leaf node \( m \) is promoted by the node-based algorithm iff there exist two marked nodes \( l, n \) such that \( l \leq m \leq n \). In terms of slide-dependence edges, this is equivalent to the existence of a directed path from \( l \) to \( n \) that includes \( m \), according to Theorem 13. The existence of such a path is equivalent to \( m \) being both reachable from \( l \) and reaching \( n \), in the slides-first search of Fig. 5. This proves that a control-leaf node \( n \) is promoted by the node-based algorithm iff it is promoted by the slide-based algorithm.

For the \( before \) bucket, recall that in node-based bucketing a control-leaf node \( m \) is forced into that bucket iff there exists a node \( n \in N \) such that \( m \leq n \) and \( n \) is either in the \( before \) or \( marked \) buckets. This, in turn, is equivalent to stating that node \( n \) is in the set of initially marked nodes, since (a) any control-leaf node \( n' \) added to the \( before \) or \( marked \) buckets is guaranteed to have \( n' \leq n \) hold for some initially marked node \( n \), and (b) any predicate node \( p \) that forces \( m \) to the \( before \) bucket with no initially marked node \( n \) such that \( p \leq n \) is guaranteed to be a control-dependence ancestor of at least one such node \( n \), ensuring \( m \leq n \) too, through Extension Rule 3.

According to Theorem 13, the existence of such a control leaf \( n \) with \( m \leq n \) is equivalent to \( m \) reaching the set of marked nodes through a slide-dependence path. Moreover, as \( m \) was not promoted, there exists no marked control-leaf node \( l \) such that \( l \leq m \). Again, according to Theorem 13, the absence of such a node \( l \) is equivalent to \( m \) not being reachable from the set of marked nodes through a slide-dependence path.

The proof is completed by noticing that a similar argument in the reverse order (of slide-dependence edges, not control-dependence ones) would show that a control-leaf node \( n \) is forced into the \( after \) bucket by the node-based algorithm (ahead of the assignment of buckets to arbitrary nodes) iff \( n \) is reachable from marked node \( m \) through a slide-dependence path, and it is not reaching any marked node through any such path.

In the presence of unstructured jumps—as in \texttt{break}, \texttt{continue}, or \texttt{return} statements—the tightest hammock containing a given set of statements might be too large for effective extraction, potentially leading to excessive promotion of loosely related code. To successfully extract such code fragments, Komondoor and Horwitz define a hammock with exiting jumps, or \textit{e-hammock}, as a set of CFG nodes that by replacing exiting jumps with no-ops would form a hammock—\textit{i.e.} replacing their CFG node’s outgoing edge with an edge to the fall-through node [15]. In the code of Fig. 6, for example, whereas the tightest hammock containing statements 17-19 includes statements 2-45, the tightest e-hammock comprises statements 4-19. The undesired promotion of unmarked statements in the presence of exiting jumps can be avoided by restricting the transformation to the code fragment corresponding to the tightest e-hammock containing the marked statements. For preservation of semantics, this is complemented by the addition of compensatory code: a new variable is added, to hold an exit code from the extracted method; upon returning from the extracted method, the refactored version would test the exit code and jump accordingly. In this paper we have elected to focus on the subset of the original transformation that involves no exiting jumps. The original algorithm, in turn, includes further steps and rules for the generation of ordering constraints to handle exiting jumps correctly. We conjecture that this treatment of exiting jumps, if desirable, could be equivalently replaced by two complementary code transformations ahead and following the bucketing transformation. Ahead of bucketing, the exiting jumps would be replaced by non-exiting ones, whereas after bucketing these jumps would be replaced again with the original ones. This way, there would be no need to adapt the slide-based bucketing algorithm for this particular case of exiting jumps. A formal development of such a solution is left however for future work.

V. Conclusion

We have presented in this paper a new algorithm for the preparation of non-contiguous code for extraction into a new method. The algorithm, inspired by Komondoor’s approach, is proved to yield identical results to that original algorithm, faster. We have also discussed a number of optimization problems whose solution could contribute to the automation of type-3 clone elimination. We have demonstrated by example how such automation might require a massive number of runs of a bucketing algorithm, before an optimal combination can be found. The discussed optimization problems act as the main motivation for improving the efficiency of the original bucketing algorithm.

The improved efficiency of our slide-based approach to bucketing has been the result of our choice to consider each primitive statement along with its controlling predicates—\textit{i.e.} the slide of that statement—as an atomic entity in the analysis phase. The notion of slide dependence, combining control and data dependences, has proved effective in re-implementing the partition of statements into buckets (and the promotion of unmovable statements) through bidirectional reachability on the PDG.

These notions of slides, slide dependence, and possibly the slide-dependence graph too, are expected to be amenable for a re-formulation of all method-extraction algorithms that preserve syntax and control flow (\textit{e.g.} [17], [18], [7], [4], [23], [24], [5]). Such a re-formulation is likely to improve the efficiency of implementations of these existing algorithms. Being based on a common foundation, it will be interesting to investigate how these formulations may act as a basis for both theoretical and practical comparisons of existing algorithms—hopefully inspiring the development of new ones too.
public void generateOptimizedLogicalAnd(BlockScope currentScope, CodeStream codeStream, Label trueLabel, Label falseLabel, boolean valueRequired) {
    int pc = codeStream.position;
    Constant condConst;
    if ((left.implicitConversion & 0xF) == T_boolean) {
        condConst = left.conditionalConstant();
        if (condConst != NotAConstant) {
            if (condConst.booleanValue() == true) {
                left.generateOptimizedBoolean(currentScope, codeStream, trueLabel, falseLabel, false);
                if ((bits & OnlyValueRequiredMASK) != 0) {
                    right.generateCode(currentScope, codeStream, valueRequired);
                } else {
                    right.generateOptimizedBoolean(currentScope, codeStream, trueLabel, falseLabel, valueRequired);
                }
            } else {
                left.generateOptimizedBoolean(currentScope, codeStream, trueLabel, falseLabel, false);
                right.generateOptimizedBoolean(currentScope, codeStream, trueLabel, falseLabel, false);
                if (valueRequired) {
                    if ((bits & OnlyValueRequiredMASK) != 0) {
                        codeStream.iconst_0();
                    } else {
                        if (falseLabel != null) {
                            codeStream.goto_(falseLabel); }
                    }
                } else {
                    if (falseLabel != null) {
                        codeStream.goto_(falseLabel); }
                }
            }
        } else {
            left.generateOptimizedBoolean(currentScope, codeStream, trueLabel, falseLabel, false);
            right.generateOptimizedBoolean(currentScope, codeStream, trueLabel, falseLabel, false);
            if (valueRequired) {
                if ((bits & OnlyValueRequiredMASK) != 0) {
                    codeStream.iconst_0();
                } else {
                    if (falseLabel != null) {
                        codeStream.goto_(falseLabel); }
                }
            } else {
                if (falseLabel != null) {
                    codeStream.goto_(falseLabel); }
            }
        }
    } else {  
        condConst = right.conditionalConstant();
        if (condConst != NotAConstant) {
            if (condConst.booleanValue() == true) {
                if ((bits & OnlyValueRequiredMASK) != 0) {
                    left.generateCode(currentScope, codeStream, valueRequired);
                } else {
                    left.generateOptimizedBoolean(currentScope, codeStream, trueLabel, falseLabel, valueRequired);
                }
                right.generateOptimizedBoolean(currentScope, codeStream, trueLabel, falseLabel, false);
                if (valueRequired) {
                    if ((bits & OnlyValueRequiredMASK) != 0) {
                        codeStream.iconst_0();
                    } else {
                        if (falseLabel != null) {
                            codeStream.goto_(falseLabel); }
                    }
                } else {
                    if (falseLabel != null) {
                        codeStream.goto_(falseLabel); }
                }
            } else {
                left.generateOptimizedBoolean(currentScope, codeStream, trueLabel, falseLabel, false);
                right.generateOptimizedBoolean(currentScope, codeStream, trueLabel, falseLabel, false);
                if (valueRequired) {
                    if ((bits & OnlyValueRequiredMASK) != 0) {
                        codeStream.iconst_0();
                    } else {
                        if (falseLabel != null) {
                            codeStream.goto_(falseLabel); }
                    }
                } else {
                    if (falseLabel != null) {
                        codeStream.goto_(falseLabel); }
                }
            }
        } else {
            left.generateOptimizedBoolean(currentScope, codeStream, trueLabel, falseLabel, false);
            right.generateOptimizedBoolean(currentScope, codeStream, trueLabel, falseLabel, false);
            if (valueRequired) {
                if ((bits & OnlyValueRequiredMASK) != 0) {
                    codeStream.iconst_0();
                } else {
                    if (falseLabel != null) {
                        codeStream.goto_(falseLabel); }
                }
            } else {
                if (falseLabel != null) {
                    codeStream.goto_(falseLabel); }
            }
        }
    }
    codeStream.recordPositionsFrom(pc, this.sourceStart);
    return;
}

Fig. 6: Left side of the clone pair of case 23 from the Tiarks benchmark: column C1 marks those statements that have an identical counterpart in the other clone instance. The maximal non-promoting subset of C1 is given in column C2.
REFERENCES


