Effective, Automatic Procedure Extraction*

Raghavan Komondoor Susan Horwitz
University of Wisconsin-Madison
1210 W. Dayton St, Madison, WI 53706 USA.
{raghavan,horwitz}@cs.wisc.edu

Abstract

Legacy code can often be made more understandable and maintainable by extracting out selected sets of statements to form procedures and replacing the extracted code with procedure calls. Sets of statements that are non-contiguous and/or include non-local jumps (caused by gotos, breaks, continues, etc.) can be difficult to extract, and usually cause previous automatic-extraction algorithms to fail or to produce poor results.

The chief contributions of this paper are a semantics-preserving algorithm for extracting "difficult" sets of statements, and a study that compares the algorithm both to an ideal extractor (a human) and to previously reported automatic approaches. We found that "difficult" examples do arise frequently in practice, and that our algorithm is a significant improvement over previous work, achieving ideal results on over 70% of the difficult cases.

1. Introduction

The understandability and maintainability of legacy code can often be improved by extracting out selected sets of statements to form procedures (and replacing the extracted code with procedure calls). This operation is useful in several contexts:

- Duplicated code occurs frequently in real programs, as indicated by the results of several studies [2, 13, 16]. Replacing each instance of copied code with a procedure call makes the program easier to understand, as only one copy has to be read and understood. Also, maintenance becomes easier because updates and bug fixes need only be applied on one copy.
- Legacy programs often have large procedures that contain multiple strands of distinct computations; such strands often occur one after the other within the procedure, but it is not uncommon for them to be interleaved with each other [19, 21]. Extracting the individual strands into separate procedures aids program comprehension as each new procedure performs a single cohesive computation [9]. This activity also eases maintenance by localizing the effects of changes, and facilitates future code reuse [22, 18].
- Extracting embedded sets of statements that form conceptually independent operations can be an important part of the process of converting poorly designed, "monolithic" code to modular or object-oriented code.

Procedure extraction is a three-step process. First, the statements to be extracted are identified (we refer to them as the "marked" statements). Next, semantics-preserving transformations are applied if necessary to make the marked statements form a contiguous, well-structured block that is suitable for extraction. Finally, the marked statements are extracted into a new procedure, and replaced with a call.

The focus of this paper is the second step of procedure extraction. The first step can be performed either by the programmer, or by program analysis tools, such as [18, 5, 12]. The third step is essential, but the main issue (determining what the parameters to the new procedure should be and how they should be used), is straightforward (e.g., discussed in [10, 17]). The chief contributions of this paper are:

1. An algorithm for the second step of procedure extraction that handles "difficult" sets of marked statements: marked statements that are not contiguous and/or involve exiting jumps (jumps from within the region that contains the marked statements to outside that region). These features can increase the likelihood of errors in manual extraction; therefore, tool support is particularly important when extracting difficult sets of statements.
2. A study that compares our algorithm both to an ideal extraction (performed by us, using our best judgement) and to previously reported automatic approaches. We

*This work was supported in part by the National Science Foundation under grants CCR-9970707 and CCR-9987435, and by IBM.

found that "difficult" examples do arise frequently in practice, and that our algorithm is a significant improvement over previous work, achieving ideal results on over 70% of the difficult cases.

1.1. Motivating Example

The upper left column of Figure 1 contains a code fragment that serves as a motivating example. It consists of a loop that reads a sequence of arrays from a file. If the first element of the current array is greater than 100, the elements of the array are all set to their absolute values, the sum of the elements is computed, and variable `numSums` is incremented. The inner loop that computes the sum includes an overflow check; in that case, an error message is printed, and the processing is terminated via a `return`. If there is no overflow, variable `totalSum`, the sum of the values over all arrays, is updated.

The marked statements are indicated by the "++" signs. These statements, together with "`A[k] = abs(A[k])`", are the ones that compute the sum of the elements in the current array. Notice that the marked statements are interleaved with other (unmarked) statements that are not part of the sum computation.

The upper-right column of Figure 1 shows the output of our algorithm; the region that originally contained the marked statements (i.e., everything from the first marked statement to the last) has been transformed to make the marked code suitable for extraction. This example illustrates all four transformation techniques that our algorithm incorporates:

1. **Statement reordering:** As many unmarked statements as possible are moved out of the way to make the marked statements contiguous. In this example, the two statements "`read(fd, A, sizeof(int)*N)`" and "`numSums++`" are moved.

2. **Predicate duplication:** Moving the statement "`numSums++`" requires creating a copy of the predicate "`if (A[0] > 100)`".

3. **Promotion:** The unmarked statement "`A[k] = abs(A[k])`" cannot be moved out of the way without affecting semantics. Therefore it is *promoted* (i.e., marked), so that, as illustrated in the upper-right and lower-left columns, it will occur in the extracted procedure.

4. **Handling exiting jumps:** The marked `return` statement, which is an exiting jump, cannot simply be included in the extracted procedure with no other compensatory changes. Rather, the procedure sets a flag (the new global variable `exitKind`) to indicate whether the exiting jump must be executed after the procedure returns.

   In the code produced by the algorithm (shown in the upper-right column of the figure) the appropriate assignments to `exitKind` are included in the set of marked statements, a copy of the jump (in this case, a `return`), conditional on `exitKind` is added immediately after the marked statements, and the original exiting jump is converted to a `goto` to the new conditional statement. At the time of actual extraction the `goto` in the extracted procedure is converted into a `return`, as illustrated in the lower-left column.

   Other exiting jumps (caused by `breaks`, `continues` and `gotos`) are handled similarly, with `exitKind` set to a value that encodes the kind of jump.

1.2. Contributions over previous work

Previous work that is related to ours falls into two broad categories: eliminating `gotos` in source code [1, 20], and automatic procedure extraction. The problem solved in the first category (eliminating `gotos`) is different from ours; however, our technique of using the variable `exitKind` to handle exiting jumps bears some resemblance to their techniques. Within the second category, some of the techniques focus on compressing assembly code by detecting duplicated fragments and extracting them into procedures [24, 7, 8], while others concern procedure extraction in source code [10, 17, 11]. The main contributions of our work over previous work on automatic procedure extraction are that our algorithm handles exiting jumps, and uses a combination of transformation techniques. These two features enable our algorithm to succeed on many difficult inputs in practice.

The work of [10] is for Scheme programs, and thus does not address programs that contain jumps, whether they are exiting jumps or not. Other previous approaches to procedure extraction handle jumps, but not exiting jumps. For the example in Figure 1, the smallest exiting-jump-free region that contains the marked code is the entire outer `while` loop plus everything that follows this loop until the end of the procedure. [24, 7, 8, 11] would be able to extract this *entire* region, but not just the marked code shown in the figure. The approach of [17], which is discussed in detail in Section 4, would be able to extract the marked code, but would include duplicate copies of all but one of the marked statements after the call to the new procedure; this outcome is clearly undesirable, as the purpose of the extraction is defeated.

No single transformation technique (moving code, promotion, handling exiting jumps, duplicating predicates) is sufficient to handle all difficult cases. Previous approaches to automatic extraction either employ only a nar-

Original Fragment

```
   j = 0;
   while(j < NumArrays) {
++   sum = 0;
     read(fd, A, sizeof(int)*N);
++   if (A[0] > 100) {
       numSums++;
++     k = 0;
++     while (k < N) {
         A[k] = abs(A[k]);
++       if (MAXINT-A[k] < sum) {
++         print("overflow");
++         return;
++       }
++       sum += A[k];
++       k++;
++     }
++   }
     totalSum += sum;
     j++;
   }
```

Algorithm Output

```
     read(fd, A, sizeof(int)*N);
     if (A[0] > 100)
       numSums++;
++   sum = 0;
++   if (A[0] > 100) {
++     k = 0;
++     while (k < N) {
++       A[k] = abs(A[k]);
++       if (MAXINT-A[k] < sum) {
++         print("overflow");
++         exitKind = RETURN;
++         goto L;
++       }
++       sum += A[k];
++       k++;
++     }
++   }
++   exitKind = FALLTHRU;
  L: if (exitKind==RETURN) return;
```

Extracted Procedure

```
int doSum(int N, int A[],
          int *sumPtr) {
  *sumPtr = 0;
  if (A[0] > 100) {
    int k = 0;
    while (k < N) {
      A[k] = abs(A[k]);
      if (MAXINT-A[k] < (*sumPtr)){
        print("overflow");
        exitKind = RETURN;
        return;
      }
      *sumPtr += A[k];
      k++;
    }
  }
  exitKind = FALLTHRU;
}
```

Final Fragment

```
   j = 0;
   while(j < NumArrays) {
     read(fd, A, sizeof(int)*N);
     if (A[0] > 100)
       numSums++;
     doSum(N, A, &sum);
     if (exitKind==RETURN) return;
     totalSum += sum;
     j++;
   }
```

Figure 1. Example illustrating extraction of a “difficult” set of marked statements

row range of techniques, or employ restrictive versions of these techniques. [24, 7, 10] do not handle extraction of non-contiguous code at all ([10] does provide semantics-preserving primitives that the user can use to move individual unmarked statements; however they provide no automatic assistance in determining which statements need to be promoted, and in which direction the others can be moved – before or after the marked code). [8] handles extraction of non-contiguous fragments, but simply promotes *all* intervening unmarked code. [11] uses code reordering, but no promotion or duplication (and thus fails on cases where those techniques are required for extraction). [17] does employ several transformations (promotion, moving code, and duplication); however, our approach uses these transformations more effectively than theirs in most cases (see Section 4). Our work is an advance over all these previous approaches in that we not only employ a wide range of transformations, but also identify appropriate conditions under which to apply each transformation so that results are usually close to ideal.

The rest of this paper is organized as follows. Section 2 presents basic assumptions and terminology. Section 3 describes our procedure-extraction algorithm. Section 4 presents the results of a study that provides some quantitative data on how well the algorithm works in practice compared to "ideal" extraction and to previous techniques. Section 5 concludes the paper.

2. Assumptions and Terminology

We assume that the reader is familiar with the standard definitions of control and data dependence. We assume that programs are represented using a set of control-flow graphs (CFGs), one for each procedure. A CFG's exit node has no outgoing edge; predicate nodes have two outgoing edges; and all other nodes (assignments and procedure calls) have a single outgoing edge. Jumps (`gotos`, `returns`, `continues`, and `breaks`) are considered to be pseudo-predicates (i.e., predicates that always evaluate to true) as in [3, 6]. Therefore, each jump is represented by a node with two outgoing edges: the edge labeled *true* goes to the target of the jump, and the (non-executable) edge labeled *false* goes to the node that would follow the jump if it were replaced by a no-op. Jump statements are treated as pseudo-predicates so that the statements that are semantically dependent on a jump—as defined in [15]—are also control dependent on it.

Our algorithm makes use of the following definitions:

Definition 1 *A* **hammock** *is a subgraph of a CFG that has a single* entry node*, and from which control flows to a single* outside-exit node*. More formally: A hammock in CFG G is the subgraph of G induced by a set of nodes $H \subseteq \mathcal{N}(G)$ such that:*

1. *There is a unique entry node e in H such that:* $(m \notin H) \wedge (n \in H) \wedge ((m,n) \in \mathcal{E}(G)) \Rightarrow (n = e)$.
2. *There is a unique outside-exit node t in $\mathcal{N}(G) - H$ such that:* $(m \in H) \wedge (n \notin H) \wedge ((m,n) \in \mathcal{E}(G)) \Rightarrow (n = t)$.

Definition 2 *An* **e-hammock** *(a hammock with exiting jumps) is a subgraph of a CFG that has a single* entry node*, and, if all jumps are replaced by no-ops, a single* outside-exit node*; i.e., an e-hammock is a hammock that is allowed to include one or more exiting jumps (jumps whose targets are not inside the hammock and are not the hammock's outside-exit node).*

3. Procedure-Extraction Algorithm

The inputs to the algorithm are the control-flow graph of a procedure, and the set of nodes in that CFG that have been chosen for extraction (the marked nodes). When the algorithm finishes, the marked nodes will form a hammock, and thus extracting them into a separate procedure and replacing them with a procedure call will be straightforward. The algorithm runs in polynomial time (in the size of the code region that contains the marked code), and always succeeds (which is not the case for some previous approaches). It performs the following steps:

Step 1: Identify the code region to be transformed. (The region will be an e-hammock that contains the marked nodes, as explained later in this section.)

Step 2: Determine a set of ordering constraints among the nodes in the region based on flow and control dependences, and loop structure.

Step 3: Promote any unmarked nodes that cannot be moved out of the way of the marked nodes due to ordering constraints. From this point on, the promoted nodes are regarded as marked.

Step 4: Partition the nodes in the region into three "buckets": *before*, *marked*, and *after*. The *marked* bucket includes all the marked nodes; the *before* bucket includes all nodes that are forced by some constraint to precede some node in the *marked* bucket, and the *after* bucket includes nodes that are forced to follow some node in the *marked* bucket. During the partitioning process, the algorithm may also create copies of some *if*-statement predicates and some jumps if those copies are needed to preserve control dependences. (Assignment statements and loop predicates are not duplicated; the reason for this is discussed later.)

Step 5: Create three sequences of code using the nodes in the three buckets, and letting the relative ordering of nodes within each sequence be the same as in the original code. The code for the *marked* bucket will form a "normal" hammock, while the code for the other two buckets will form either normal hammocks or e-hammocks. Create the output by "stringing together" the three hammocks (i.e., using the entry node of the *marked* hammock as the outside-exit node of the *before* hammock, and using the entry node of the *after* hammock as the outside-exit node of the *marked* hammock). Finally, replace the original region identified in Step 1 with the output created here to obtain a resultant program that is semantically equivalent to the original.

The rest of this section provides more detail about each step of the algorithm, using the example in Figure 1 to illustrate each step. The CFG for the example is shown in Figure 2; the marked nodes are shaded, the non-executable edge out of the `return` is shown using a dashed edge, and the e-hammock $\mathcal{H}$, identified in Step 1 of the algorithm, is circled.

Step 1.

This step identifies the code region to be transformed: the smallest e-hammock $\mathcal{H}$ that includes all of the marked nodes, and that includes no backward exiting jumps. (An e-hammock H includes a backward exiting jump if for some exiting jump node j in H, H's entry node postdominates j's target.)

Example: In the example of Figure 2, the `return` is an exiting jump; if it were replaced by a no-op, the circled portion of the CFG would be a hammock (with "`sum=0`" as its entry node, and "`totalSum += sum`" as its outside-exit node). The circled portion includes no backward exiting jumps, and no smaller hammock includes all of the marked nodes, and therefore it is $\mathcal{H}$. □

Step 2.

This step is the heart of the extraction algorithm; it determines constraints based on data dependences, control dependences and loop structure among the nodes in $\mathcal{H}$. The constraints generated are of three forms: "$\leq$" constraints, "$=$" constraints, and "$\Rightarrow$" constraints. The constraints are used in Step 3 to determine which unmarked nodes must be promoted; they are also used in Step 4 to determine how to partition the remaining unmarked nodes between the *before* and *after* buckets, while preserving data and control dependences, and therefore the original semantics.

Data-dependence constraints: A data-dependence constraint $m \leq n$ is generated for each pair of nodes m, n such that there is a flow, def-order [4], anti [14], or output dependence [14] from m to n. Only dependences induced by paths in $\mathcal{H}$ are considered. The constraint $m \leq n$ means that node m must either go into a bucket that precedes n's bucket, or must go into n's bucket (where the ordering of the buckets is *before* < *marked* < *after*).

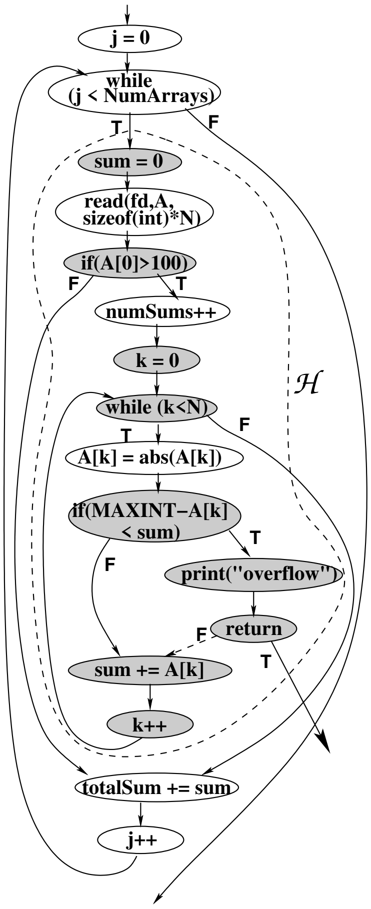

Figure 2. CFG for original fragment in Figure 1; $\mathcal{H}$ is the circled region.

Example: One of the data-dependence constraints generated for the running example is: "`read(fd, A, sizeof(int)*N)`" $\leq$ "`if (A[0]>100)`" (due to a flow dependence). This constraint forces the `read` statement to be placed in the *before* bucket in Step 4, since the `read` is an unmarked node and the `if` is marked. □

Loop-structure constraints: Assignment statements in $\mathcal{H}$ are never duplicated across multiple buckets. This is because duplicating assignments, in general, requires transformations such as renaming variables or copying and restoring entire data structures. Such transformations can reduce code quality and adversely affect the program's efficiency. Consequently, loops belonging to $\mathcal{H}$ are not split across multiple buckets (doing so would require duplicating the assignments in the loop body that update variables used in the loop predicate). Therefore, for each pair of nodes n, m

such that both are part of a strongly-connected component (a loop) in $\mathcal{H}$, a constraint $m = n$ is generated, which means that m and n must be placed in the same bucket.

Example: For the running example, an "$=$" constraint is generated between "`A[k] = abs(A[k])`" and every other statement in the inner loop. Since these other statements are marked, the only way to satisfy the "$=$" constraints is to promote "`A[k] = abs(A[k])`" (so that it will be placed in the *marked* bucket). The promotion is done in Step 3. □

Control-dependence constraints: For each node n in $\mathcal{H}$ and for each control ancestor p of n in $\mathcal{H}$, the constraint $n \Rightarrow p$ is generated, meaning: *a copy of node p must be included in the same bucket as node n*. This constraint, together with the actions in Step 5 (described below) ensure that control dependences in the original code are preserved. Notice that $n \Rightarrow p$ does not imply $n = p$, as copies of p can be present in other buckets besides n's bucket.

Two additional sets of constraints are generated to ensure that control dependences due to exiting jumps are preserved in the code output by the algorithm. We define an *antecedent* of an exiting jump j to be any node n such that n is not a "normal" predicate (i.e., n could be a jump), there is a j-free path in $\mathcal{H}$ from $\mathcal{H}$'s entry node to n, and there is a path in $\mathcal{H}$ from n to j (i.e., n could execute before the exiting jump executes). For each such n and j, we generate two constraints: (i) *if n is placed in the* after *bucket then a copy of j must be included in the same bucket*, and (ii) *if n but not j is placed in the* marked *bucket then a copy of j must be included in the* after *bucket*.

Example: In the running example, the control-dependence constraint "`numSums++`" $\Rightarrow$ "`if (A[0] > 100)`" is generated, which says that a copy of the `if` predicate must be placed in the same bucket as the increment of `numSums`. Since the `if` predicate is also a control parent of several marked nodes, a copy will also be placed in the *marked* bucket. This is the reason for the duplication of the `if` predicate in the algorithm output shown in Figure 1.

In this example, constraints of the second kind (i.e., involving exiting-jump antecedents) are generated, but they are never used; no node is placed in the *after* bucket, and the only exiting jump (the `return`) is placed in the *marked* bucket. □

Extended constraints: The data-dependence, loop-structure, and control-dependence constraints generated in the previous steps are used to generate extended constraints. The extended constraints are implied by the base constraints, but must be made explicit in order for Step 3 (promotion) and Step 4 (partitioning of unmarked nodes) to work correctly.

The first rule for generating extended constraints is to use standard rules of relational operations; e.g., $p \leq q$ and $q = n$ produces a new constraint $p \leq n$. In a similar vein, in the presence of a constraint $n \Rightarrow p$, the constraints $a \leq p$, $b = p$, and $p \leq c$ give rise to new constraints $a \leq n$, $b = n$ and $n \leq c$ respectively.

The second rule for generating extended constraints is: *if j is an exiting jump and n is an antecedent of j, then for each constraint $j \leq m$ or $j = m$ generate a new constraint $n \leq m$.*

Example: In the running example, extended constraints are generated using the first rule, but none of them are interesting. However, the second rule does generate some interesting extended constraints: The `return` is part of the inner loop, and so there are "=" constraints between the `return` and every other node in that loop (e.g., "`return`" = "`k++`"). Because `numSums++` is an antecedent of the `return`, the second rule for generating extended constraints causes a "$\leq$" constraint to be generated between `numSums++` and every node in the inner loop (e.g., "`numSums++`" $\leq$ "`k++`"). Since the nodes in the inner loop are all marked, these constraints force `numSums++` to be placed in the *before* bucket. □

Step 3.

This step promotes all unmarked nodes n such that there exists a constraint $n = m$ for some marked node m, or such that there exist constraints $m_1 \leq n$ and $n \leq m_2$ for some marked nodes m_1 and m_2. From this point on, the promoted nodes are regarded as marked.

Example: In the running example, the node "`A[k] = abs(A[k])`" is promoted because of the "$=$" constraints, as discussed earlier. However, even if there were no loop, this node would have been promoted because of two flow constraints: "`k = 0`" $\leq$ "`A[k] = abs(A[k])`" and "`A[k] = abs(A[k])`" $\leq$ "`sum += A[k]`". □

Step 4.

This step starts by placing all marked nodes, as well as copies of all of their control ancestors, in the *marked* bucket. Next, the unmarked nodes are assigned to the *before* and *after* buckets. Any unmarked node n that is forced into a particular bucket (*before* or *after*) by the constraints is placed in that bucket along with copies of all its control ancestors (no node can be forced into both buckets, because such a node would have been promoted in Step 3). If at any point there is no forced unmarked node, then an as-yet-unassigned, unmarked node that is not a "normal" predicate is chosen at random, and is randomly placed in either the *before* or the *after* bucket, along with copies of all its control ancestors.

Note that, as mentioned earlier, nodes that are not forced by constraints can be placed in *either* bucket safely. In other words, there may be more than one partitioning that satisfies all constraints, and the algorithm constructs one such partitioning (any partitioning that satisfies all constraints is semantics-preserving).

Example: As discussed above, the two unmarked nodes in the running example are both forced into the *before* bucket. A copy of "`if (A[0] > 100)`" is also placed in the *before* bucket because it is a control parent of `numSums++`. □

Step 5.

In this final step, the three buckets are converted into hammocks (a "normal" hammock for the *marked* bucket, and an e-hammock for the *before* and *after* buckets). A bucket is converted into the corresponding result hammock by making a copy of $\mathcal{H}$ and removing from that copy the nodes that are not in the bucket. These nodes are guaranteed to form multiple-entry single-outside-exit regions within $\mathcal{H}$, so removing them simply involves redirecting all edges that enter a region to its outside-exit node. If the outside-exit node of a region is also the outside-exit node of $\mathcal{H}$, then the edges are redirected to the outside-exit node of the created result hammock.

Note that copies of a jump node can be present in multiple result hammocks (jump nodes are pseudo predicates and thus can be duplicated in Step 4). Note also that labels are not included in the CFG, and are implicitly represented by the (executable) edge from a `goto` node to its target. If copies of a `goto` node are present in multiple buckets, then during conversion of the result CFG back to actual source code, unique labels will need to be supplied for each `goto` (but this is straightforward).

Certain jump nodes are processed specially in this step, as described below.

Non-exiting jumps: Copies of non-exiting `breaks`, `continues` and `gotos` in the result hammocks remain unchanged (the hammock-creation procedure described earlier in this step ensures that edges out of these jumps have their correct targets). A `return` in $\mathcal{H}$ is non-exiting when the outside-exit node of $\mathcal{H}$ is also the end of the containing procedure. Any copy of a non-exiting `return` in the *before* or *marked* hammock is converted into a `goto` whose target is the outside-exit node of that hammock. Finally, a new assignment "`exitKind = FALLTHRU`" is introduced before every `goto` node in the *marked* hammock that is either a copy of a non-exiting `goto` or that was obtained by the `return` conversion just described, and whose target is the outside-exit node of the *marked* hammock. (Recall that `exitKind` is a newly introduced global variable.)

Exiting jumps: If copies of an exiting-jump node j are present in multiple result hammocks, then each copy except the last one is converted into a `goto` whose target is the outside-exit node of the hammock that contains that copy. This is done to ensure that antecedents of j (i.e., nodes that can be reached before control reaches j during an execution of $\mathcal{H}$), which may be present in the result hammock that has the last copy of j or in any hammock preceding that one, are not incorrectly skipped in the execution of the resultant code.

Additionally, if the *marked* hammock contains the last copy of an exiting jump j, then the following are done: This copy is converted into a `goto` whose target is the entry of the *after* hammock. An assignment "`exitKind =` *enc*" is inserted just before this new `goto`, where *enc* is a value (such as `BREAK`, `RETURN`) that encodes the original kind of j. An assignment "`exitKind = FALLTHRU`" is introduced at the end of the *marked* hammock. Finally, the following new compensatory code is placed at the entry of the *after* hammock (as the target of the new `goto`): an `if` statement of the form "`if (exitKind ==` *enc*`)` *jump*", where *jump* is a copy of the exiting jump j.

At this point Step 5 (and hence the entire algorithm) is finished. A key property of this step is that the ordering of nodes on any path within any of the three result hammocks is identical to the ordering of these same nodes in the corresponding path in $\mathcal{H}$. This, together with the constraints used to assign nodes to buckets, guarantees semantics preservation.

Note that the algorithm converts certain jump nodes into `gotos`, but does not change any other original CFG nodes in any way (it only moves them around and/or duplicates them). It however does introduce a small amount of *new* code – the assignments to `exitKind` as well as the compensatory code in the *after* hammock. Note also that, as mentioned earlier, the algorithm makes the marked nodes form a hammock, but does not perform actual extraction. At the time of extraction, all `gotos` in the *marked* hammock whose targets are the outside-exit node of this hammock are simply converted into `returns`.

Example: The final hammocks for the running example are shown, in source-code form, in the upper-right column of Figure 1. The code marked "++" is the marked hammock, the code preceding that is the *before* hammock while the code following it is the *after* e-hammock.

Note that Step 5 converted the copy of the `return` node in the *marked* hammock into a `goto`, created the assignments to `exitKind`, and placed compensatory code in the *after* e-hammock. Since no nodes were placed in the *after* bucket in Step 4, the *after* e-hammock contains nothing besides the compensatory code.□

4. Experimental Results

We performed some studies to evaluate the performance of our algorithm, both in comparison to an "ideal" extraction (performed by us, using our best judgement), and to previously reported automatic approaches. Our dataset consisted of 157 computations to extract; the maximum size (number of simple statements and predicates) of a computation was 57, and the median was 7. The dataset was drawn from three programs: the Unix utilities *bison* and *make*, and NARC[1] [23], a graph-drawing engine developed by IBM. These programs range in size from 11,000 to 30,000 lines of code.

To expedite the process of finding computations to extract, we used the tool reported in [12] that finds duplicated code (including "near" duplicates that are not trivial to extract). We call the duplicate copies identified by the tool *clones*. For each clone, we ensured that all intervening statements that were logically part of the computation were marked. For example, given the code in Figure 1, we would add the statement "`A[k] = abs(A[k])`" to the set of marked statements because it is an integral part of the "sum" computation. We did this because, although our algorithm can handle inputs that are not logically complete computations (as illustrated in Figure 1), previously defined approaches are generally meant to handle logically complete computations only. Therefore, using "incomplete" computations would have given our algorithm an unfair advantage over previous approaches.

4.1. Comparison to ideal extraction

As the first step in the comparison, we extracted each clone according to our best judgement. We refer to those results as "ideal". Some of the techniques used during this process (reordering statements, handling exiting jumps, duplicating predicates) are also used by the algorithm; other techniques not incorporated in the algorithm were also used as necessary (this is described later). The second step in the comparison was to (manually) apply the algorithm to each clone. Figure 3 presents the results of the comparison.

Figure 3(a) summarizes the performance of the algorithm compared to the ideal extraction. The set of all clones is divided into three disjoint categories, one per row. The first row shows that 93 (of the 157) clones are "not difficult"; i.e., they are contiguous and they do not involve exiting jumps. Such clones are extractable to begin with, and therefore our algorithm has nothing to do. Going on to the second row, there are 46 difficult clones on which our algorithm produces exactly the ideal output. Of these 46, 22 are non-contiguous and 27 involve exiting jumps (i.e., 3 of the 46 clones exhibit both difficult characteristics). On 18 other difficult clones, our algorithm succeeds but produces non-ideal output (more discussion on this later).

Figure 3(b) enumerates, for each transformation technique incorporated in the algorithm, the number of difficult clones on which the technique was used in both the ideal extraction and the extraction performed by the algorithm. Each technique appears in its own row. The second column (labeled "Ideal output") pertains only to the difficult clones on which the algorithm performed ideally; therefore

[1]NARC is a registered trademark of IBM.

Category	total # clones	# non contig.	# exiting jumps
Not difficult	93	-	-
Difficult, ideal output	46	22	27
Difficult, non ideal output	18	18	6
	157		

(a) Characterization of algorithm output.

Technique	Ideal output	non-ideal output	
		human	algo.
Moving with duplication	18	15	5
Moving without duplication	4	3	-
Exiting jumps	27	6	6
Promotion	-	-	17

(b) Techniques used on difficult clones, with number of clones

Figure 3. Comparison of algorithm and ideal extraction.

the numbers in this column pertain both to the ideal extraction and the extraction performed by the algorithm. The third and fourth columns pertain to the clones on which the algorithm performed non-ideally; two separate sets of numbers are required here because the algorithm and the ideal extraction do not involve the exact same techniques on these clones.

Regarding the techniques, "moving with duplication" is actually a combination of two techniques, statement reordering and predicate duplication, while "moving without duplication" is simply statement reordering. Both of these are used to move intervening unmarked code out of the way (as illustrated in Figure 1, by the unmarked statements "`numSums++`" and "`read(fd,A,sizeof(int)*N)`", respectively). The technique of handling exiting jumps is applicable on each clone that involves exiting jumps, and therefore the numbers in that row are the numbers in the last column of Figure 3(a).

From Figure 3(a) it is clear that many of the clones in the data set (41% – 64 out of 157) are "difficult" to extract (i.e., they need to be transformed to be made extractable). Specifically, 25% of the clones are non-contiguous and 21% involve exiting jumps (5% involve both difficult aspects). From Figure 3(a), it is also clear that the algorithm performs ideally on most (46 out of 64) of the difficult clones.

However, the algorithm performs non-ideally on 18 difficult clones. On 17 of these 18 the algorithm promoted certain intervening unmarked nodes that the ideal extraction managed to move out of the way. In all but two of these 17 clones, the ideal extraction used a single technique that is not incorporated in the algorithm: saving values of variables or expressions into temporaries, and using these temporaries later.

The deviation of the algorithm's performance from ideal is noticeable, but perhaps not unacceptable, considering that no automatic algorithm is likely to be able to employ the full range of transformation techniques used by a human. In fact, on 16 of the 18 clones on which our algorithm produced non-ideal output (but succeeded), the automatic approach proposed by Lakhotia *et al* [17], which is the one that is most closely related to ours, fails completely, as discussed below.

4.2. Comparison to previous work

The second goal of our study was to compare our algorithm with two previously defined automatic approaches, those of [17] and [11]. [11] uses a general notion of code movement (which our approach shares), but does not handle exiting jumps. More significantly, they do no duplication of predicates. As a result, their performance on the dataset was poor: they performed ideally on 4 of the 64 difficult clones (none of which involved exiting jumps), and failed on all others. This is evidence that a general notion of code movement alone is not enough, and that a combination of several other techniques (handling exiting jumps, promotion, predicate duplication) is necessary for successful extraction of a wide class of difficult clones.

The approach of Lakhotia [17], briefly, is to find the tightest (normal) hammock H containing the marked nodes, and to create a *marked* and an *after* hammock from the nodes in H (they do not use a *before* hammock). The key differences between our approach and theirs are:

- They promote all nodes in H that are in the backward slice from the marked nodes. We do not do this, because we can move such code to the *before* bucket (which they do not use).

- We allow dataflow from the *marked* hammock to the *after* hammock. They disallow this, and instead place in the *after* hammock all nodes in H that are in the backward slice from unmarked/unpromoted nodes. This can cause duplication of the marked code in the *after* bucket, thereby defeating the purpose of extraction. (This is why, as described in Section 1.2, their approach duplicates nearly all the marked nodes in the example of Figure 1).

Category	total # clones	# non contig.	# exiting jumps
Both outputs ideal	3	3	-
Both outputs non-ideal	2	2	2
Their output non-ideal (ours ideal)	19	6	16
They fail (we succeed)	40	29	15
	64		

Figure 4. Comparison of our algorithm and Lakhotia's algorithm

- Our use of code motion is better than theirs, and so is our use of promotion. As a result we always succeed in transforming the marked code to make it extractable, whereas they may fail under several circumstances.

- They do not handle exiting jumps, and therefore have to start from the tightest hammock containing the marked code. The tightest hammock is usually larger (and never smaller) than the tightest e-hammock, which means they have more unmarked nodes to deal with, which exacerbates all the problems mentioned earlier.

- They do allow duplication of assignments, and saving and restoring variable values (although they do not address the difficult issues that come up in this context when arrays and pointers are present). Our approach does not save and restore values, and duplicates only predicates. Although these features of their approach can potentially make it better than ours in some cases, it can also increase duplication of marked code. In practice, their other drawbacks outweighed these features, preventing their approach from performing better than ours on even a single clone in the dataset.

Figure 4 provides data comparing the performance of our algorithm and Lakhotia's on the dataset. In the figure and in the following discussion we talk about difficult clones only, because no transformation is required by either algorithm to make the non-difficult clones extractable. The 64 difficult clones are divided into four disjoint categories (based on the performance of the two algorithms on the clones), with one category per row. The first row is for clones on which both algorithms succeeded and produced the ideal output; the second row is for clones on which both produced non-ideal output; the third row is for clones on which our algorithm produced ideal output whereas theirs produced non-ideal output, and the fourth row is for clones on which they failed while we succeeded. Their algorithm did not produce ideal output on any clone on which ours produced non-ideal output.

Notice that on all but 5 clones (those in the first two rows) their algorithm performed worse than ours. An important reason for this is that they do not handle exiting jumps: They failed on 15 clones (on which our algorithm succeeded) and performed non-ideally on 13 clones (on which our algorithm performed ideally) solely because of exiting jumps; i.e., if the exiting jumps were removed they would succeed on the 15 clones, and perform ideally on the 13.

However, handling exiting jumps is not the only advantage of our algorithm over theirs; our notion of when unmarked nodes can be moved away is less restrictive than theirs, and our rules for promotion are better. There are 31 clones (in addition to the 28 discussed in the previous paragraph) on which their algorithm would fail or perform non-ideally even if all exiting jumps were removed. Our algorithm (in the presence of the exiting jumps) succeeded on all 31, and produced ideal output in most cases.

5. Conclusions

Extracting selected sets of statements into separate procedures can often improve the understandability and maintainability of legacy programs. We have described an algorithm for transforming "difficult" sets of statements to make them extractable. Our work makes three key contributions over previous work: we handle exiting jumps, we use a range of transformations (and identify conditions under which each one is applicable) to make code extractable in a manner that is close to ideal, and we performed a study using three real programs. The study revealed that "difficult" sets of statements do occur often – 25% of the sets of statements encountered were non-contiguous, while 21% involved exiting jumps. Our algorithm performed well, achieving ideal results on over 70% of the difficult cases, and outperforming previous approaches in all but a few cases.

References

[1] E. Ashcroft and Z. Manna. The translation of 'go to' programs to 'while' programs. In *Proc. Inf. Processing '71*, pages 250–255. North-Holland Publishing Company, 1972.

[2] B. Baker. On finding duplication and near-duplication in large software systems. In *Proc. IEEE Working Conf. on Reverse Eng.*, pages 86–95, July 1995.

[3] T. Ball and S. Horwitz. Slicing programs with arbitrary control flow. In *Lecture Notes in Computer Science*, volume 749, pages 206–222. Springer-Verlag, 1993.

[4] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proc. ACM Symp. on Principles of Prog. Langs.*, pages 384–396, Jan. 1993.

[5] R. Bowdidge and W. Griswold. Supporting the restructuring of data abstractions through manipulation of a program visualization. *ACM Trans. on Software Eng. and Methodology*, 7(2):109–157, Apr. 1998.

[6] J. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Trans. on Prog. Langs. and Systems*, 16(4):1097–1113, July 1994.

[7] K. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. ACM Conf. on Prog. Lang. Design and Impl.*, pages 139–149, May 1999.

[8] S. Debray, W. Evans, R. Muth, and B. DeSutter. Compiler techniques for code compaction. *ACM Trans. on Prog. Langs. and Systems*, 22(2):378–415, Mar. 2000.

[9] M. Fowler. *Refactoring – Improving the design of existing code*. Addison Wesley Longman, Reading, Mass., 1999.

[10] W. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Trans. on Software Eng. and Methodology*, 2(3):228–269, July 1993.

[11] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *Proc. ACM Symp. on Principles of Prog. Langs.*, pages 155–169, Jan. 2000.

[12] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. Int. Symp. on Static Analysis*, pages 40–56, July 2001.

[13] K. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Eng.*, 3(1–2):77–108, 1996.

[14] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proc. ACM Symp. on Principles of Prog. Langs.*, pages 207–218, Jan. 1981.

[15] S. Kumar and S. Horwitz. Better slicing of programs with jumps and switches. In *Proc. Fundamental Approaches to Software Eng.*, pages 96–112, Apr. 2002.

[16] B. Lague, D. Proulx, J. Mayrand, E. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Int. Conf. on Software Maintenance*, pages 314–321, 1997.

[17] A. Lakhotia and J. Deprez. Restructuring programs by tucking statements into functions. *Inf. and Software Technology*, 40(11–12):677–689, Nov. 1998.

[18] F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Trans. on Software Eng.*, 23(4):246–258, Apr. 1997.

[19] S. Letovski and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, pages 198–204, May 1986.

[20] G. Oulsnam. Unravelling unstructured programs. *The Computer Journal*, 25(3):379–387, Aug. 1982.

[21] S. Rugaber, K. Stirewalt, and L. Wills. Understanding interleaved code. *Automated Software Eng.*, 3(1–2):47–76, June 1996.

[22] H. Sneed and G. Jandrasics. Software recycling. In *Proc. Conf. Software Maintenance*, pages 82–90, 1987.

[23] V. Waddle and A. Malhotra. An E log E line crossing algorithm for leveled graphs. In *Lecture Notes in Computer Science*, volume 1731, pages 59–71. Springer-Verlag, 1999.

[24] M. Zastre. Compacting object code via parameterized procedural abstraction. Master's thesis, Department of Computer Science, University of Victoria, British Columbia, 1995.