Loop Squashing Transformations for Amorphous Slicing*

Lin Hu, Mark Harman, Robert M. Hierons
Brunel University
Uxbridge, Middlesex
UB8 3PH, UK.
{lin.hu,mark.harman,rob.hierons}@brunel.ac.uk

David Binkley
Loyola College
4501 North Charles Street
Baltimore, MD 21210–2699, USA
binkley@cs.loyola.edu

Abstract

Program slicing is a source code extraction technique that can be used to support reverse engineering by automatically extracting executable subprograms that preserve some aspect of the original program's semantics. Although minimal slices are not generally computable, safe approximate algorithms can be used to good effect. However, the precision of such slicing algorithms is a major factor in determining the value of slicing for reverse engineering.

Amorphous slicing has been proposed as a way of reducing the size of a slice. Amorphous slices preserve the aspect of semantic interest, but not the syntax that denotes it, making them generally smaller than their syntactically restricted counterparts. Amorphous slicing is suitable for many reverse engineering applications, since reverse engineering typically abandons the existing syntax to facilitate structural improvements.

Previous work on amorphous slicing has not attempted to exploit its potential to apply loop–squashing transformations. This paper presents an algorithm for amorphous slicing of loops, which identifies induction variables, transformation rule templates and iteration–determining compile–time expressions. The algorithm uses these to squash certain loops into conditional assignments. The paper also presents an inductive proof of the rule templates and illustrates the application of the algorithm with a detailed example of loop squashing.

1. Introduction

Program slicing is an automated source code extraction technique which produces a version of a program that preserves a projection of the original program's semantics [7, 17, 22, 33, 36]. Traditionally, this projection is defined in terms of a subset of variables of interest and is constructed using the sole transformation of statement deletion [36]. The slice is therefore a subprogram which preserves a subcomputation.

Program slicing has been applied to several stages of the reverse engineering process, such as program restructuring [4, 9, 10, 28, 31] program comprehension [14, 18, 28, 29, 30] regression testing [5] and program integration [26]. In all these applications, the important aspect of slicing is the way in which it allows the reverse engineer to extract a semantically meaningful sub–computation, based on a slicing criterion which captures the aspect of the overall computation.

Program slicing can simplify a program. However, the resulting slice may remain large; perhaps too large to be useful for reverse engineering. Amorphous slicing [23, 21] has been proposed as a way of reducing the size of a slice. Amorphous slices preserve the aspect of semantic interest, but not the syntax that denotes it, making them generally smaller than their syntactically restricted counterparts.

Amorphous slicing is suitable for many reverse engineering applications, since when re–engineering a system, the program is typically restructured; thus the existing syntax is modified. It is unlikely that a re–engineered system will faithfully preserve a subset of the syntax of the program from which it is constructed. In this situation amorphous forms of slicing are more attractive than their syntax–preserving counterparts because amorphous slices are always no bigger than (and typically are smaller than) their syntax–preserving counterparts.

This paper presents an algorithm that improves the amorphous slicing of loops. The algorithm incorporates a type of loop transformation, which is employed to further reduce dependences while amorphous slicing. When this dependence reduction transformation is applied to a loop, it transforms the loop into a conditional assignment; thus, further reducing the slice size.

Figure 1 shows an example of a syntax–preserving slice. The left–hand column shows a program that takes as input number n, and computes the sum and product of the first n positive integer. The right–hand column of the figure is the syntax–preserving slice with respect to variable `sum` at the end of the program. The slice captures a semantic projection of the original program.

Original program	A slice w.r.t *sum*

`i:=1;`
`sum:=0;`
`product:=1;`
`while i <= n do`
`  sum:=sum+i;`
`  product:=product*i;`
`  i :=i+1;`
`od`
	`i:=1;`
	`sum:=0;`
	
	`while i <= n do`
	`  sum:=sum+i;`
	
	`  i:=i+1;`
	`od`

Figure 1. Syntax–preserving program slicing

The example in Figure 1 is well–known and is widely used in the literature to illustrate slicing. One would be forgiven for thinking that nothing more can be said about it. However, as this paper will show, it is possible, using loop squashing, to further reduce the size of the slice (and many like it). Observe that, though the slice is reduced, the restriction to syntax–preservation prevents the slicer from achieving any further simplification. Previous work on amorphous slicing [6, 23] also cannot reduce this program further, since no attempt is made to squash loops into conditionals. However, applying loop–squashing reduces the slice to the code shown in Figure 2.

```
i := 1;
sum := 0;
if i <= n then
  sum := n*(n+1)/2;
fi
```

Figure 2. An amorphous slice using loop–squashing

Furthermore, under the assumption that all loops execute at least once, this can be further transformed to a single assignment

```
sum := n*(n+1)/2;
```

Our experiments indicate that conditioned slicing [8, 13, 20] can detect when a loop must execute at least once. In this case the example slice reduces to a single statement. However, this paper focuses upon the problem of loop squashing; all loops will be reduced to conditionals to ensure semantic preservation (in the case that the loop does not iterate). The problem of integrating conditioned slicing with loop squashing remains a problem for future work.

The primary contributions of this paper are as follows.

1. The paper presents a loop–squashing algorithm for amorphous slicing,
2. The algorithm is based upon a set of rules for loop transformation. The rules are proved correct using inductive proofs.
3. The paper illustrates the application of the algorithm with a detailed example.

The rest of this paper is organized as follows. Section 2 introduces the loop squashing algorithm and proves that the transformation upon which it is based is correct. Section 3 illustrates the application of the algorithms with a simple case study. Section 4 presents related work and Section 5 concludes with directions for future work.

2. Loop Squashing Algorithm in WSL

This section presents the loop squashing algorithm, which has been incorporated into the DRT component of GUSTT amorphous slicer [24]. The slicer considers program written in a C-like language with considerably cleaner semantics, named WSL [34, 35]. In WSL, the `while` loop construct of interest in this paper is

```
while e do
    ...
od
```

For presentation clarity, algorithm and examples appearing herein treat `while` loops. The techniques extend to other kind of loops. For example, `for` loops actually simplify the application of the transformation as they syntactically identify the loop induction variable. The examples also ignore nested loops. The technique can be applied to nested loops by processing the innermost loop and working its way out until a loop that cannot be squashed is encountered.

The loop–squashing transformation has five steps: induction variable identification, normalization, pattern matching, iteration computation, and loop replacement. Each of these steps is described and then they are combined into the algorithm for loop squashing.

Step 1 identifies a loop induction variables using the sets REF(e), the referenced variables in expression e, and DEF(s), the variables assigned in the statement sequence s. If the set “REF(e) $\cap$ DEF(s)” contains a single element, v, that is incremented or decremented the same amount on each loop iteration, then v can be used as a loop induction variable. This definition is similar to the standard definition [1, 2].

```
function Normalise(v: variable, sl: Statement Sequence) returns a Statement Sequence
declare
  cl, WorkList, DoneList, TopList: Statement Sequences
  b: Predicate Expression
  c, s, s1, s2: Statements
  Stuck: Booleans
begin
  WorkList ← sl
  DoneList ← []
  TopList ← []
  while WorkList ≠ [] do
    if head(WorkList) is an assignment statement
    then
      if assigned variable in head(WorkList) is v
      then
        (c,cl,Stuck) ← PUSH(head(WorkList),tail(WorkList))
        if not(Stuck)
        then
          WorkList ← cl
          DoneList ← Append(c,DoneList)
        else
          TopList ← Append(TopList, WorkList)
          return (Append(TopList, DoneList))
        fi
      else
        TopList ← Append(TopList,c)
        WorkList ← tail(WorkList)
      fi
    else    /* Head(WorkList) is not an assignment statement */
      if head(WorkList) is an if statement
      then
        let [[if b then s1 else s2]] be head(WorkList)
        TopList ← Append(TopList,[[if b then Normalise(v, s1) else Normalise(v, s2)]])
      else   /* unrecognized statement encountered: Leave head(WorkList) untransformed */
            TopList ← Append(TopList,head(WorkList))
      fi
      WorkList ← tail(WorkList)    /* move on to consider the next statement in WorkList */
    fi
  od
 return (Append(TopList, DoneList))
end
```

Figure 3. The Normalise Function

Assuming a suitable induction variable exists, Step 2 attempts to place the loop in canonical form using the function Normalise() shown in Figure 3. The normalization process attempts to move and merge all assignments to the loop induction variable to the end of the loop body.

Function Normalise walks through the abstract syntax tree of the loop body, applying the auxiliary transformation tactic PUSH [23, 25]. The PUSH tactic takes a statement list and attempts to push the first assignment statement forward in a statement sequence. As the assignment passes statements which reference its defined variable, they have to be updated to reflect the symbolic effect of the assignment's execution. The pushing forward of an assignment is achieved by applying the Push rule shown in Figure 4.

An assignment can get 'stuck' in the pushing process. This happens when an attempt is made to push an assignment which references a variable past a statement that defines the variable. In this situation the PUSH tactic has no effect upon the statement sequence through which the assignment is to be pushed.

For example, suppose that $f()$, $g()$, $h()$, and $p()$ are linear functions in the following code sequence

Rule 1 (Assignment to Assignment: Absorb)

$$\frac{e_3 = SUB(e_2, i, e_1)}{[\![i := e_1;\ i := e_2]\!] \Rightarrow [\![i := e_3]\!]}$$

Rule 2 (Assignment to Assignment: MoveToRight)

$$\frac{i_1 \neq i_2, i_2 \notin REF(e_1), e_3 = SUB(e_2, i_1, e_1)}{[\![i_1 := e_1;\ i_2 := e_2]\!] \Rightarrow [\![i_2 := e_3;\ i_1 := e_1]\!]}$$

Rule 3 (Assignment to if-then-else)

$$\frac{e_2' = SUB(e_2, i, e_1),\quad [\![i{:=}e_1;\ c_1]\!] \Rightarrow [\![c_1'\ \ i{:=}e_1;]\!],\quad [\![i{:=}e_1;\ c_2]\!] \Rightarrow [\![c_2'\ \ i{=}e_1;]\!]}{[\![i{:=}e_1;\ if\ (e_2)\ then\ c_1\ ELSE\ c_2\ fi]\!] \Rightarrow [\![if\ (e_2')\ then\ c_1'\ ELSE\ c_2'\ fi;\ \ i{:=}e_1]\!]}$$

Rule 4 (Others: Statement to Statement)

$$\frac{DEF(st_1) \cap REF(st_2) = \phi,\ REF(st_1) \cap DEF(st_2) = \phi, DEF(st_1) \cap DEF(st_2) = \phi}{[\![st_1;\ st_2]\!] \Rightarrow [\![st_2;\ st_1]\!]}$$

Figure 4. A subset of Push Rules, where the term $SUB(e_2, i, e_1)$ returns the expression that results from substituting all occurrences of the variable i in the expression e_2, with the expression e_1, and REF is a function which returns the referenced variables of an expression

```
while f(t) <= 0 do
   x := x+g(t);
   t := p(t);
   y := y+h(t);
od
```

To normalise this loop with respect to induction variable `t` requires computing

Normalise(`t`, $[\![$`x:=x+g(t);t:=p(t);y:=y+h(t);`$]\!]$)

which in turn requires computing

PUSH($[\![$ `t:=p(t); y:=y+h(t);`$]\!]$).

The PUSH transformation applies Rule 2 of Figure 4, to move the assignment `t:=p(t)` forward. The resulting normalised loop is

```
while f(t) <= 0 do
   x := x+g(t);
   y := y+h(p(t));
   t := p(t);
od
```

This loop is in the canonical form required in the next step.

Step 3 first checks that the transformed loop pattern matches with the required canonical form as shown in Figure 5. It can fail to be in the form, for example, if the PUSH tactic gets stuck. If the matching succeeds, it extracts certain values from the loop. These include the values of the variables p, n, d, r, and q as used in Figure 5.

Step 4 computes the number of the iterations the loop will execute based on the increment (or decrement) of the induction variable. For example, when $p*d \neq 0\ and\ r = 1$, the maximum integer n that satisfies $p*(i+n*d)+q \leq 0$ = $Integer(\frac{-p*i-q}{p*d})$ is the iteration count.

while $p*i+q \leq 0$ do
 $S_1 := a_1 * S_1 + b_1 * i + c_1$;
 $S_2 := a_2 * S_2 + b_2 * i + c_2$;
 ...
 $S_m := a_m * S_m + b_m * i + c_m$;
 $i := r * i + d$;
od;

Figure 5. The canonical form of `while` loop to be considered, where m is an integer and $m \geq 0$

Finally, assuming it is reached, Step 5 performs the actual squashing transformation. The result is in the form shown in Figure 6. The rewriting of a canonical `while` loop uses the five *Loop Squashing Transformation Rules* shown in Figure 7. Each rule describes the transformation of the entire loop. For loops with more than one loop–body statement (*i.e.*, where m of Figure 6 is more than one), only the effect on the statement S_i is of interest. This is because each rule performs the same transformation on the other parts of the loop.

The five rewriting rules from Figure 7 have been proved correct using mathematical induction. There is insufficient space in this paper to provide a proof of all the rules in full detail. The proof for Case 3 is typical and it is presented in Figure 9.

```
while p*i+q ≤ 0 do
   S := a*S + b*i + c;
   i := r*i + d;
od;
```

If the number of iterations is n, then the `while` loop above can be transformed to the `if` statement as below

(1) When $a = 1, r = 1$:

```
if p*i+q ≤ 0 then
   S := S + b*(n*i + n*(n-1)*d/2) + n*c;
   i := i + n*d;
fi;
```

$$S := S + b * (n * i + \frac{n*(n-1)*d}{2}) + n * c;$$

(2) When $a = 1, r \neq 1$:

```
if p*i+q ≤ 0 then
   S := S + (b*i - b*d/(1-r)) * (1-r^n)/(1-r) + (b*d/(1-r) + c)*n;
   i := i*r^n + d*(1-r^n)/(1-r);
fi;
```

$$S := S + (b * i - \frac{b*d}{1-r}) * \frac{1-r^n}{1-r} + (\frac{b*d}{1-r} + c) * n; \quad i := i * r^n + d * \frac{1-r^n}{1-r};$$

(3) When $a \neq 1, r = 1$:

```
if p*i+q ≤ 0 then
   S := S*a^n + (b*i + c)*(1-a^n)/(1-a) + b*d*((a^n-1)/(a-1)^2 - n/(a-1));
   i := i + n*d;
fi;
```

$$S := S * a^n + (b * i + c) * \frac{1-a^n}{1-a} + b * d * (\frac{a^n-1}{(a-1)^2} - \frac{n}{a-1});$$

(4) When $a \neq 1$, $r \neq 1$, and $a = r$:

```
if p*i+q ≤ 0 then
   S := S*a^n + n*(b*i - b*d/(1-r))*r^n + (b*d/(1-r) + c)*(1-a^n)/(1-a);
   i := i*r^n + d*(1-r^n)/(1-r);
fi;
```

$$S := S * a^n + n * (b * i - \frac{b*d}{1-r}) * r^n + (\frac{b*d}{1-r} + c) * \frac{1-a^n}{1-a}; \quad i := i * r^n + d * \frac{1-r^n}{1-r};$$

(5) When $a \neq 1$, $r \neq 1$, and $a \neq r$:

```
if p*i+q ≤ 0 then
   S := S*a^n + (b*i - b*d/(1-r))*(a^n-r^n)/(a-r) + (b*d/(1-r) + c)*(1-a^n)/(1-a);
   i := i*r^n + d*(1-r^n)/(1-r);
fi;
```

$$S := S * a^n + (b * i - \frac{b*d}{1-r}) * \frac{a^n-r^n}{a-r} + (\frac{b*d}{1-r} + c) * \frac{1-a^n}{1-a}; \quad i := i * r^n + d * \frac{1-r^n}{1-r};$$

Figure 7. Loop Squashing Transformation Rules

Combining the five steps produce the loop–squashing transformation. An algorithm for carrying out this transformation is presented in Figure 8.

3. Case Study

This section illustrates the application of the loop squashing algorithm using the example program shown in Figure 10. The program, referred to as P_0, computes the weight and the coordinate (x, y) for the centre of gravity given the density and the shape of the board. Figure 11 shows an example board shape.

Slicing program P_0 with respect to the variable `gravity` yields the syntax–preserving slice shown in Figure 12. Using the loop squashing algorithm, the `while` loop from Figure 12 can be transformed to an `if` statement shown in Figure 13; thus squashing provides additional simplification.

The rest of this section demonstrates the application of loop squashing in conjunction with program conditioning.

```
Function Squash(L : while loop) returns while loop × iteration count
    assume L = while e do s od
    If |REF(e) ∩ DEF(s)| ≠ 1 then return L × 0                                   – Step 1
    let {< i >} = REF(e) ∩ DEF(s)
    L_norm = Normalise(L)                                                         – Step 2
    let (p, q, r, d) = pattern_match_with_canonical_form(L_norm, i)               – Step 3
    if the pattern match fails then return L × 0
    if p * d ≠ 0 and r = 1 then                                                   – Step 4
        let iteration_count = Integer((-p*i-q)/(p*d))
    else if p ≠ 0 and r ≠ 1
        let iteration_count = Integer((ln(p*d+q*(1-r))-ln(p*d-p*i*(1-r)))/ln(r))
    else
        return L × 0
    fi
    if a Loop Squashing Transformation Rule applies to each statement in L_norm then   – Step 5
      return the transformed loop × iteration_count
    else
      return the L × 0
    fi
end
```

Figure 8. Loop Squashing Algorithm

```
if p * i + q ≤ 0 then
  S_1 := φ_1(a_1, S_1, b_1, i, c_1);
  S_2 := φ_2(a_2, S_2, b_2, i, c_2);
  ...
  S_m := φ_m(a_m, S_m, b_m, i, c_m);
  i := ω(r, i, d, p, q);
fi;
```

Figure 6. The resulting conditional, where ϕ_1, ϕ_2, ..., ϕ_m and ω are generated by loop–squashing algorithm

This work remains experimental, but the initial results using the *ConSUS* conditioned slicing system [13, 16] are encouraging (though, at present, anecdotal).

Program conditioning [8, 15, 20] is a program simplifying technique which has been used in conditioned slicing. Program conditioning is used to identify and remove paths which become infeasible when the execution conditions are applied. The resulting program is called a conditioned program, which can be computed using a symbolic executor.

Using the condition $n > 1$, conditioning of program P_2 (Figure 13) followed by dependence reduction transformation (*e.g.*, application of the PUSH tactic), yields the simplified program shown in Figure 14. Looking back at the original program P_0, we can observe that if a software engineer is interested only in the computation of variable `gravity` under condition $n > 1$, the resulting program P_3 provides the answer in the simplest possible form.

This case study illustrates that loop squashing can help program comprehension by enhancing the power of the existing source code analysis and manipulation techniques. Where these techniques are used as part of a re–engineering approach, the precision of the slices produced may also offer an additional advantage over and above that offered by traditional slicing approaches.

4. Related Work

An induction variable is the simplest form of recurrence. An induction variable may be computed efficiently by use of indexing or simply as a function of a loop index. Induction variables may be used in program transformation, for example, loop fusion and loop distribution.

Generally, induction variables must be recognized before use can be made of their algebraic properties.

Ammarguellat et al. [2] suggest an approach to automatic recognition of induction variables by abstract interpretation [11, 12]. In their method, first a map, which associates each variable assigned in a loop with a symbolic form of its value, is constructed by abstract interpretation. Second, the elements of this map are unified with patterns that describe recurrence relations in which one is interested. The

We use mathematical induction to prove Rule 3 in Figure 7, which is:

If the number of iterations is n, the following loop

while $p*i+q \leq 0$ do
  $S := a*S+b*i+c$;
  $i := i+d$;
od;

can be transformed to the if statement

if $p*i+q \leq 0$ then
  $S := S*a^n+(b*i+c)*\frac{1-a^n}{1-a}+b*d*(\frac{a^n-1}{(a-1)^2}-\frac{n}{a-1})$;
  $i := i+n*d$;
fi;

Proof
(1). When n=1 the rule holds obviously.

(2). Suppose the rule holds for any given iteration number k ($k \geq 1$), then the result after the $(k+1)^{th}$ iteration can be worked out by the execution of the following statements:

$S := S*a^k+(b*i+c)*\frac{1-a^k}{1-a}+b*d*(\frac{a^k-1}{(a-1)^2}-\frac{k}{a-1})$;
$i := i+k*d$;
$S := a*S+b*i+c$;
$i := i+d$;

By pushing the second line to the end, we obtain

$S := S*a^k+(b*i+c)*\frac{1-a^k}{1-a}+b*d*(\frac{a^k-1}{(a-1)^2}-\frac{k}{a-1})$;
$S := a*S+b*(i+k*d)+c$;
$i := i+(k+1)*d$;

Now let us compute the value of S. After substituting S in the first line into the second line, S is expressed as

$S := S*a^{k+1}+(b*i+c)*(\frac{a-a^{k+1}}{1-a}+1)+b*d*(\frac{a^{k+1}-a}{(a-1)^2}-\frac{a*k}{a-1}+k)$;

where

$\frac{a-a^{k+1}}{1-a}+1 = \frac{a-a^{k+1}}{1-a}+\frac{1-a}{1-a} = \frac{1-a^{k+1}}{1-a}$

and

$\frac{a^{k+1}-a}{(a-1)^2}-\frac{a*k}{a-1}+k$
$= \frac{(a^{k+1}-1)-(a-1)}{(a-1)^2}-\frac{a*k}{a-1}+\frac{a*k-k}{a-1}$
$= \frac{a^{k+1}-1}{(a-1)^2}-\frac{1}{a-1}-\frac{a*k}{a-1}+\frac{a*k-k}{a-1}$

$= \frac{a^{k+1}-1}{(a-1)^2}-\frac{k+1}{a-1}$

So after the $(k+1)^{th}$ iteration, the state is

$S := S*a^{k+1}+(b*i+c)*\frac{1-a^{k+1}}{1-a}+b*d*(\frac{a^{k+1}-1}{(a-1)^2}-\frac{k+1}{a-1})$;
$i := i+(k+1)*d$;

as required.

Figure 9. Proof of Case 3 in Figure 7, other cases can be proved similarly

```
x := d/2;
y := h/2;
s := d*h;
i := 1;
while i <= n do
    s0 := s;
    s := s+d*h+i*h0;;
    x := x+(1-s0/s)*(i*d-x);
    y := y+(1-s0/s)*(h+i*h0/2-y);
    i := i+1;
od;
gravity := density*S;
```

Figure 10. The original program P_0

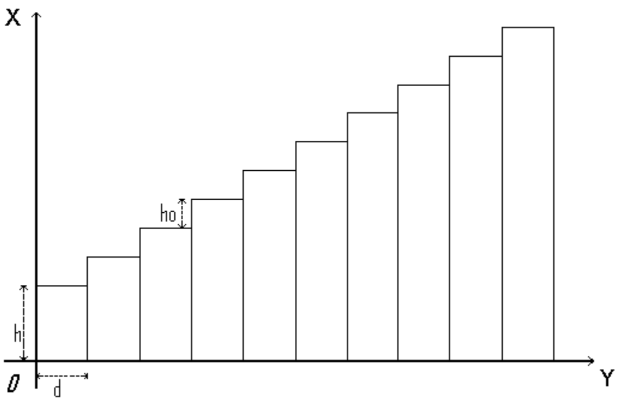

Figure 11. A shape to compute the gravity and its centre

induction variables can also be detected by finding strongly connected regions in directed graphs [37].

In this paper induction variable detection and loop transformation are considered together. The algorithm starts by detecting a loop induction variable. Provided it is found, it is used as parameter to the loop pattern matching. If successful, both the detection of the induction variable and loop conversion are achievable.

Squashing combines the use of induction variables and

```
s := d*h;
i := 1;
while i <= n do
    s := s+d*h+i*h0;
    i := i+1;
od;
gravity := density*S;
```

Figure 12. Syntax–preserving slice P_1 w.r.t variable `gravity`

```
s := d*h;
i := 1;
if i <= n then
  s := s+h0*((n-1)*i+(n-1)*(n-2)/2)+(n-1)*d*h;
  i := i+n;
fi;
gravity := density*S;
```

Figure 13. Program P_2 obtained with loop squashing algorithm

```
gravity :=
  density*(h0*(n-1)*(n-1)/2+n*d*h);
```

Figure 14. Conditioned amorphous slice P_3 w.r.t variable `gravity` and condition $n > 1$

transformation. By contrast, in the literature on source–to–source loop transformation [3, 19], the transformations preserve loop structure, *i.e.*, such transformations are from loop(s) to loop(s).

Finally, the amorphous slicing algorithm described by Binkley [6] is capable of performing a rather limited form of squashing. The transformation, referred to as loop–induction variable elimination, looks for subgraphs of a program's System Dependence Graph [27] that represent while loops only two statements. The first statement updates the induction variable. The second updates a boolean variable and is of the form "$S = S$ && $i >= 0$ && $i < n$" This rule arises from work on finding array bound violations [21] where the array has been declared to have size n. The second statement is tracking safety thus far in the variable S.

5. Conclusion and Future Work

This paper shows how loop squashing can improve amorphous slicing, making slices smaller by squashing loops into conditionals. The algorithm for loop squashing is proved correct using an inductive proof and is illustrated with a worked example.

Amorphous slicing is more suitable to reverse engineering applications than traditional, syntax–preserving slicing because it tends to produce smaller slices (they are guaranteed to be no bigger). It does this at the expense of dropping the restriction that a slice preserves the syntax of the program from which it is produced. However, for reverse engineering, this is typically an acceptable price to pay for a smaller slice, because the program is usually transformed during the re–engineering process.

Future work will consider the integration of the loop squashing transformations into an amorphous **conditioned** slicer. Such a tool would be able to detect many situations where it can be shown that the loop must execute at least once, and thereby reduce the conditional statement produced by quashing into a single assignment statement.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, techniques and tools*. Addison Wesley, 1986.

[2] Z. Ammarguellat and W. L. H. III. Automatic recognition of induction variables and recurrence relation by abstract interpretation. In *the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, New York, USA, June 1990.

[3] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, Dec. 1994.

[4] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *IEEE/ACM* 15^{th} *Conference on Software Engineering (ICSE'93)*, pages 509–518. IEEE Computer Society Press, Los Alamitos, California, USA, 1993.

[5] D. W. Binkley. The application of program slicing to regression testing. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 583–594. Elsevier, 1998.

[6] D. W. Binkley. Computing amorphous program slices using dependence graphs and a data-flow model. In *ACM Symposium on Applied Computing*, pages 519–525, The Menger, San Antonio, Texas, U.S.A., 1999. ACM Press, New York, NY, USA.

[7] D. W. Binkley and K. B. Gallagher. Program slicing. In M. Zelkowitz, editor, *Advances in Computing, Volume 43*, pages 1–50. Academic Press, 1996.

[8] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V., 1998.

[9] G. Canfora, A. Cimitile, A. De Lucia, and G. A. D. Lucca. Software salvaging based on conditions. In *International Conference on Software Maintenance (ICSM'96)*, pages 424–433, Victoria, Canada, Sept. 1994. IEEE Computer Society Press, Los Alamitos, California, USA.

[10] A. Cimitile, A. De Lucia, and M. Munro. Identifying reusable functions using specification driven program slicing: a case study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'95)*, pages 124–133, Nice, France, 1995. IEEE Computer Society Press, Los Alamitos, California, USA.

[11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *the 6th ACM Symposium on principles of Programming Languages*, pages 269–282, Jan. 1979.

[12] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. *ACM SIGPLAN Notices*, 31(1):178–190, Jan. 2002.

[13] S. Danicic, M. Daoudi, C. Fox, M. Harman, R. M. Hierons, J. Howroyd, L. Ouarbya, and M. Ward. Scalable conditioned slicing. *Journal of Systems and Software*, 2003. Conditional accept subject to minor revision. (Notified: July 8th 2003).

[14] S. Danicic, A. De Lucia, and M. Harman. Building executable union slices using conditioned slicing. In 12^{th} *International Workshop on Program Comprehension (IWPC 2004)*, Bari, Italy, June 2004. To appear.

[15] S. Danicic, C. Fox, M. Harman, and R. M. Hierons. ConSIT: A conditioned program slicer. In *IEEE International Conference on Software Maintenance (ICSM'00)*, pages 216–226, San Jose, California, USA, Oct. 2000. IEEE Computer Society Press, Los Alamitos, California, USA.

[16] M. Daoudi, S. Danicic, J. Howroyd, M. Harman, C. Fox, L. Ouarbya, and M. Ward. ConSUS: A scalable approach to conditioned slicing. In *IEEE Working Conference on Reverse Engineering (WCRE 2002)*, pages 109 – 118, Richmond, Virginia, USA, Oct. 2002. IEEE Computer Society Press, Los Alamitos, California, USA. Invited for special issue of the Journal of Systems and Software as best paper from WCRE 2002.

[17] A. De Lucia. Program slicing: Methods and applications. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, Florence, Italy, 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[18] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension*, pages 9–18, Berlin, Germany, Mar. 1996. IEEE Computer Society Press, Los Alamitos, California, USA.

[19] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):305–327, July 1987.

[20] C. Fox, S. Danicic, M. Harman, and R. M. Hierons. ConSIT: a fully automated conditioned program slicer. *Software—Practice and Experience*, 34:15–46, 2004. Published online 26th November 2003.

[21] M. Harman, D. W. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, Oct. 2003.

[22] M. Harman and R. M. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[23] M. Harman, L. Hu, M. Munro, X. Zhang, D. W. Binkley, S. Danicic, M. Daoudi, and L. Ouarbya. Syntax-directed amorphous slicing. *Journal of Automated Software Engineering*, 11(1):27–61, Jan. 2004.

[24] M. Harman, L. Hu, X. Zhang, and M. Munro. GUSTT: An amorphous slicing system which combines slicing and transformation. In 1^{st} *Workshop on Analysis, Slicing, and Transformation (AST 2001)*, pages 271–280, Stuttgart, Oct. 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[25] M. Harman, L. Hu, X. Zhang, M. Munro, S. Danicic, M. Daoudi, and L. Ouarbya. An interprocedural amorphous slicer for WSL. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 105–114, Montreal, Canada, Oct. 2002. IEEE Computer Society Press, Los Alamitos, California, USA. Selected for consideration for the special issue of the Journal of Automated Software Engineering.

[26] S. Horwitz, J. Prins, and T. Reps. Integrating non–interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[27] S. Horwitz, T. Reps, and D. W. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[28] R. Komondoor and S. Horwitz. Effective automatic procedure extraction. In *11th IEEE International Workshop on Program Comprehension*, page To appear., Portland, Oregon, USA, May 2003. IEEE Computer Society Press, Los Alamitos, California, USA.

[29] B. Korel and J. Rilling. Dynamic program slicing in understanding of program execution. In 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)*, pages 80–89, Dearborn, Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA.

[30] B. Korel and J. Rilling. Program slicing in understanding of large programs. In 6^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'98)*, pages 145–152, Ischia, Italy, 1998. IEEE Computer Society Press, Los Alamitos, California, USA.

[31] A. Lakhotia and J.-C. Deprez. Restructuring programs by tucking statements into functions. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 677–689. Elsevier, 1998.

[32] L. Ouarbya, S. Danicic, D. M. Daoudi, M. Harman, and C. Fox. A denotational interprocedural program slicer. In *IEEE Working Conference on Reverse Engineering (WCRE 2002)*, pages 181 – 189, Richmond, Virginia, USA, Oct. 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[33] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[34] M. Ward. *Proving Program Refinements and Transformations*. DPhil Thesis, Oxford University, 1989.

[35] M. Ward. The formal approach to source code analysis and manipulation. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 185–193, Florence, Italy, 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[36] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[37] M. Wolfe. Beyond induction variables. In *the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, CA, USA, June 1992.