Specializing C and x86 Machine-Code Software with OS Assistance

by

Michael B. Vaughn

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2024

Date of final oral examination: 01/16/2024

The dissertation is approved by the following members of the Final Oral Committee:

Thomas Reps, Professor, Computer Sciences
Remzi Arpaci-Dusseau, Professor, Computer Sciences
Somesh Jha, Professor, Computer Sciences
Parameswaran Ramanathan, Professor, Electrical & Computer Engineering

To my parents, for everything.

Acknowledgments

I am beyond grateful to so many people. I find that much of the joy in life is inextricable from its winding nature; we branch off down different paths, go off into the woods, find an odd little cul-de-sac and gaze at the flowers. Innumerable meetings and interactions, large and small have brightened my days over the past eight years, and condensing it into one or two pages seems a little bit wrong. All that is to say that if you, dear reader, find that I have missed you, it is not an intentional slight on my part.

I wouldn't have completed this without Tom Reps; his patience and guidance kept me on track throughout the construction of GenXGen, and his astonishing breadth and depth of knowledge was an invaluable resource in completing this work. Anybody who's worked with him knows he is sincerely, deeply, committed to his students' success, and it shows in his (seemingly indefatigable) work ethic. I am grateful for his advice on good academic writing, on how to break down a research problem, and for our chats on a little bit of everything: math, literature, theater, travel, life in general. I also wouldn't have found my way into computer science research without the mentorship of Remzi Arpaci-Dusseau in undergrad, and he has been a source of good advice in the years since. I am also thankful for my other committee members, Parmesh Ramanathan and Somesh Jha for their advice and insight.

One of the joys of working in the PL group at UW is how friendly and

warm the community is. There were so many people I've bounced ideas off of, groused about bugs to, and generally just enjoyed the company of in my time here. Loris D'Antoni's mentorship early in my PhD was invaluable, and not many faculty members anywhere have such an excellent repertoire of close-up magic. My officemates John Cyphert and Jordan Henkel, who are just plain kind and funny, and generally two people you are glad to (Tom, please ignore the rest of the sentence) do the weekly Ken Jennings Trivia Newsletter with/chat about film and video games with/get a whiteboard lecture on general relativity from. Jason Breck is one of the kindest and most encouraging colleagues around, and a great person to run ideas past. David Bingham-Brown is a C++ wizard who helped me debug some truly gnarly stuff, and is always game for a chat about e.g., your favorite 60s/70s session musicians. Tom Johnson, Eric Schulte, Junghee Lim, Chi-Hua Chen, Evan Driscoll, and many others at GrammaTech were great help in troubleshooting my increasingly arcane STK scripts and helped save me untold hours. There are also so many great staff members in the department: Angela Thorp and Gigi Mitchell are tireless, immensely helpful, and great conversation.

I would also like to thank all of my friends and family. My parents, who I love dearly, instilled an appreciation for education — and more importantly, curiosity — in me from a young age. Their kindness, warmth, and encouragement helped me throughout grad school. I have so many friends, I don't even know where to start. Jon Sieg is basically my brother, and we've shared so many road trips and pizzas. Of course, I can't leave out the rest of my friends from our decade-old group chat. Joe Bogumill, Scott Ondracek, and Tyler Blach are some of the funniest, most encouraging people around. Liz Keyes and Nate Arendt are two of the kindest people I know, and I'm grateful for giving me an excuse to run off and dogsit for many an afternoon. I also want to thank Mary Pei for her encouragement over the years, it's great having a PhD student from a wildly different part

of academia to commiserate with. Finally, thanks to Roger Pei, Eric Britigan, Laraine Zimdars, Allison Neumann, Matt Borysewicz, Jack Ehlers, and Darcy Strayer for being so wonderful, open-hearted, and encouraging. Here's to many more years of Summer Friendfest.

This work was supported, in part, by a gift from Rajiv and Ritu Batra; by ONR under grants N00014-17-1-2889 and N00014-19-1-2318; and by the UW-Madison OVCRGE with funding from WARF.

Contents

List of Tables

List of Figures

Abstract

There is an intrinsic tension between the incentives for developers of commodity software and the desires of users of commodity software. Developers of commodity software must support a multitude of users and use cases, and must often develop and maintain large sets of optional features to address and anticipate diverse use cases. Individual organizations and users often have specific, well-defined use cases that depend only on a subset of a program's features. From the perspective of the end user, unused features constitute bloat, which has ramifications in terms of program size, performance, and attack surface. Thus, for end-users, it is desirable to have an automated means of producing programs specialized for their use case.

One means of performing program specialization, is via a *partial evaluator*. A partial evaluator takes as input a subset of a program's input (referred to as *static input*), and identifies a portion of the program's text that can be executed safely on the static input. The partial evaluator executes the "safe" portion of the program on the static input, performing an exploration of the partial program's state-space. In the course of this state-space exploration, the partial evaluator simplifies the the program, using information computed from the static input. Partial evaluation can remove unreachable code, and perform optimizations such as constant propagation, loop-unrolling and function inlining. For low-level languages such as C and machine code, performing partial evaluation on programs

of non-trivial complexity requires solving problems related to the state-space exploration of partial programs, particularly (i) saving and restoring program states, and (ii) identifying previously visited states.

In this thesis, I describe GenXGen, a system for partially evaluating programs written in C and x86 machine code. With GenXGen, I improve on the state-of-the-art for problem (i) by creating an OS-assisted mechanism to save and restore states. By using additional information made available by the OS, I improve on previous techniques for problem (ii) by using incremental-hashing techniques to implement an $O(1)$ technique for identifying previously visited states (with high probability). In addition, I improve on the scalability of existing tools by using program-slicing techniques to identify the "safely executable" portion of a program's code. These techniques allow GenXGen to produce specialized versions of real-world Linux programs.

Chapter 1

Introduction

Commodity software poses a quandary for the performance and security-minded. The received wisdom in systems design is that the end user knows their workload best, and thus is best suited to make, e.g., performance-relevant design choices [Saltzer et al., 1984]. However, software is complex, money and developer hours are finite, and thus organizations necessarily rely on commodity programs and libraries for the majority of their needs. Widely adopted commodity software, by its nature, tends to support large sets of features to meet the diverse use-cases of end-users. However, any given user may only need a subset of a program's features.

Unused features constitute "bloat" in terms of binary size, program performance, and attack surface. Users may configure a program via flags or configuration files, thus incuring overhead in terms of parsing and dispatch code. Given better knowledge of a given workload, a developer may have been able to, e.g., unroll program loops or hard-code pre-computed data into the program. Moreover, attackers may attempt to re-activate unused code, directly as in cypher-suite downgrade attacks, or indirectly as raw material for, e.g., ROP attacks.

A means of producing specialized versions of programs that only include features relevant to a given use case would be a useful tool for

simplifying and hardening COTS software. In particular, given certain configuration settings, a developer or administrator may wish to remove features irrelevant to their particular configuration, thereby improving space usage and performance, and reducing the program's attack surface.

Constructing a tool that provides this "debloating" functionality is an instance of the general problem of constructing *program specializers*. A program specializer can be thought of as a program that, when provided with a program P and concrete values for *some* of P's input, produces a new program P′. This new program is specialized with respect to the static concrete inputs.

For example, consider the problem of specializing a substring-matcher with respect to a given target string. The C procedure `match` in Fig. 1.1(a) is an implementation of an $O(|s||p|)$ naive substring-matching algorithm. It returns 1 if and only if the string pointed to by `s` contains the string `p` as a substring. Note that `s` and `p` are presumed to point to valid C strings, and thus `match` terminates whenever the null terminator (ASCII 0) for `s` is encountered.

We may wish to specialize `match` with respect to the case where `p` points to the string "`hat`". One possible specialized program for this case is illustrated by the procedure shown in Fig. 1.1(b). In this version, the inner loop has been unrolled, and all manipulations and uses of `pat` and `p` have been eliminated: the characters in "hat" are hard-coded into the tests in the specialized procedure.

One approach for obtaining the procedure `match_s` from `match` and the input pattern "hat" is *partial evaluation*. A *partial evaluator* is essentially a non-standard interpreter for a programming language, which executes a program over *partial states*, and at each program point simplifies the current statement with respect to the current partial state. More specifically, a partial evaluator is a program that takes three inputs:

1. A program to specialize

```
int match(char *p, char *s) {
  while(*s != 0) {
    char *s1 = s;
    char *pat = *p;
    while(1) {
      if(*pat == 0) return 1;
      if(*pat != *s1) goto l;
      pat++; s1++;
    }
l:  s++;
  }
  return 0;
}
```

(a)

```
int match_s(char *s){
  while(*s != 0){
    char *s1 = s;
    if(*s1 != 'h')
      goto l;
    s1++;
    if(*s1 != 'a')
      goto l;
    s1++;
    if(*s1 != 't')
      goto l;
    s1++;
    return 1;
l:  s++;
  }
  return 0;
}
```

(b)

Figure 1.1: (a) String-matching program `match`; (b) `match` partially evaluated on `p = "hat"`.

```
int gen_match(char *p){
  printf("int match_s(char *s){\n");
  printf("while(*s != 0){\n");
  printf("char *s1 = s;\n");
  char *pat = *p
  while(1){
    if(*pat == 0){
      printf("return 1;\n")
      break;
    }
    printf("if(%d != *s1)\n", *pat);
    printf("goto l;\n")
    pat++;
    printf("s1++;\n")
  }
  printf("l: s++\n")
  printf("}\n")
  printf("return 0;\n")
  printf("}\n")
}
```

Figure 1.2: A generating extension that specializes `match` with respect to concrete values of `p`.

2. A partition of the program's input variables into *static* and *dynamic* sets.

3. A concrete assignment to each variable in the static set.

For every point in the program, a partial evaluator extends the partition of input variables into static and dynamic sets to cover all of the variables in the program. For each point in the program, the partial evaluator identifies all variables that depend solely on the static input variables, placing all such variables in the static set. Conversely, any variables that cannot be computed based solely on the static set are placed in the dynamic set.

In the case of `match`, the static input set consists of `p`, while the dynamic input set consists of `s`; the static set can be safely extended to contain `pat`, and the dynamic set is extended to contain `s1`. The concrete assignment to variables in the static input set is `p = "hat"`.

The partition of the program variables is such that every unboxed statement and expression in Fig. 1.1(a) can be evaluated over partial states consisting of concrete values of `p` and `pat`. Informally, a partial evaluator executes the inner loop of `match_s`, and simplifies the boxed statements with respect to the current value of `pat` at each iteration. Each time the loop body is evaluated in the course of unrolling, the statement `if(*pat != *s1) goto l;` can be simplified by substituting the concrete value of `*s1` into the condition.

In general, a partial evaluator may be able to identify parts of a program's control-flow graph (CFG) that are unreachable given particular configuration settings, and produce a residual program that does not contain the identified parts. Moreover, code in the program that is dependent solely on the static inputs can be executed by the partial evaluator, and elided from the resulting specialized program. In practice, these abilities allow a partial evaluator to perform a multitude of optimizations, without the developer of the partial evaluator needing to write explicit implementations of each optimization [Jones et al., 1993]. For example, a partial

evaluator will perform removal of unreachable code and constant folding, as well as more sophisticated optimizations, such as loop-unrolling and function in-lining. For debloating, a partial evaluator can (i) simplify code so that the resulting program incorporates specific features based on particular configuration parameters, and (ii) collapse abstraction layers in the original program via function in-lining.

In some contexts—including in my work—an alternative formulation of the above approach, based on the creation of *generating extensions*, is more desirable. Informally, a generating extension is a program-specific specializer. For example, `gen_match` in Fig. 1.2 is a generating extension that takes a pattern string, and produces a version of `match` specialized with respect to the pattern. In particular, when given the input `p = ‘‘hat’’`, `gen_match` produces the procedure in Fig. 1.1(b).

Generating extensions have the advantage that they can execute as native programs, avoiding the need to interpret the target language *at specialization time*. This avoidance of interpretation can be thought of as a *shallow embedding* of the static program in the subject language of the specializer. The notions of shallow and deep embeddings originate in research into implementation of domain-specific languages (DSLs) [Gibbons and Wu, 2014]. An implementation of a DSL is considered a *deep embedding* in the implementation language if the developer implements code mapping DSL source to AST data-types representing the syntax of the DSL, along with the necessary procedures to implement the DSL semantics as a traversal of the ASTs. For example, a deep embedding of a basic language of arithmetic expressions would implement an AST representing arithmetic expressions, along with per-node code to map each node to its interpretation.

Thus, arithmetic constants in the DSL would give rise to AST nodes representing constants, which the semantics would map to arithmetic

values. Similarly, arithmetic operations in the DSL would give rise to nodes representing operations, with each operand as a subtree, and the procedures implementing the semantics would map each node to a value in the implementation language, typically via a recursive traversal of the AST.

Conversely, an implementation of a DSL is a shallow embedding if DSL code is mapped directly to operations in the implementation language. For example, a shallow embedding of a DSL for arithmetic expressions would simply map the constants directly to numeric constants in the implementation language, while the operations are mapped directly to the underlying implementation-language operations;[1] that is, the DSL code is translated *directly* to the implementation language.

Similarly, a generating extension like the one pictured in Fig. 1.2 can be thought of as a shallow embedding of a C program specializer into the semantics of C. The key insight, which is subtle but important, is that given an appropriately chosen partition of the subject program into static and dynamic sets, such that all static code depends only on statically computable values, the static portion of a program written in language L is a perfectly usable shallow embedding of itself into L. Similarly, the dynamic portions, if they don't contain static values, can simply be wrapped in a print statement.

Moreover, a pre-made generating extension can be delivered to an end user who wishes to specialize a program without needing to deliver additional special-purpose tools for specializing programs. *For these reasons, I chose to work with generating extensions. In particular, I chose to work with generating extensions for x86 machine-code and C.*

For any specialization system to be applicable to non-trivial commodity software in either C or machine code, three main problems must be solved:

[1] Or procedures implementing the operations, as in [Gibbons and Wu, 2014].

1. *Binding-time analysis* (BTA), the process that partitions variables into static and dynamic sets exerts significant influence on the efficacy of the subsequent specialization phase. Binding-time analyses that mark relatively little code as dynamic may cause specialization to diverge, and overly conservative binding-time analysis that marks too much code as dynamic limits the amount specialization that can be done.

2. Implementing the specialization phase that executes the static portion of the program and specializes the dynamic portion entails solving several state-management issues. In particular, a specializer must be able to save, restore, and compare program states.

3. Generating extensions must be able to correctly and efficiently use statically known values to rewrite dynamic code at specialization time. In particular, heap and stack addresses known at specialization time cannot simply be written into the residual program. Additional bookkeeping must be performed to ensure that the address written into the residual program is a valid reference to the correct residual-program memory object.

In addition, to be of practical use for specializing commodity hardware, a generating-extension system targeting C code must be able to produce generating extensions for programs with numerous `.c` and `.h` files, and whose builds are orchestrated by, e.g., a makefile. Such a system must be able to extract the appropriate information from the makefile and construct a generating extension, and the residual code must be in a form that can be compiled into a working program.

No existing natively-executing generating-extension-based specializer for C or x86 machine code has addressed all of these in a satisfactory manner. Machine-code specializers such as WIPER [Srinivasan and Reps, 2015] and LLPE [Smowton, 2014] can handle problems (1), (2), and (3), but are

interpretation-based partial evaluators, rather than generating-extension systems. Interpretation-based systems are capable of manipulating sophisticated representations of state and code, which are unavailable to a generating extensions in the style of Fig. 1.2, which runs natively on hardware without interpretation. Moreover, these systems handle problem (2) in an inefficient way, and specialization is, in the worst case, $O(N^2)$, where N is the number of states[2] that arise.

The state-of-the-art generating-extension-generator for C, C-MixII [Makholm, 1999], still must resort to interpreter-like techniques to handle problem (3) in many cases. In addition, C-MixII solves problem (1) with a conservative binding-time analysis that constrains the amount of possible specialization. Moreover, C-MixII cannot process large programs produced by `make` without significant manual intervention, and in practice struggles to function usefully on programs larger than ten source files.

In my work, I constructed GenXGen, a system that produces generating extensions that perform classical partial evaluation on non-trivial real-world programs. In doing so, I constructed a general-purpose langauge-agnostic generating-extension runtime that can be repurposed for other low-level languages whose compiled code adheres to the System V AMD64 ABI [Michael Matz, 2012], or other similar C/Unix-style ABI. Moreover, these generating extensions solve problems (2) and (3), while remaining a shallow embedding of a program specializer—that is, every expression from the static portion of the subject program is translated to the generating extension verbatim, and executes natively. The semantics of the *static statements themselves* are not augmented with additional semantics for saving and restoring states, or operations to aid in computing state equality. In other words, the essence of my approach is that the generating extension uses a shallow embedding, where the language of the specializer is ***native code, together with the primitives of the underlying operating system***.

[2] In WiPEr, states are represented as AVL trees.

In resolving issues (1), (2), and (3) for GenXGen, I made the following contributions:

1. I harnessed an improved method for program slicing to create a more precise binding-time analysis for generating-extension generation
2. I developed OS-assisted state-representation and state-management techniques that enable efficient program specialization
3. I developed generating-extension-construction techniques that admit specialization of non-trivial real-world programs.

GenXGen's runtime system, which solves problem (2) efficiently, is language agnostic. Because of this property, I was able to implement two different versions of GenXGen: GenXGen[C], a generating-extension system for C programs, and GenXGen[MC], a generating-extension system for machine code.

1.1 Overview of Problems

In its classical formulation[Jones, 1988; Jones et al., 1993] partial evaluation is implemented as a two-phase process, consisting of:

1. Binding-time analysis (BTA)
2. Specialization

Given the desired partition of the inputs into static and dynamic sets, BTA extends the partition to the program's variables at all program points, identifying variable occurrences that can safely be included in partial states. The specialization phase executes the program over partial states, starting with an initial partial state (e.g., $[p \mapsto \text{“}hat\text{''}]$), simplifying the program as it executes.

This classical conception of partial evaluation—a transformation that performs general-purpose computation over a subset of the program, and

uses the statically computed values to simplify other parts of the program—is generally accepted to date back to the 1970s [Jones et al., 1993; Sestoft and S: Gfndergaard, 1988] in the work of Futamura [Futamura, 1971], and subsequently explored by foundational authors, including Turchin [Turchin, 1986], Beckman [Beckman et al., 1976], and Ershov [Ershov, 1977]. Broadly speaking, this view of partial evaluation can be thought of as implementation-oriented, centered on state-space-exploration and constant-folding.

Over the subsequent decades, researchers attempted to produce partial-evaluation systems capable of specializing non-trivial programs written in industry-standard languages. Tools such as C-Mix and C-MixII, for example, produce generating extensions for C code, and others have implemented systems such as LLPE for partially evaluating intermediate representations. These systems, while impressive and technically sophisticated, nonetheless struggle to scale to programs larger than ten source files, and this style of partial evaluation has found little industrial use.

However, many modern systems, such as PyPy and Trimmer perform program specializations that produce specialized code that is structurally similar to that produced by partial evaluators, with, e.g., loop unrolling, constant propagation, and procedure inlining. In these systems, program specialization is implemented in a variety of ways. Some program-specialization systems, such as Trimmer [Sharif et al., 2018a], are implemented as a collection of disjoint optimizations, each of which is essentially a more powerful version of a standard compiler optimization, such as loop-unrolling or constant propagation.

Others, like Truffle, do perform classical partial evaluation over constrained subsets of a program's code. Truffle is a system that takes as input an interpreter for a language, and produces a high-performance JIT compiler by partially evaluating portions of the interpreter that was passed as input. However, even Truffle is not a general-purpose partial

evaluator in the sense of Futamura and Jones: it is a partial evaluator for interpreters, and the interpreters must adhere to a specific "specialization-aware" coding style.[3]

The members of this spectrum of modern program-specialization tools are generally referred to as "partial evaluators," despite deviating significantly from the original general-purpose conception of the technique. This situation is natural and reasonable: in most contexts, producing useful, fast production software *should* win out over theoretical purity. However, the lack of distinction in the techniques used is often a point of confusion for readers (and writers) of the literature on program specialization.

Thus, for the purposes of this dissertation, the term partial evaluation will be taken to mean, and used interchangeably with, *classical partial evaluation* in the style of Futamura and Jones.

To implement GenXGen, I improved upon the state-of-the-art for generating extensions in three areas:

1. Binding-time analysis
2. Specialization-phase state management
3. Processing the structure of real-world software projects.

1.1.1 Binding-Time Analysis

In the literature on partial evaluation, there are a variety of methods for producing a partition of a program's variables into static and dynamic sets. Systems such as C-Mix[Andersen, 1994a] have used type-inference, while others [Jones et al., 1993] have used data-flow analysis, and yet others have used abstract interpretation [Consel, 1990].

[3]Jones observed that any interpreter, to be specializable in a meaningful way, must be implemented in a "specialization-aware" manner. The point here is that the Truffle VM is specifically a partial evaluator for interpreters, and exposes domain-specific control and data primitives to the program being specialized, and its architecture cannot be readily adapted to specialize other kinds of programs.

Regardless of the technique chosen, there are many possible partitions that a BTA algorithm could produce. A BTA algorithm is acceptable for our purposes as long as the partition that it produces for each program point is *congruent* [Jones et al., 1993].

Informally, congruence ensures that in every subject-program statement that updates a static variable, the update to the static variable does not depend on any dynamic values. A partition of the variable occurrences at the different program points of p into static and dynamic sets (V_s and V_d, respectively) is congruent if at every statement l in P where a variable $v \in V_s$ is updated, the new value of v is computed solely from variables in V_s. Congruence is important because it ensures that the partial state induced by the set of static inputs can always be safely updated during specialization.

A BTA algorithm can use *forward slicing* [Weiser, 1981; Horwitz et al., 1990] to compute a congruent partition of a single-procedure program. Given a set of variables V and a set of program points L, forward slicing computes the the set of program points that may be affected by the values of V at points in L. For BTA, we compute the forward slice from the *dynamic inputs*. The boxed statements in Fig. 1.1(a) show the program points included in the forward data-dependence slice starting at formal parameter `s`.

A congruent partition of the program-variable occurrences is implicit in the slice. The forward slice contains all assignments to, and uses of, variable occurrences that are transitively dependent on `s`, while the complement of the slice contains all assignments to and uses of variable occurrences *not* dependent on `s`. Thus, to ensure that the specialization phase only performs safe updates, it executes only the statements in the complement of the slice. Moreover, slicing can be viewed as an extension of BTA results from variable occurrences to statements: all statements dependent only on static state are marked as static; the remainder are

```
int g;
int p(int s, int d) {
    int rs, rd;
    res = 0;
    rs = q(s, d);
    rd = g;
    rd = q(d, s);
    rs = g;
    res += rs;
    res += rd;
    return res;
}
int q(int a, int b){
    return a + 1;
    g = b;
}
```

(a)

```
int g;
int p(int s, int d) {
    int rs, rd;
    res = 0;
    rs = q(s, d);
    rd = g;
    rd = q(d, s);
    rs = g;
    res += rs;
    res += rd;
    return res;
}
int q(int a, int b){
    return a + 1;
    g = b;
}
```

(b)

Figure 1.3: (a) A standard forward slice, which exhibits the parameter-mismatch problem. (b) The results after applying Binkley's algorithm to eliminate the parameter mismatch.

marked as dynamic.

However naive applications of forward slicing to multi-procedure programs lead to the *parameter-mismatch problem*. Consider the program in Fig. 1.3, in which we flag `d` as dynamic and perform a standard forward slice.

By examination, we can see that the value of `rd` always depends on `d`, while `rs` always depends solely on `s`. The standard (summary-edge-based) forward slice in Fig. 1.3(a) correctly leaves `rs` out of the slice in `p`. However, standard forward slicing is *monovariant*; in this example, there is only one version of procedure `q` in the sliced program, but the slice contains two different callsites at which `q` is called, one of which has the first actual-in parameter [4] in the slice, and the other has the second actual-in in the slice.

[4]Actual-in parameters are the expressions passed as parameters at a specific call-site. Formal-in parameters are the parameters in a procedure's definition. For example, at the

Thus, both formal parameters of q and both statements in q's body must be in the slice.

This slicing result violates congruence: both assignments to rs are dependent on dynamic statements in q, and cannot be safely executed at specialization time.[5]

One standard way of rectifying the parameter mismatch problem is *Binkley's Algorithm* [Binkley, 1993]. For every pair of formal and actual-out parameters for which the formal-out is in the slice, but the actual-out is not, we perform a forward slice from the formal-out. The re-slicing is repeated until there are no more parameter mismatches.

However, as shown in Fig. 1.3(b), Binkley's Algorithm yields extremely conservative results that are suboptimal for specialization purposes. Here, congruence is recovered at the expense of leaving no static code that can be meaningfully eliminated.

The crux of the problem is the aforementioned monovariance of slicing. A conventional slicing algorithm must incorporate the information from all of q's callsites into its only representation of q. What is desired is a slicing algorithm that can behave as if there are two different versions of q, one for each binding-time pattern of its input parameters.

Specialization slicing [Aung et al., 2014], operationalizes this idea

first call to q in Fig. 1.3, s is the actual-in parameter in p corresponding to the formal-in a in q. Symmetrically, there exist formal-out and actual-out parameters: `return res` defines an (unnamed) formal-out parameter that represents the return value of q, and there is an actual-out parameter representing the return value of of q at every call-site. Formal and actual parameters need not be explicitly defined parameters, either. The program representation used to compute slices in GenXGen does not have global variables in the traditional sense: global variables are modeled as local variables passed around as implicit parameters of the procedures. For example g in p and q is treated as two local variables passed between caller and callee via implicit actual/formal-in/out parameters.

[5]Note that the correctness-relevant parameter mismatch in forward slicing is not the immediately visible mismatch between the formal and actual-in parameters, but instead the one between the formal and actual-*out* parameters. This situation is the dual of how the mismatch problem is conventionally discussed in slicing literature, which frames the problem in terms of using a *backwards* slice to identify all program points that could affect a variable at a given point.

with an automata-theoretic slicing technique that produces polyvariant program slices that produce more precise results than Binkley, but the slices do not exhibit the parameter-mismatch problem.

1.1.2 Specialization-Phase State Management

Given a partition of program statements into static and dynamic sets, a program specializer can begin executing static code and simplifying and emitting dynamic code. However, unlike the program and generating extension in Fig. 1.1 and Fig. 1.2, not all programs can be specialized by performing a conventional linear execution of the static subset of the subject program's code. Consider the specialization of procedure `b` in Fig. 1.4 with respect to `x = 5`.

```
void b(int d, int s){
  int v = 10;
  if(d){
    v += s;
   }else{
     v -= s;
   }
   printf("%d", d + v");
}
```

(a)

```
void b_s(int d){
  if(d){
    goto post_add;
  }else{
    goto post_sub;
  }
post_add: printf("%d", d + 15);
goto exit;
post_sub: printf("%d", d + 5);
exit: return;
}
```

(b)

Figure 1.4: (a) `b` a procedure with control-flow governed by dynamic state. This cannot be specialized via simple straight-line execution over static state. (b) The desired specialization of `b` with respect to `s = 5`.

Because `v` is statically known, the computations inside each branch can be eliminated, and the value of `v` can be propagated to the argument of the final `printf`. However, normal execution clearly will not suffice because each branch is dependent on the value of the dynamic variable `d`. Moreover, each arm of the if statement yields a different static post state, with `v` equal to either 5 or 15. Thus, if a specializer is to carry partial

states forward and use that state to simplify later dynamic statements, the specializer must be able to juggle *multiple states at once*. In particular, the partial evaluator needs to be prepared to *go both ways* at a branch. That is, the specializer needs to traverse both arms of the if statement, and produce residual code for each arm, *as well as all* blocks after the arms of the if statement.

If one has a state representation that provides a convenient means for capturing and saving states, it is not so difficult to satisfy this requirement, which is one reason why most partial resemble interpreters. One can, for example, modify an existing interpreter to save and restore an interpreter's stack representation, along with mappings from symbols to values and the collection of live heap objects.

In addition, a classical specializer must be able to determine whether a state has been seen before to avoid producing redundant states and to ensure convergence where possible. Consider the procedure `f` in Fig. 1.5. The general case poses several significant challenges for the implementer of a classical program specializer. Consider specializing `f` in Fig. 1.5 with respect to `v = 0, L = 0`.

The procedure takes as input `d`, `v`, and `L`, and computes a sum that depends on all three inputs. In particular, the loop counts down from `d` to zero, accumulating every value in the range from `d` to zero. But, also, at each iteration the procedure also incorporates `v` mod `L` into the accumulator, and then increments *v*.

Given the BTA results, we know that the only static variable that changes is `v`, and that inside the loop `v` ranges over $\{1, 2\}$. Thus, we can partially unroll the loop body so that it contains one copy of the body for each value that `v` takes on. Moreover, `f` now has two exit points, depending on the value of `v` whenever the dynamic counter `d` reaches zero.

Clearly, this transformation can be done manually based on our reasoning, but how do we automate this reasoning? Unlike the example in

```
void f(int d, int v, int L){
  int sum = 0;
  while(d > 0){
  sum += v + d;
    v++;
    v = v %  L;
    d--;
  }

 printf("%d, %d", v, sum);
}
```

(a)

```
void f_s(int d){
 while(d > 0){
  sum += 0 + d;
  d--;
  if(d <= 0) goto odd_break;
  sum += 1 + d;
  d--;
 }
 printf("%d, %d", 0, sum);
 goto exit;
odd_break:
  printf("%d, %d", 1, sum);
exit:
  return;
}
```

(b)

Figure 1.5: (a) `f`, a procedure whose loop can be partially unrolled. (b) A possible `f` specialized with respect to `v = 0`, `L = 2`.

Fig. 1.1, it doesn't seem immediately implementable as a straightforward linear execution of a loop. How do we determine when to terminate the specialization of `f`?

To solve these problems, the specialization phase of a classical specializer is implemented as a special-purpose interpreter that performs a state-space exploration controlled by a worklist of (state, basic block) configurations. The specializer simplifies and emits code as it explores, using special state-management techniques to avoid exploring redundant (state, basic block) pairs.

The specializer interprets the CFG of the program, using a partial state to track the values of the variable occurrences in the static set. The interpretation is non-standard because at a condition classified as dynamic, such as the boxed conditional in Fig. 1.6(a), there are *two* successor basic blocks to interpret. A worklist is used to keep track of basic blocks that still need to be processed. Every basic block is interpreted linearly, statement-by-statement, and each statement is evaluated in one of three ways.

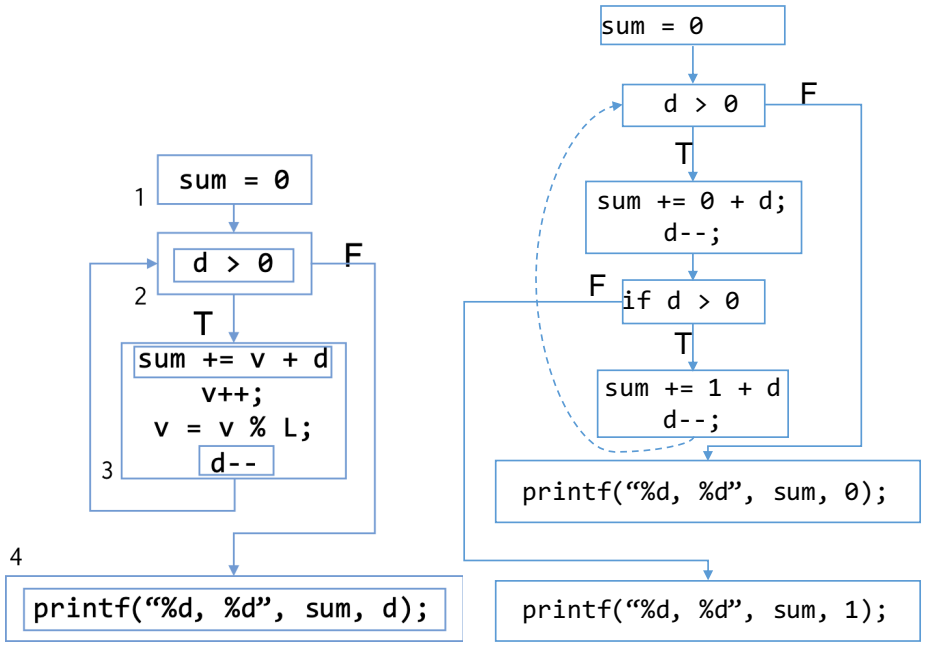


Figure 1.6: (a) and (b) are the respective original and residual versions of `f` from Fig. 1.5

1. All statements marked as "static" are evaluated, and the partial state is updated accordingly. For example, the statement `v++` will cause the value of `v` in the partial state to be incremented by 1.

2. Statements marked as "dynamic" are not evaluated, but are emitted to the residual program instead. For instance, the single occurrence of "`d--`" in the original procedure is emitted at two different times during the specialization of `f`.

3. However, some statements marked as "dynamic" cannot just be emitted as is; if a dynamic statement s depends on the value of a static variable v, the value of v must be *lifted* into the residual program's state at s. Lifting can be performed by replacing every occurrence of v in the emitted statement with the current value of

v. For example, lifting is required for the `sum += v + d` statement in the loop body of `f`: every emitted instance of the statement in Fig. 1.5(b) has `v` replaced with 0 and 1.

Unlike a standard interpreter, the specialization phase is prepared to handle control flow governed by dynamic state. Consider the flow control statement in basic block 2 in Fig. 1.6(a). Due to the comparison against the (dynamic) integer `d`, there is not sufficient information in the partial state to determine which branch will be taken. Consequently, the specializer must arrange to specialize the blocks at *both* successors.

In essence, the specializer needs to "go both ways" when encountering a branch governed by dynamic state. In practice, the specializer is generally implemented as a worklist-based algorithm: basic blocks are specialized and residuated using the approach described earlier; however, upon reaching a branch classified as "dynamic," the specializer records the current state, σ, and adds a $(\sigma, \mathtt{l})$ pair to the worklist for every successor block `l`. The specializer then removes an $(\mathtt{s}, \mathtt{b})$ pair from the worklist, and executes basic block `b`, starting with state `s`. Thus, at the basic-block level, specialization is similar to execution, except that code can also be emitted; at the end of a basic block, the specializer creates the appropriate (partial-state, basic-block) pair(s) for the block's successor(s), and inserts them into the worklist.

The partial evaluation of `f` shows that a partial evaluator needs to be able to check state equality efficiently. Consider the loop in `f`. Every time the block constituting the loop body is executed, `v` is incremented by one and reduced modulo `L`. The pair consisting of the state with the updated value of `v` and block 2 are enqueued. Every time block 2 is removed from the worklist, two successors are enqueued: one for the final `printf` and one for the loop body; this residuation of the loop body will eventually lead back to block 2. Thus, a partial evaluator that always enqueues the successor of block 2 will never terminate, despite the fact that `v` can only

take on the values from 0 up to $\mathtt{L} - 1$

To prevent this infinite unrolling, the partial evaluator must be able to detect duplicate partial-state/basic-block pairs. In particular, the first time we evaluate block 2, we want to enqueue the pair $(\sigma$, block $3)$ where σ is the state mapping $\mathtt{v}$ to $\mathtt{0}$.

After two evaluations of block 3, we have re-encountered the state-pair $(\sigma$, block $2)$. The partial evaluator will not terminate unless it can determine that $(\sigma$, block $2)$ has repeated.

Thus, a worklist-based partial-evaluation algorithm requires two key state-management features:

1. the ability to save and restore partial states,

2. the ability to efficiently check state equality

When partial evaluation is performed on a program written in a type-safe high-level language, both features can be implemented in a relatively straightforward fashion. States can be saved, restored, and compared by traversing the graph of memory objects induced by the reachability relation over the static state, in a manner similar to the walk performed by a mark-and-sweep garbage collector.

However, when creating a program-specialization tool for real-world programs, one faces a multitude of problems. For programs implemented in strongly-typed languages, a mark-and-sweep walk requires traversing all memory objects reachable from the stack. Moreover, for weakly-typed languages, like C and machine-code, the problems are intensified. In the general case, it is difficult or impossible to determine whether a value stored in a memory location or register is a memory address or an integer. This situation makes it difficult to identify the portion of memory over which the mark phase should be carried out [Boehm, 1993], and to handle cyclic data structures. Moreover, when working with machine code, memory is *undifferentiated* beyond the coarse-grained divisions into program

text, global variables, stack, and heap. Building on mathematical background described in §2.3, in Chapter 3 I describe state-management techniques that allow for efficient summarization and comparison of weakly-typed and undifferentiated program states, allowing me to implement efficient generating extensions for both machine code and C.

1.1.3 Generating Extensions in the Real World

Pointer Lifting. In §1.1.2, I described *lifting*, the act of using a value known at specialization-time to rewrite an expression marked as dynamic by the BTA. Specialization systems such WIPER and C-MixII perform the sort of lifting shown in Fig. 1.5(b), where variables in expressions are replaced by concrete values from the static state. When the value is a pointer,the implementation of lifting becomes more challenging. For example, in the course of specialization, a static variable `p` may take on the address of a heap-allocated linked-list node. However, if the specializer encounters dynamic code such as `dyn->ptr = p`, simply lifting the concrete specialization-time value of `p` is problematic. In all likelihood, the original, and residual programs will have wildly different memory layouts, and p's specialization-time address will not reference the same memory object, if it references valid memory at all.

WIPER and C-MixII address this issue by treating addresses as symbolic. WIPER, by virtue of being an interpreter, is free to represent pointers to memory objects as base/offset pairs, and can implement pointer arithmetic appropriately. C-MixII, despite being a generating extension, behaves more like an interpreter when dealing with pointer types. Pointers reference special-purpose wrapper types, and address arithmetic is rewritten and is subject to special rules. This approach compromises the key advantage of generating extensions—namely, that the static portion of the program is able to run natively on hardware. My work takes a different approach, eliminating interpreter-like behavior. Addresses are treated

normally until the point that they are lifted.

C Generating Extensions and Build Frameworks. As discussed in §1.1, existing classical partial evaluators for low-level languages like C have struggled to scale beyond small examples. Beyond the issues of binding-time analysis and state management, there are other pragmatic issues that existing systems have not satisfactorily addressed.

First, the current state-of-the-art generating-extension tool for C, C-MixII, cannot process multi-compilation-unit projects without significant manual intervention. One must essentially construct a parallel makefile to the original project, and there is no straightforward way to take C-MixII's output and reconstitute the residual code into a new multi-source-file-program. The main problem is that C-MixII's interface is purely source level, and has no means of integrating with build systems.

It would be preferable to have a more "turnkey" system, in which a user provides an initial-binding-time annotation and, e.g., a makefile, and the system produces a generating extension. Moreover, such a system should be able to produce residual source code such that the residual program can be built correctly.

Tools like LLPE and Trimmer can work with large projects. However, these systems are not generating-extension-based, and instead function as interpreters that analyze and transform a compiler's intermediate representation (IR), and thus have access to information after the compiler link phase. Because I am producing natively-executing generating extensions that performs source-to-source transformations, I do not have this luxury, and must take a different approach—namely, extracting build information from the build process, which is then used to build the generating extension and the residual program.

1.2 Overview of Results

1.2.1 Specialization-Slicing-Based BTA

Consider the (forward) Binkley slice of procedures p and q from §1.1.1, shown again in Fig. 1.7(a). Because the slice of q must incorporate information from all callsites, all output parameters[6] of q are in the slice, and the conservative nature of Binkley's algorithm causes nearly all the code in p to be put in the slice.

We can obtain better results by creating a polyvariant version of the program. In Fig. 1.7(b), we have a semantically identical version of the program, which contains a distinct copy of q for each callsite. Because each version of q is associated with exactly one binding-time pattern for its parameters, the formal-outs are guaranteed to match the actual-out parameters.

In the absence of recursion, inducing polyvariance by making one copy of every procedure for each of its callsites will produce parameter-mismatch-free slices that are more precise than those produced by Binkley's algorithm. However, copying can produce an excessive amount of redundant code. For example, a two-parameter procedure can have at most four binding-time-patterns: all parameters are dynamic, one of the two parameters is dynamic, or no parameter is dynamic. Thus, if there are more than four callsites, copying is guaranteed to produce redundant code. This situation could be particularly costly if there are widely reused utility procedures. Thus, it is desirable to use a slicing strategy that produces a minimal set of procedure copies.

Additionally, Aung et al. [Aung et al., 2014] observe that due to recursion, naive inlining will not work in the presence of recursion. To produce polyvariant slices, they propose a new algorithm: specialization slicing.

[6] Recall that a procedure's output parameters include not just the return values, but also global variables and any values or memory regions passed by reference.

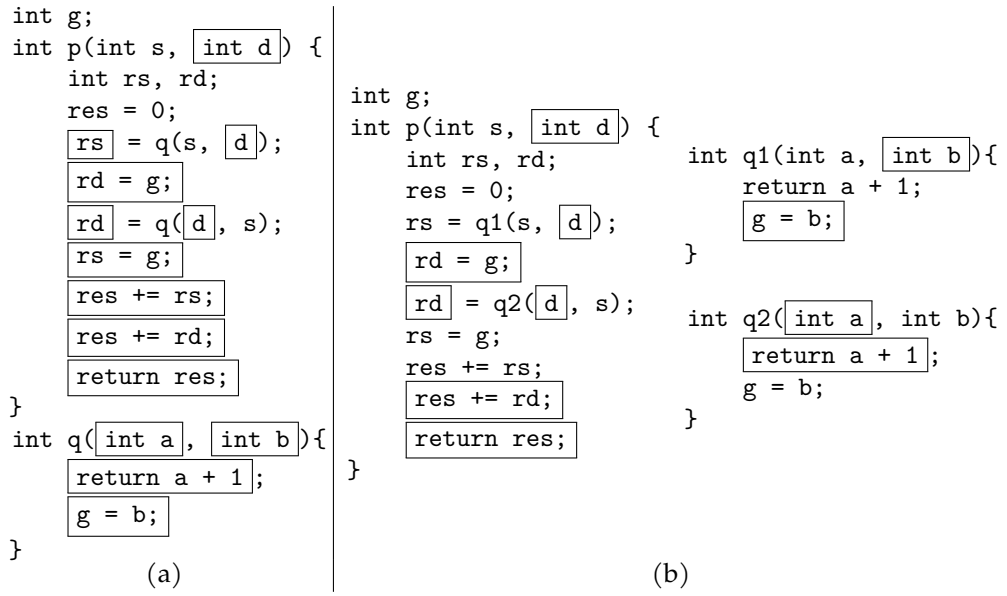


Figure 1.7: (a). The Binkley slice from Fig. 1.3(b) in §1.1.1. (b) The forward data-dependence slice on a polyvariant version of the program.

Specialization slicing works symbolically, performing automata-theoretic operations to identify the minimum set of copies of a procedure needed to capture all possible binding-time patterns induced by the slice sources. In certain cases, though, given the presence of an n-parameter procedure specialization slicing can produce 2^n copies. Experimental evaluation shows that, for the real-world programs tested, worst-case blow-up does not occur, and the automata-theoretic algorithm comprises a small proportion of the time taken to produce a generating extension. (See §6.3.1 and §6.3.2.)

I give a technical overview of SDGs and the basics of slicing in §2.2.1, and in §2.2.3 I give a high-level overview of specialization slicing. In Chapter 3, I discuss the technical details of integrating specialization slicing into GenXGen. In Chapter 6, I evaluate the effects of specialization slicing on specialization. In particular, I measure the amount of copied code, and show that it is relatively modest in practice, with at most 30% more

code added, and that the worst-case blow-up does not occur on the real programs tested.

1.2.2 OS-Assisted State Management

My initial work in program specialization was to investigate the feasibility of creating a generating-extension generator that produces generating extensions for x86 binaries, without requiring access to source code. In particular, my aim was to produce generating extensions that execute as native x86 binaries. That is, *at generating-extension execution time, there is no need for the generating extension to have a sophisticated semantic model of x86 machine code; the semantic model of x86 comes from the native instruction set, implemented in hardware*.

This approach is fundamentally different from prior approaches to machine-code specialization. WiPER [Srinivasan and Reps, 2015] is a partial-evaluation-based framework, which is able to leverage CodeSurfer/x86's [codesurfer, 2018; Anderson et al., 2003] sophisticated model[Lim, 2011] of the x86 ISA to interpret the subject program. Other existing run-time code-generation approaches, such as the one used in the Fox Project [Leone and Lee, 1996] do produce natively executing generating extensions. However, they implement a *staged-compilation* approach: binding-time annotations must be supplied by the user at the type level in the source code, and a modified compiler uses source-level information to produce the machine-code generating extensions.

In addition, because the runtime library that implements GenXGen's CFG-exploration is language-agnostic,[7] I also implemented GenXGen[C]. GenXGen[C] produces generating extensions for C code, but uses the same state-management runtime library as GenXGen[mc]. Because of limitations of the platform on which GenXGen[mc] is based—discussed

[7]It is language-agnostic, modulo the assumption that the language can be compiled to x86 machine code that uses a conventional ABI.

in §5.2.2 and Chapter 8—GenXGen[C] became the primary showcase for the techniques I developed.

My work addressed the following research question:

> Is it possible to produce generating extensions for low-level languages that (a) specialize non-trivial real-world programs, (b) execute natively, without needing to be packaged with an interpreter, and (c) perform program specialization in a time- and space-efficient fashion?

Constraints (a), (b), and (c) pose non-trivial state-management challenges not present when specializing strongly-typed high-level languages. Any program-specialization tool implementing either the generating-extension approach or the partial-evaluation approach must address two state-management problems:

1. A program specializer needs to be able to save and restore program states efficiently.

2. A program specializer uses a worklist-based algorithm that executes a program over partial program states (§2.1.3 and §2.1.5). To prevent redundant exploration of the program's state space, there needs to be an efficient means of determining whether a (partial) state has *repeated*.

Unlike source-code programs, machine-code program states consist of memory that is *undifferentiated* beyond the coarse division into regions for the stack, the heap, and global data. Moreover, for a program specializer that runs natively, the states that need to be captured and compared in issues (1) and (2) are *native hardware states* (at the level of the instruction-set architecture).

Naive approaches to these issues are extremely costly:

- A straightforward approach to issue (1) means copying the entire state for each save and restore operation.

- The need to test a new state against all states that have previously arisen (issue (2)) suggests the use of hashing. However, resolving collisions requires the ability to compare two states for equality.

These state-management operations have never been addressed in a completely satisfactory manner in prior work on program specialization — even for interpretation-based specialization of high-level languages. The best prior solution take advantage of the fact that a partial evaluator is similar to a language interpreter [Jones et al., 1993]—except that a partial evaluator operates on partial states, and an interpreter operates on full states.

One can design an abstract datatype of partial states for which saving/restoring states and identifying state repetition can be performed with low time and space overhead. In particular, the components of (partial) states can be hash-consed [Goto, 1974] so that a unique representative—i.e., a canonical address—is maintained for each partial state.[8] A set of the addresses of the unique representatives is then maintained, with hashing used to assist membership testing (and collision resolution performed by comparing addresses).

However, due to constraints (a) and (b), I did not have the option of implementing memory as an explicit data structure that can be readily swapped to save and restore states—which raises the following question:

How can 1) saving and restoring states and 2) checking state equality be handled efficiently in a generating extension that runs natively?

To address issue (1), I use two OS-level mechanisms—copy-on-write (CoW) and process context-switching—to create an efficient mechanism for *state-snapshotting and restoration*. However, the main element that al-

[8] More precisely, to support the unique-representative property, one would make use of applicative maps (see [Reps et al., 1983, §6.3] and [Myers, 1984]), hash-consing, and a hash table to detect duplicates. (The hash-code would be based on the contents of the map's entries, rather than the structure of the tree that represents the map.)

lowed us to devise a solution is that we changed the requirements associated with issue (2) slightly. In particular, *we do not insist that there be a mechanism to resolve collisions*, as long as we have control over parameters that ensure that *the probability of a collision* ***ever*** *arising is below a value of our choosing*. In other words, we allow the use of a hash, as long as the parameters of the hash function can be tuned to keep the collision probability at an acceptable level. Moreover, the hash function is *incrementally updatable*: as execution of $ge_{P,S}$ mutates one (partial) state σ_1 to another (partial) state σ_2, the hash value for σ_2 can be computed efficiently by updating the hash value of σ_1. We implement this hashing algorithm using Rabin fingerprinting [Rabin, 1981]. Although often presented as a sliding hash, the algebriac properties of the algorithm admit the construction of an in-place-updatable hash algorithm.

In §2.3, I provide a mathematical overview of the Rabin-fingerprinting algorithm. In Chapter 3, I discuss the OS-assisted state-management techniques, including my process-based state representation (§3.2) and copy-on-write-fault-based hash-update method (§3.3). In Chapter 6, I evaluate the performance of this multi-process approach on microbenchmarks and real-world programs, and show that the technique is feasible and effective for non-trivial programs. In particular, for tasks that involve removing program features, the generating extensions terminate in under one minute, and the time contributed by state hashing constitutes 10-20% of specialization time.

1.2.3 Generating-Extension Pragmatics

Pointer Lifting. As described in §1.1.3, concrete pointer values cannot be lifted into residual programs. Existing systems, even generating extensions that otherwise execute static code natively, solve the problem by treating pointers symbolicly during specialization-time. My work takes a different approach to pointer lifting, and performs lightweight *lazy sym-*

bolization at lifting time, by using debug symbols (in the case of C code), coarse-grained information about memory layout, and a special `malloc` implementation that permits identifying the allocated memory region associated with a specific address held by a pointer variable. By taking this approach, ordinary specialization-time pointer operations can be handled by hardware in the usual way, without rewriting or interpretation.

C Generating Extensions and Build Frameworks. Prior approaches to partially evaluating C code either worked at the source-file-level and could not readily process large projects, or were interpreters that specialized compiler IR. I present a method to create a generating extension for a C program, and can process makefile-based projects in an automated manner. Prior to constructing a generating-extension, GenXGen[C] uses the `strace` command to track Makefile or shell build-script execution, and collects all compiler invocations in the build. By taking this approach, GenXGen[C] can automatically produce generating extensions for files containing hundreds of source files.

In Chapter 3, I provide an overview of generating extensions, describing their basic structure, and challenges in performing tasks such as lifting. In §5.1.5, I describe the implementation details of the lazy-symbolization technique. The build-tracing technique that allows GenXGen[C] to process non-trivial projects is described in §5.2.1. In addition to these contributions, the multiprocess approach described in §1.2.2 is a novel approach to constructing a generating extension, which entails solving several smaller, but still non-trivial design and implementation problems, relating to IPC, code-generation, and the process-based state representations. The bulk of Chapter 5 describes the solutions to these problems.

1.3 Thesis Organization

Chapter 2 provides a formal overview of program specialization with an emphasis on generating extensions. In particular, Chapter 2 provides an in-depth discussion of the structure of a generating extension, as well as the state-management challenges a generating extension must solve. Chapter 2 also provides a theoretical overview of specialization slicing, which forms the basis of GenXGen's BTA algorithm, and Rabin fingerprinting, which is a key component of GenXGen's state-management technique. Chapter 3 provides the technical details of how I use the slicing algorithms from Chapter 2 to implement the BTA phase of GenXGen. Chapter 4 discusses our process-based state-representation, and the Rabin-fingerprint-based technique for state hashing. In addition, Chapter 4 describes how GenXGen constructs generating extensions, with a focus on how the generating extension architecture described in Chapter 2 is adapted to operate using the process-based state representation. Chapter 5 discusses pragmatic implementation concerns needed to implement GenXGen for real-world programs. Chapter 6 discusses the experimental evaluation of our techniques for both machine-code and C generating extensions. Chapter 7 discusses related work not covered in other sections, and provides more information on some work mentioned elsewhere. Chapter 8 provides concluding remarks.

Chapter 2

Background

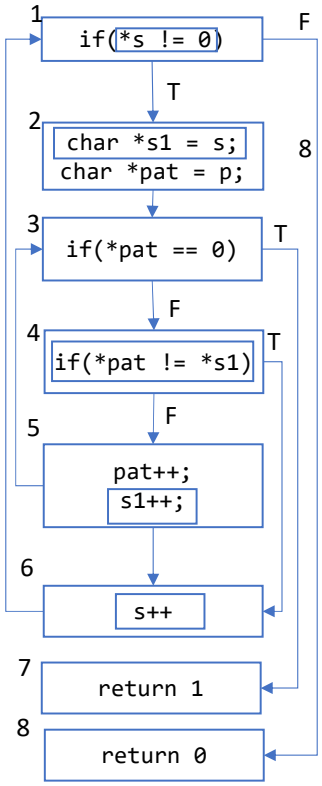
In §2.1, I give an introduction to classical partial evaluation, and explain how a generating extension works to specialize a program. In §2.2.1, I discuss forward slicing, and show that forward slicing can be used to produce results that satisfy the requisite properties of a valid BTA, and thus can be used as a valid, albeit imprecise, BTA. I also show how an alternative slicing strategy can be used to obtain better precision. In §2.3, I present the algebraic properties of Rabin's fingerprinting scheme, emphasizing how to use it as an in-place-updatable hashing algorithm, which will be used as a component in an efficient version of the classical specialization algorithm from §2.1 that is suitable for specializing programs written in low-level languages.

2.1 A Précis on Partial Evaluation

This section provides an overview of classical partial evaluation. §2.1.1 provides a formal definition of the desired functional relationship between a partial evaluator's inputs and outputs. Classical partial evaluation is a *two-phase* algorithm, which consists of (1) a binding-time-analysis phase, and (2) a specialization phase. §2.1.2 defines the formal properties

```
int match(char *p, char *s){
  while(*s != 0){ // block1
    char *s1 = s; // block 2
    char *pat = p;
    while(1) {
      if(*pat == 0) // block 3
        return 1; // block 7
      if(*pat != *s1) // block 4
        break;
      pat++; s1++; // block 5
    }
    s++; // block 6
  }
  return 0; // block 8
}
```

(a)



(b)

Figure 2.1: (a) String-matching procedure `match`; (b) the CFG of `match`.

that a valid binding-time-analysis algorithm's results must satisfy. §2.1.3 discusses the classical worklist-driven specialization algorithm. §2.1.4 discusses the need for an efficient state-management strategy in a classical partial evaluator or generating extension (a stand-alone, program-specific partial evaluator). §2.1.5 provides a concrete example of how the classical specialization algorithm from §2.1.2 and §2.1.3 can be instantiated in a generating extension.

2.1.1 Partial Evaluation: A Functional Definition

A partial evaluator *pe* takes as inputs (i) a program P (expressed in some language L); (ii) a partition of P's inputs into two sets, *static* and *dynamic* (for short, S and D, respectively); and (iii) an assignment $A(S)$ to the variables in S. As output, *pe* produces a *residual program* $P_{A(S)}$ that is specialized with respect to $A(S)$. Letting $[\![\cdot]\!]$ denote the meaning function for the language in which *pe* is written, we have[1]

$$[\![pe]\!](P, A(S)) = P_{A(S)}, \tag{2.1}$$

The requirement on $P_{A(S)}$ is

$$[\![P_{A(S)}]\!]_L(A(D)) = [\![P]\!]_L(A(S \cup D)), \tag{2.2}$$

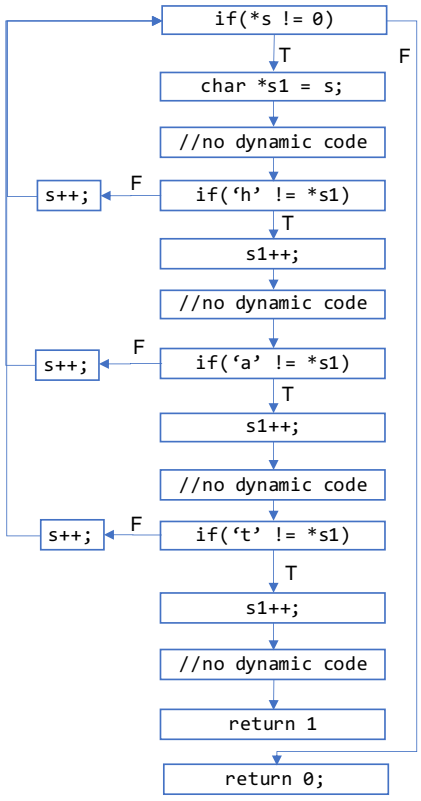
where $[\![\cdot]\!]_L$ is the meaning function for L. That is, $P_{A(S)}$ with input $A(D)$ produces the same output as P with input $A(S \cup D)$; however, $P_{A(S)}$ has fewer input arguments, and is specialized with respect to the assignment $A(S)$.

The C procedure `match` in Fig. 2.1(a) is an implementation of an $O(|s||p|)$ substring-matching algorithm. It returns 1 if and only if the string pointed to by `s` contains the string `p` as a substring. Note that `s` and `p` are presumed to point to valid C strings, and thus `match` terminates whenever the null terminator (ASCII 0) for either string is encountered. The CFG of `match` is given in Fig. 2.1(b).

If we partially evaluate `match` with `p` pointing to the string "hat", we obtain the program whose CFG is shown in Fig. 2.2(a). In this version, the inner loop has been unrolled, and all manipulations and uses of `pat` and `p` have been eliminated: the characters in "hat" are hard-coded into the tests in the specialized procedure. For this example, Eqns. (2.1) and (2.2) become

$$[\![pe]\!](\texttt{match}, [\texttt{p} \mapsto \text{“}\texttt{hat}\text{”}]) = \texttt{match}_{[\texttt{p}\mapsto\text{“}\texttt{hat}\text{”}]} = \texttt{match\_s}$$

[1]Here, the partition of P's inputs into S and D is implicit in $A(S)$.



(a)

```
int match_s(char *s){
  while(*s != 0){
    char *s1 = s;
    if('h' != *s1){
      s++; continue;
    }
    s1++;
    if('a' != *s1){
      s++; continue;
    }
    s1++;
    if('t' != *s1){
      s++; continue;
    }
    s1++;
    return 1;
  }
  return 0;
}
```

(b)

Figure 2.2: (a) The CFG of the residual program (in structured program form). (b) The residual program in the form of a structured program. (This version is given for the sake of pedagogical clarity. As will be seen, the true residual program is in unstructured form.)

and

$$[\![\texttt{match\_s}]\!]_C(A(D)) = [\![\texttt{match}]\!]_C([p \mapsto \text{“hat”}] \cup A(D)),$$

where $[\![\cdot]\!]_C$ denotes the meaning function for C.

2.1.2 Binding-Time Analysis (BTA).

Given a partition of the input variables into "static" and "dynamic" sets, the first stage, called BTA [Jones et al., 1993], extends the partition to one over all occurrences of variables anywhere in the program. The goal is to find a partition of the variable occurrences at the different program points of P into static and dynamic sets (V_s and V_d, respectively) such that, at every statement l in P where a variable $v \in V_s$ is updated, the new value of v is computed solely from variables in V_s (i.e., it does not depend on any dynamic values). Such a partition is said to be *congruent* [Jones et al., 1993]. At specialization-time, partial states are maps from V_s to values; congruence ensures that the variables in V_s can always be evaluated at specialization-time.

To compute a congruent partition, a BTA algorithm can use a *forward slice* [Horwitz et al., 1990, §4.5] from the *dynamic input variables* D. The slice identifies the set V_d of variable occurrences that could be affected by D's values. (One can also think of the BTA algorithm as performing static taint analysis [Livshits and Lam, 2005] with respect to the dynamic input variables.) The static variable occurrences V_s are those in the *complement* of the slice.

The boxed statements in Fig. 2.1(a) show the program points in the forward slice with respect to the dynamic formal parameter s. The slice contains all assignments to, and uses of, variable occurrences that are transitively dependent on s, while the complement of the slice contains all assignments to and uses of variable occurrences *not* dependent on s.

2.1.3 Specialization

The second stage of partial evaluation is *specialization*. The specializer takes as input the program (or CFG), annotated with BTA results, together with the assignment of values for the static input variables. The specializer

creates the residual program by executing the original program over partial states [Jones et al., 1993; Andersen, 1994b] (starting with an initial partial state, such as $[p \mapsto \textit{"hat"}]$). To ensure that the specializer only performs safe updates, it computes only with the variable occurrences V_s of a congruent partition.

One way of structuring the second stage is to *materialize*[2] *the results of BTA as a program* $ge_{P,S}$ that takes as input an assignment $A(S)$, and—when run—emits the code of $P_{A(S)}$:

$$[\![ge_{P,S}]\!](A(S)) = P_{A(S)}, \tag{2.3}$$

where $P_{A(S)}$ is the specialized residual program defined previously, which obeys Eqn. (2.2). A program $ge_{P,S}$ that obeys Eqn. (2.3) is called a *generating extension* for P (and S) [Ershov, 1977].

To automate this approach, one creates what is called a *generating-extension generator*. A generating-extension generator *ge-gen* is a program that takes as inputs (i) a program P, and (ii) a partition $\mathit{Part}(S \cup D)$ of P's input variables into disjoint sets S and D of static and dynamic inputs, respectively, and creates as output a generating extension $ge_{P,S}$

$$[\![\textit{ge-gen}]\!][P, \mathit{Part}(S \cup D)] = ge_{P,S} \tag{2.4}$$

that obeys Eqn. (2.3).

The end-to-end partial evaluator is defined as follows:

$$pe \stackrel{\text{def}}{=} \lambda P.\lambda A(S)\,.\,\textbf{let}\, ge_{P,S} = [\![\textit{ge-gen}]\!](P, \mathit{Part}(S \cup D))\ \textbf{in}\ [\![ge_{P,S}]\!](A(S)). \tag{2.5}$$

(Again, the S/D partition of P's inputs is implicit in $A(S)$.)

Before delving into the concrete implementation details of a generating extension, it will be instructive to informally walk through part of a generating extension's execution, to get a more concrete idea of how classical partial evaluation works.

[2]The use of "materialize" here should not be confused with the use of "materialize" in the term *slice materialization* described in §4.3.

A generating extension executes P on partial states over V_s, where the results of BTA are used to know whether each action of P can be carried out (i) by the generating extension itself (an action classified "static"), or (ii) delayed until the residual program is executed (an action classified "dynamic"). The generating extension performs actions that are "static" (using/updating the current partial state), but emits code to residual program $P_{A(S)}$ for actions that are "dynamic." When emitting dynamic code, the specializer incorporates known static values into the emitted code.

For the purposes of this discussion, assume that a generating extension has some unspecified means of saving and restoring static states over V_s.

This execution is performed in a basic-block-by-basic-block fashion, as an exploration of the state space of V_s. Within a basic block, the execution is a conventional execution of straight-line code, with static code being executed, and dynamic code being emitted as described above. The state-space exploration is a non-standard execution however, because a basic block may end with a control-flow statement governed by a dynamic value, and thus the branch to take cannot be determined at specialization time. Thus, the specialization must be marshalled by means of a worklist of outstanding states to explore both successors of each dynamic control flow statement.

Upon reaching the end of a basic block that is *not* governed by a control-flow statement, or a control-flow statement for which the successor can be determined using the static state control, then there is a single successor block. The generating extension records the current static state σ, as well as a single successor block,[3] and inserts a state/block pair for each success into the worklist.

If, instead, the generating extension reaches a condition classified as "dynamic," such as the two boxed conditionals in blocks 1 and 4 of

[3]Unless the block is the final block of the program, or, e.g., a call to an abort procedure, such as `exit` in Unix.

Fig. 2.1(a), there are *two* successor basic-blocks to interpret. Thus, the partial evaluator instead enqueues *two* partial state/block pairs.

For example, consider what happens when executing the body of the outer loop in Fig. 2.1 for the first time, starting at block 2 (ignoring the dynamic loop head for now). The current state is $\sigma_1 = [p \mapsto \text{"hat"},]$. First, the statement `char s1 = s;` is emitted. Then, the static statement `char *pat = p` is evaluated. Thus, at the end of block 2, the static state is now $\sigma_1 = [p \mapsto \text{"hat"}, pat \mapsto \text{"hat"}]$. Because block 2 does not end with a control-flow statement, there is only one successor, and the pair $(\sigma_1, \texttt{block3})$ is placed in the worklist.

To ensure specialization works correctly, a small piece of bookkeeping must be handled correctly. The residual program's control flow will need to be linked up via gotos. The reasons for doing so will become more apparent upon discussing dynamic control-flow, but for now we will cover the simple case. Every basic block is preceded by a label corresponding to both the identity of the basic block, and the static state on which the block was executed and specialized. For the purposes of this example, we will identify each basic block with the numeric ID of the basic block in Fig. 2.1(b), and for a given state σ_i, the ID will just be i. Thus, before specializing block 2 on σ_1, the generating extension would emit the label "`block_2_state_1:`". Note that due to this, the structure of the residual program will look like Fig. 2.3, rather than Fig. 2.2(b).

Moreover, for reasons that will become apparent upon discussing dynamic control, the generating extension will also emit a goto to the successor of this block. Because the successor is known to be block 3, and the post state is σ_2, the successor's label will be `block_3_state_2`. Thus, after residuating block 2 for the first time, the residual code is:

```
blk_1_1h:
  if(*s != 0)
    goto blk_2_2h;
  else goto blk_8_8h;
blk_2_2h:
  char *s1 = s;
  goto blk_3_3h:
blk_3_3h:
  goto blk_4_4h:
blk_4_4h:
  if('h' != *s1)
    goto blk_6_6h;
  else goto blk_5_5h;
blk_5_5h:
  s1++;
  goto blk_3_3a;
blk_3_3a:
  goto blk_4_4a;

//... omitted ...

blk_5_5t:
  s1++;
  goto blk_3_30;
blk_3_30:
  goto blk_7_70;
blk_7_70:
  return 1;
blk_8_8h:
  return 0;
```

Figure 2.3: The residual string matcher in unstructured form.

```
block_2_state_1:
char s1 = s;
goto block_3_state_2;
```

Now consider what happens if the generating extension de-queues $(\sigma_2, \texttt{block3})$ next. The only code in the block is the if statement `if(*pat == 0)`, which is governed by a static value. Thus, the successor is known to be the false branch, because $\sigma_2 = [\texttt{pat} \mapsto \text{"hat"}]$. The static state is unchanged, so the generating extension enqueues $(\sigma_2, \texttt{block4})$. The code that is emitted is[4]

```
block_3_state_2:
goto block_4_state_2;
```

Now consider the specialization performed when removing $(\sigma_2, \texttt{block4})$ from the worklist. The only statement is a dynamic control-flow statement, and thus, it cannot be determined at specialization-time which successor is taken. However the condition does contain a dereference of a static pointer, and can thus be simplified using static state from `*pat != *s1` to `'h' != *s1`. Because there are two possible successor states: $(\sigma_2, \texttt{block6})$, and $(\sigma_2, \texttt{block5})$, the generating extension emits to the residual program a block that ends with a conditional statement (and, in this case, contains only that conditional statement), as follows:

[4]An attentive reader may notice that this code seems pointless, and that with the information now in hand, the jump at the end of `block_2_state_1` could be amended to target `block_4_state_2`. That technique is called *jump compression* in the partial-evaluation literature. For pedagogical purposes, I ignore this for now. GenXGen's jump-compression method is discussed in Chapter 5.

```
block_4_state_2:
if('h' != s1)
  goto block_6_state_2;
else
  goto block_5_state_2;
```

If $(\sigma_2, \texttt{block5})$ is de-queued next, `pat++;` is evaluated, yielding the post-state $\sigma_3 = [\texttt{p} \mapsto \texttt{"hat"}, \texttt{pat} \mapsto \texttt{"at"}]$. The only successor is the head of the inner loop, block 3, so $(\sigma_3, \texttt{block3})$ is enqueued. The emitted code is:

```
block_5_state_2:
s1++;
goto block_3_state_3;
```

At this point, the generating extension will visit block 3 again, but at σ_3, a new state. The generating extension can again execute the loop body as we've just described, except with $[\texttt{pat} \mapsto \texttt{"at"}]$. It is easy to see that that doing so would yield:

```
block_3_state_3:
goto block_4_state_3;
block_4_state_3:
if('a' != *s1)
  goto block_6_state_3;
else
 goto block_5_state_3;
block_5_state_3:
s1++;
goto block_3_state_4;
```

Note that having done so, the residual code contains multiple versions of blocks from the subject program. This circumstance is what is referred to in the specialization literature as *polyvariance*. Polyvariance is frequently described as "encoding static state into control," or "re-bracketing of the division between code and state." What these phrases describe can be seen explicitly in the code above; there are two variants of block 4, one that has a label associated with σ_2 and one associated with σ_3. All explicit references to `pat` have been removed, but its value in the block's associated static state has been incorporated into the code of the block itself.

Furthermore, note that upon specializing block 5 again, the static increment of `pat` occurs again, yielding a new state $\sigma_4 = [\mathtt{p} \mapsto \text{"}\mathtt{hat}\text{"}, \mathtt{pat} \mapsto \text{"}\mathtt{t}\text{"}]$, and another loop unrolling can occur. After this unrolling, the resulting state is $\sigma_5 = [\mathtt{p} \mapsto \text{"}\mathtt{hat}\text{"}, \mathtt{pat} \mapsto \text{""}]$. At this point the loop is fully unrolled, and a final jump to block 7 is emitted.

What about the dynamic control, however? At block 4, control is returned to block 1, the head of the loop over `s`. This loop cannot be unrolled, even if the inner loop can. What should this traversal look like? This question leads to the first implementation concern for generating

extensions: ensuring termination.

2.1.4 Termination

For a generating extension to terminate, it needs to be able to check state equality. Let's consider this issue for the example from §2.1.3, in which a generating extension is specializing the program from Fig. 2.1. Every time block 1 is traversed, the true branch takes the generating extension back to block 2. Block 2 contains two assignments, and ends with an unconditional branch into the inner loop.

The first time block 2 is executed, `pat` is set to the beginning of string `p`, and an occurrence of block 3 is added to the worklist. As previously seen, the generating extension can unroll the inner loop (i.e., blocks 3, 4, and 5). However, each time block 4 is executed, two successors are enqueued, one corresponding to block 6, and one corresponding to block 5. Note that program specialization can be viewed as a traversal of an implicit graph—the graph of (state, block) pairs induced by the semantics of the static portion of the program—and the traversal is performed as a worklist-marshalled frontier search. Because it is a graph traversal, we are free to choose, e.g., a depth-first traversal strategy, and thus for expository purposes we can assume that the unrolling of the inner loop is performed before the generating extension handles the enqueued entries corresponding to block 6.

Having unrolled the inner loop, one of the worklist entries associated with block 6 will be enqueued. Without loss of generality, consider $(\sigma_4, \texttt{block6})$. Block 6 will be specialized, as will block 1. If specialization next follows the true branch from block 1, block 2 is visited again. The pre-state at block 2 will thus be $\sigma_4 = [\texttt{p} \mapsto \text{"hat"}, \texttt{pat} \mapsto \text{"t"}]$. Executing block 2 sets sets pat to p again, and the post-state is $[\texttt{p} \mapsto \text{"hat"}, \texttt{pat} \mapsto \text{"pat"}]$, which is equal to state σ_2. If block 3 were added to the worklist again, it would lead to an identical unrolling of the inner loop, and specialization

would never terminate.

Instead, the specializer detects repeated (partial-state, basic-block) pairs: the first time `handle_block_2` runs, $(\sigma_2, \text{block } 3)$ is inserted into a record V of visited block/state pairs; each subsequent time `handle_block_2` runs, the specializer determines that $(\sigma, \text{block } 3)$ has repeated because it is already in V.

Thus, it is important that whatever state representation GenXGen uses, it must be (1) inexpensive to save and restore states, and (2) it must be inexpensive to determine when a state/block pair is in V. Chapter 4 describes the implementation of this state-management scheme. §2.3 describes mathematical aspects of our solution to (2).

2.1.5 Generating-Extension Structure

A generating extension can be implemented so that the structure of its code reflects the basic-block structure of the subject program [Andersen, 1994b]. In this approach, a generating extension can be thought of as the subject program, with the partial-evaluation code "compiled in."

The generating extension can be produced algorithmically, basic-block by basic-block: each basic-block in the subject program has an associated basic-block procedure in the generating extension that updates the partial state, generates residual code, and inserts new successor states into the worklist; finally, the basic-block procedure yields control to a central dispatch procedure, which uses the worklist to select the next basic-block procedure to call.

This design was used in the C-Mix partial evaluator for C [Andersen, 1994b]. Figs. 2.4 and 2.5 show such a generating extension for procedure `match` from Fig. 2.1(a), for static input `p`. The central dispatch procedure is `match_ge`. Until worklist `L` is empty, it repeatedly removes a (partial-state, basic-block) pair (σ, b) from `L`, and calls `b` with partial state

```
// All visited (block,state) pairs
state_record_t visited = emptyset;
worklist_t L = empty_worklist();

int match_ge(char *p){
  int cur_block;
  state_t cur_state, init_state;
  //Initialize partial state
  init_state.p = p; init_state.pat = NULL;
  worklist_enqueue(L, 1, init_state);
  insert(visited, 1, init_state);
  printf("match_s(char *s){");
  while(!is_empty(L)){
    cur_block = get_worklist_head(L).block;
    cur_state = get_worklist_head(L).state;
    remove_worklist_head(L);
    switch(cur_block){
      case 1:
      handle_block_1(L, cur_state); break;
      case 2:
      handle_block_2(L, cur_state); break;
      //code elided
      case 8:
      handle_block_8(L, cur_state); break;
    }
    printf("}");
}
```

```
void handle_block_1(worklist_t L, state_t S){
  char *p = S.p; char *pat = S.pat;
  printf("blk_1_%d:", S.id);
  state_t succ_state = snapshot();
  printf("if(*s != 0)");
  printf(" goto blk_2_%d;", succ_state.id);
  printf("else goto blk_8_%d;". succ_state.id);
  if(!contains(visited, 8, succ_state)
    insert(visited, 8, succ_state);
    worklist_enqueue(L, 8, succ_state);
  if(!contains(visited, 2, succ_state)
    insert(visited, 2, succ_state);
    worklist_enqueue(L, 2, succ_state);
}

void handle_block_2(worklist_t L, state_t S){
  char *p = S.p; char *pat = S.pat;
  printf("blk_2_%d:", S.id);
  printf("char *s1 = s;");
  char *pat = p;
  state_t succ_state = snapshot();
  printf("goto blk_3_%d;". succ_state.id);
  if(!contains(visited, 3, succ_state)
    insert(visited, 3, succ_state);
    worklist_enqueue(L, 3, succ_state);
}
```

Figure 2.4: Portions of a C-Mix-style generating extension for `match` from Fig. 2.1, where $S = \{\mathtt{p}, \mathtt{pat}\}$. The remainder of the generating-extension code is presented in Fig. 2.5. Each `handle_block_n` procedure produces a specialized version of a block on a state. Statements in bold produce the code of the residual program. Statements in boxes correspond to program elements in boxes in Fig. 2.1(a) (i.e., elements that depend on the dynamic formal parameter `s`). They are emitted to the residual program along with additional statements that direct the flow of control in the residual program. The `match_ge` procedure uses the worklist of outstanding block/state pairs to marshal the program specialization.

σ. Calling `match_ge` with `p = ''hat''` produces the residual program shown in Fig. 2.2.

```
void handle_block_3(worklist_t L,
                    state_t S){
  char *p = S.p; char *pat = S.pat;
  printf("blk_3_%d:", S.id);
  state_t succ_state = snapshot();
  if(*pat == 0) {
    printf(" goto_7_%s", succ_state.id);
    if(!contains(visited, 7, succ_state))
      insert(visited, 7, succ_state);
      worklist_enqueue(L, 7, succ_state);
  } else {
    printf(" goto_4_%s", succ_state.id);
    if(!contains(visited, 4, succ_state)
      insert(visited, 4, succ_state);
      worklist_enqueue(L, 4, succ_state);
  }
}
void handle_block_5(worklist_t L,
                    state_t S){
  char *p = S.p; char *pat = S.pat;
  printf("blk_5_%d:", S.id);
  pat++;
  printf("s1++;");
  state_t succ_state = snapshot();
  printf("goto blk_3_%s", succ_state.id);
  if(!contains(visited, 3, succ_state))
    insert(visited, 3, succ_state);
    worklist_enqueue(L, 3, succ_state);
}
```

```
void handle_block_4(worklist_t L,
                    state_t S){
  char *p = S.p; char *pat = S.pat
  printf("blk_4_%d:", S.id);
  state_t succ_state = snapshot();
  printf("if(´%c´ != *s1)", *pat);
  printf("  goto blk_6_%s", succ_state.id);
  printf("else goto blk_5_%s", succ_state.id);
  if(!contains(visited, 6, succ_state))
    insert(visited, 6, succ_state);
    worklist_enqueue(L, 6, succ_state);
  if(!contains(visited, 5, succ_state))
    insert(visited, 5, succ_state);
    worklist_enqueue(L, 5, succ_state);
}
```

Figure 2.5: Portions of a C-Mix-style generating extension for `match` from Fig. 2.1, where $S = \{\mathtt{p}, \mathtt{pat}\}$. The remainder of the generating-extension code is found in Fig. 2.4. Each `handle_block_n` procedure produces a specialized version of a block on a state. Statements in bold produce the code of the residual program. Statements in boxes correspond to program elements in boxes in Fig. 2.1(a) (i.e., elements that depend on the dynamic formal parameter `s`). They are emitted to the residual program along with additional statements that direct the flow of control in the residual program.

The construction of the basic-block procedure for block B of the subject program transforms each statement of B in one of three ways:

- Statements classified "static" are placed in the procedure verbatim, and their actions update the partial state accordingly. For example, the increment "`pat++`" in `handle_block_5` causes the value of `pat` in the partial state to be incremented by 1.

- Statements classified "dynamic" are converted into emit statements. Each time they are executed, they generate residual code. For instance, the single occurrence of "`s1++`" in the original `match` program is emitted three times when `match_ge` is called with `p = ‘‘hat’’`: `handle_block_5` is called three times, and thus `printf("s1++;")` is executed three times.

- Some statements classified "dynamic" cannot just be converted to emit statements as is; if a dynamic statement s depends on the value of a static variable v, the value of v must be *lifted* into the residual program's state at s. Lifting can be performed by replacing every occurrence of v in the statement to emit with a parameter for the current value of v. For example, lifting is required for the `if` statement at the end of block 4 in Fig. 2.1. In procedure `handle_block_4`, the `if` statement is emitted via `printf("if('%c' != *s1)", *pat)`, and thus every instance of the statement in the residual program shown in Fig. 2.2(a) has `*pat` replaced with a character from "`hat`".

The worklist manipulations at the end of each basic-block procedure follow three templates, depending on whether the subject-program's block ends with a `goto`, a branch classified "static," or a branch classified "dynamic."

- For a block that ends with a simple `goto`, such as block 5, given the block's single control-flow target b', and the post-state σ', the pair (σ', b') is enqueued if the pair has not previously been traversed. A goto targeting the block corresponding to the specialization of b' with respect to σ' is also emitted

- For a block that ends with a statically-controlled conditional statement, such as block 3, the appropriate successor b' is chosen. This is done by merely evaluating the static branch condition. From this point it continues in a manner identical to the single-target `goto` case.

- In the dynamic case, a generating extension needs "to go both way," as in block 4. Thus, given a post-state σ' and the respective true and false successor blocks b'_t and b'_f, the pairs (σ', b'_t) and (σ', b'_f) are placed in the worklist if they have not previously been enqueued.

  In addition, a conditional `if` statement must be emitted. The condition is simply the original program's condition, possibly with static values lifted into it. The true branch of the if contains a `goto` targeting the residual block associated with (σ', b'_t), and the false branch contains a goto targeting the residual block associated with (σ', b'_t).

State Management. The great potential of generating extensions is to have them *run at native speeds*, manipulating *native memory states*, rather than being interpreted, as in a more classical partial evaluator [Jones et al., 1993]. Unfortunately, there is a stumbling block: a generating extension must support three key state-management operations: *saving partial states*, *restoring partial states*, and *checking state equality*. How can these operations be carried on native hardware states?

In Figs. 2.4 and 2.5, the `snapshot` procedure is used to save states into a `state_t` struct. (We assume that when a state is captured by `snapshot`, an identifying value is assigned to the `state_t` struct's `id` field. The `id` fields of two `state_t` structs are equal if and only if the captured states are equal.) Restoring a state starts with the operation `get_worklist_head(L).state` in `match_ge`, and finishes with the assignments `char *p = S.p` and `char *pat = S.pat` in each `handle_block` procedure. Here we have assumed that a `state_t` struct is a value that can

be assigned to a variable and passed as a parameter. Clearly we are not talking about native hardware states, and hence the architecture illustrated in Figs. 2.4 and 2.5 do not satisfy our desire to have generating extensions run natively.

The generating extensions created by GenXGen[C] are based on the same high-level principles used in C-Mix [Andersen, 1994b]; however, they are based on a *different software architecture* (§4.4, §3.2.1), motivated by the need to support *different state-management mechanisms* (§3) so that GenXGen[C]'s generating extensions can run natively. The core result in Chapter 3 is our solution to the following previously open problem:

> How can a generating extension efficiently (1) save and restore native hardware states, and (2) compare them for equality?

A straightforward approach to implementing these operations requires traversing all of a partial state's reachable memory objects, similar to what is done in mark-and-sweep garbage collection. Such a brute-force approach is inefficient because it essentially requires *multiple invocations of mark-and-sweep when each basic-block of the generating extension executes*. This approach is expensive, hence we want O(1) switches and state comparisons, thus hashing. In Chapter 4, I describe a state-representation and management technique that uses OS primitives to facilitate efficient saving and restoration of hardware states, as well as incrementally-updatable hashing. In §2.3, I provide the mathematical background underpinning the updatable hash.

2.2 Slicing Overview

To implement GenXGen's binding time analysis, I use existing work in program slicing, specifically graph-reachability slicing [Horwitz et al., 1988] and specialization slicing [Aung et al., 2014]. I provide a high-level overview of each, giving enough information to help the reader build

intuition for the properties of the results, while directing curious and motivated readers to the original papers for specific automata-theoretic implementation details.

2.2.1 Graph-Reachability Slicing

This section provides a basic introduction to program slicing, and is a high-level summary of the graph-reachability-based interprocedural-slicing algorithm of Horwitz, Reps, and Binkley [Horwitz et al., 1988].

Given a program point p and a variable x from program P, a forward (data-dependence-)slice of P with respect to x at p is a set S containing all program points in P that depend on x at p (and, as will be shown, perhaps additional program points). It is important to note that S contains *more* than just those that depend on x at p. For example, the set of all program points in P is a (not very informative) slice of P. Because there are a multitude of valid slices, there are many slicing algorithms that provide slices that satisfy additional properties that are useful in various contexts. In §2.2.2, I discuss the limitations of Horowitz-Reps-Binkley slicing, and in §2.2.3 I discuss Aung, Horwitz, Joiner, and Reps's algorithm for specialization slicing [Aung et al., 2014], which rectifies these limitations.

The Horwitz-Reps-Binkley slicing algorithm converts a program slicing problem into a graph-reachability problem by constructing a System Dependence Graph (SDG), which encodes program points as vertices, and dataflow dependencies as edges. An SDG is a representation of a multi-procedure program, consisting of multiple Procedure Dependence Graphs (PDGs), each of which represents a single procedure, and are connected to each other by edges that represent (i) the passing of parameters from caller to callee when the procedure is invoked (including the "passing" of global variables as a kind of extended set of parameters), and (ii) the passing of return values (including globals) from callee to caller when the procedure returns. These value-passing actions are captured in the SDG

as flow-dependence edges from (i) actual-in vertices to formal-in vertices, and (ii) from formal-out vertices to actual-out vertices, respectively. SDGs may optionally contain summary edges between actual-in and actual-out vertices at a call-site, representing transitive flows of values that might occur when the callee is invoked.

For simplicity's sake, we will assume that all C code is available in a normalized form in which at most one variable is assigned to at a given vertex of a PDG (e.g., `a = b++` is converted into two assignments: `a = b` and `b = b + 1`.) By ensuring that at most one variable is assigned to at a given vertex, slicing can be performed with respect to just a program point, and program points and vertices can be discussed interchangeably.

To further simplify the discussion, we first discuss the single-procedure case, and discuss the construction of a single PDG. The vertices in the PDG for a procedure P are essentially the same vertices as in P's CFG: for each program point, input parameter, and output parameter of a procedure P, there is a vertex in the PDG for P.

For each pair of vertices v_1 and v_2, we construct a directed edge (v_1, v_2) if and only if

1. v_1 defines x
2. v_2 uses x
3. there exists a path from v_1 to v_2 in the CFG for P that does not contain an intervening assignment to x

The PDG's for a program's procedures are then connected to form the SDG by connecting the actual-in vertices and actual-out vertices at a call-site to the formal-in vertices and formal-out vertices of the callee. Edges run from actual-in vertices of the caller to corresponding formal-in vertices of the callee, and from formal-out vertices of the callee to corresponding actual-out vertices of the caller.

For example, Fig. 2.7 shows the subset of the SDG for the program in Fig. 1.3 that corresponds to the first call to `q` in `p`. Fig. 2.7(a) shows the portion of the program in Fig. 1.3 corresponding to the the first callsite of `q`, and Fig. 2.7(b) shows the subgraph of the program's SDG that corresponds to the first callsite of `q`. Consider the two formal parameters to `q`, `a` and `b`. In the PDG for `p`, there are two corresponding actual-in vertices, `actual-in a` and `actual-in b`, which are connected, respectively, to `formal-in a` and `formal-in b` in the PDG for `q`. Moreover, because `s` is the value passed as parameter `a` of `q`, there is an edge from `s` to `actual-in a`. The edge from `d` to `actual-in b` exists for the same reason.

Similarly, there are two formal-out vertices in the PDG for `q`, `formal-out g`, which corresponds to the global variable `g` at the return statement, and `formal-out q_ret`, which represents the return value of `q`. These vertices are connected to corresponding actual outs, `actual-out g` and `actual-out q_ret`; `actual-out q_ret` has an edge to `rs = q(s,d)`, encoding the dependence of `rs` on the return value of `q`. Thus, the encoding of transitive dependence as graph reachability is extended to the interprocedural case.

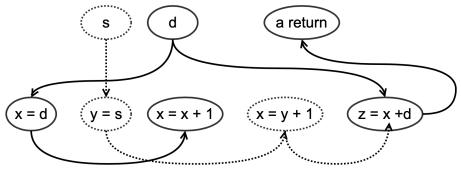
Given the SDG, a forward slice with respect to v is the set of vertices reachable from v in the SDG. In a PDG, the set of vertices in a forward slice can be computed via e.g., a depth-first search of the PDG, starting from the origin point(s) of the slice. For example, the forward slice of procedure `a` with respect to formal-in parameter `d` in Fig. 2.6(a) is the bold component of Fig. 2.6(b).

The intraprocedural case follows the same general reachability approach, except with the addition of *summary edges*. For every procedure `F` in `P`, summary edges that encode the transitive data-flow relationship between `F`'s formal-ins and actual-outs. For leaf procedures, this is straightforward: a edge is added from actual-in v_i to actual-out v_o if and only if there is a path from v_i to v_o. For non-leaf procedures in the presence

```
int a(int s, int d){
  int x, y, z;

  x = d;
  y = s;
  x = x + 1;
  x = y + 1;
  z = x + d;
  return z;
}
```

(a)



(b)

Figure 2.6: (a) The forward slice of procedure a with respect to formal parameter d. (b) The PDG for procedure a. Dotted edges denote a transitive dependence only on s. Solid edges denote a transitive dependence on d. Solid vertices are in the slice.

of recursion, the case is more complex. For our purposes, it suffices to note that the possible calling contexts of a procedure can be encoded as a context-free grammar, and that there exists a polynomial-time algorithm that computes the transitive dependencies between a procedure's input and output parameters by solving a CFL-reachability problem [Horwitz et al., 1988; Reps, 1998].[5]

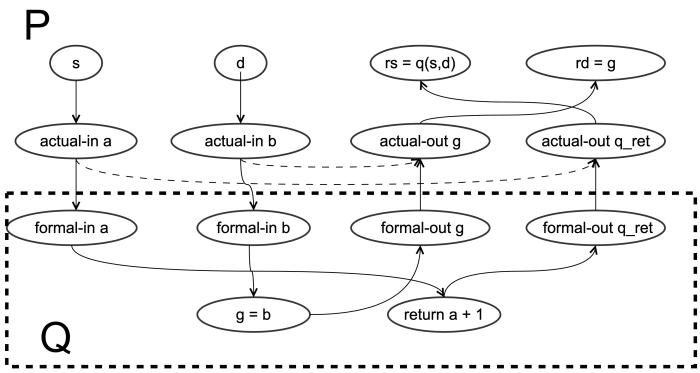
Given summary edges, the slicing is computed in a two-phase approach, with the first phase being an "across-and-out-slice" and the second phase being a "down-but-not-out-slice." More concretely, noting that the reachability computation can be generalized from individual source vertices to a set in the obvious element-wise way:

1. Given a a set of source vertices V, compute forward reachability in the SDG, but **do not follow edges from actual to formal-in vertices**. Importantly, summary edges from actual-ins to actual-outs **are** followed. Store all reachable vertices in set V'.

[5]CFL-reachability is the variant of graph reachability in which a path from node s to node t only counts as a valid s-t connection if the path's labels form a word in a given context-free language

```
int g;
int p(int s, int d) {
    int rs, rd;
    //cut
    rs = q(s, d);
    rd = g;
    //cut
}
int q(int a, int b}){
    g = b;
    return a + 1;
}
```

(a)



(b)

Figure 2.7: (a) Procedures p and q from Fig. 1.3. (b) The subset of the SDG corresponding to the shown portions of p and q. Solid lines denote regular dependence edges, dashed lines denote summary edges.

2. Compute forward reachability from V', but this time edges from actual to formal-in vertices **are** followed. Summary edges are also followed, but edges from formal-out to actual-out vertices are **not** followed. Store the reachable vertices in V''

The resulting V'' contains all vertices corresponding to program points that could be affected by the initial set V.

Unfortunately, as discussed in more detail in §2.2.2, the result of this computation is not sufficient to produce a result that can be used for binding-time-analysis.

2.2.2 Limitations of Graph-Reachability Slicing.

In §2.1.2, I said that given a set of inputs D tagged as dynamic, a forward slice from D could be used as the basis of a BTA algorithm. In this approach, the compliment of the slice of P with respect to D, $S(P, D)$ is the set of static program points. However, not all slices produce valid BTA results; in particular, the Horwitz-Reps-Binkley interprocedural graph-reachability slice can produce *incongruent* results: in certain cases items in

```
int g;
int p(int s, int d) {
    int rs, rd;
    res = 0;
    rs = q(s, d);
    rd = g;
    rd = q(d, s);
    rs += g;
    res += rs;
    res += rd;
    return res;
}
int q(int a, int b){
    return a + 1;
    g = b;
}
```

(a)

```
int g;
int p(int s, int d) {
    int rs, rd;
    res = 0;
    rs = q(s, d);
    rd = g;
    rd = q(d, s);
    rs += g;
    res += rs;
    res += rd;
    return res;
}
int q(int a, int b){
    return a + 1;
    g = b;
}
```

(b)

Figure 2.8: (a) A standard forward slice, which exhibits the parameter-mismatch problem. (b) The results after applying Binkley's algorithm to eliminate the parameter mismatch.

the complement of $S(P, D)$ may have data-flow dependences on items in $S(P, D)$.

Consider again the slice with respect to formal parameter `d` of procedure `p` in the full program from Fig. 1.3 in Chapter 1, shown again in Fig. 2.8(a). In the results, statements (1) `rs = q(s,d);` and (2) `rs += g` are in the complement of the slice. However, both the return value of `q` and the assignment `g = b;` in procedure `q` are elements in the slice. Thus, if all of the elements in the slice were tagged as dynamic, and the specializer attempted to execute the items not in the slice at specialization-time, the right-hand-side of the assignments to `rs` in statements (1) and (2) are values that are not defined at specialization-time.

Examining the subset of the SDG pictured in Fig. 2.9 elucidates the underlying algorithmic issue. In the first phase of the slicing algorithm, the parameter edges into `q` are not traversed. The summary edges are

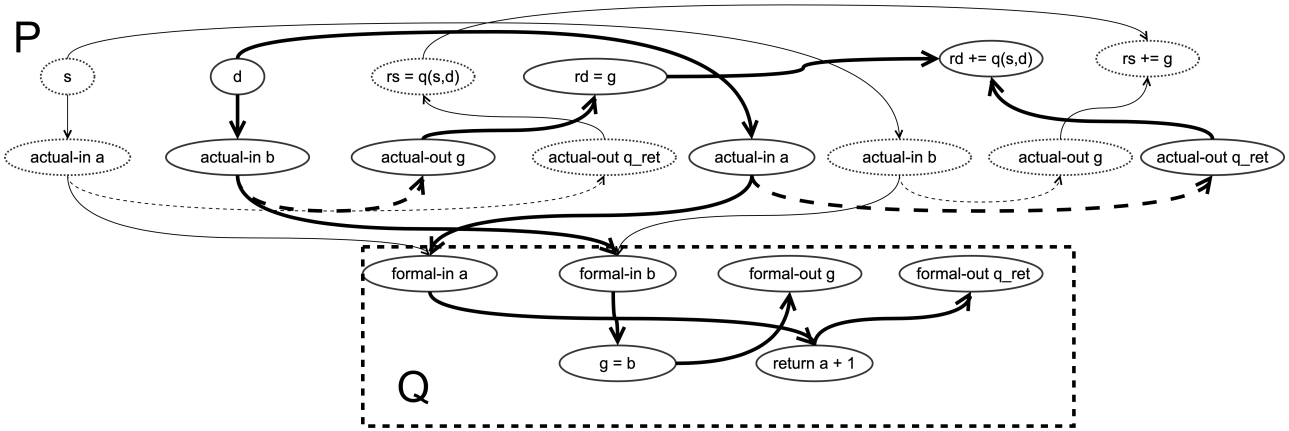


Figure 2.9: The subset of the PDG corresponding to the code in Fig. 2.8(a). Because the slice begins in p, and thus formal-to-actual-out edges can only traversed upwards from p during the execution of the slicing algorithm, the p-to-q output edges are elided. Bold lines denote edges traversed in the computation of the slice, and solid vertices denote vertices in the slice.

traversed at each call-site, however, and they correctly carry the transitive-dependence relationship between the actual-in and actual-out vertices *for that call-site, and only that call-site*, and thus `rs = q(s,d)` and `rs += g` are left out of the slice.

In the second phase, the parameter edges into `q` are traversed, but not the parameter edges out of `q`. Because the formal-in for `a` is in the slice at the first call-site and the formal-in for `b` is in the slice at the second call-site, both statements in the body of `q` are in the slice, as are the two formal-out vertices of `q`.

The key issue is that the summary edges induce a limited and asymmetric form of context-sensitivity. The summary edges cause the actual parameters to `q` to be treated in a context-sensitive manner. However, `q` itself is treated in a context-insensitive manner: the dependences at all call-sites are carried down into `q` in the second phase of the slicing algorithm.

Binkley's algorithm rectifies this issue at the expense of this caller-side context-sensitivity. For every formal-out vertex that is in the slice, Binkley's algorithm examines each associated actual-out vertex. If an actual out is not in the slice, slicing is performed from that vertex, and the results are added to the slice. This slice-augmentation step essentially carries the dynamic dependence up and out of the procedure, and back into the call-site. The slice-augmentation step is repeated until no more parameter mismatches exist. Because the slice-augmentation step cannot remove vertices, this is guaranteed to terminate.

For example, in Fig. 2.9 Binkley's algorithm would perform two more slices, one from the first instance of the vertex labeled "actual-out q_ret" and the second instance of "actual-out g," resulting in Fig. 2.8(b). Binkley's algorithm produces congruent results, but the elimination of context-sensitivity dramatically affects the precision of the slice. In this example, little specializable code remains.

In real-world programs, given the presence of globally-used flags that are dependent on a single procedure parameter, this imprecision can have significant impact on slice precision and the consequent quality of specialization. Consider a two-parameter utility procedure `u(x,y)` that has a path that sets a global flag `f` to a value dependent on `y`, but does not have any path that sets `f` based on `x`. In real-world programs such as BusyBox, small utility procedures of this form often have tens of call-sites. If, in the basic graph-reachability slice, `y` is dynamic at only one of these call-sites, that is nonetheless sufficient to place `u`'s assignment to `f` in the slice, yielding a parameter mismatch at all other call-sites of `u`. Consequently, the application of Binkley's algorithm will re-slice the program's SDG from the actual out-vertices for `f` at *all* other call-sites of `u`. In practice, such slice-augmentation steps cause extremely undesirable results, such as `errno` being classified as dynamic at virtually every point in the program.

The issue discussed above leads to the key question that specialization slicing [Aung et al., 2014] addresses: how does one extend context-sensitivity seen at the call-sites in the basic (non-Binkley) version of graph-reachability slicing *to the callee-procedure representations as well*?[6]

2.2.3 Specialization Slicing

The crux of the problem with basic graph-reachability slicing is the monovariance of the program procedure representation. Because there is one version of of every procedure `P`, the information at every call-site of `P` must be incorporated into the single representation of `P`, and thus, due to the presence of summary edges, the information about `P` itself may be less precise than the information available at any given call-site.

[6] In fact, specialization slicing also improves context sensitivity at the call-sites in certain situations, particularly in the presence of recursion.

```
int g;
int p(int s, int d) {
    int rs, rd;
    res = 0;
    rs = q1(s, d);
    rd = g;
    rd = q2(d, s);
    rs = g;
    res += rs;
    res += rd;
    return res;
}

int q1(int a, int b){
    return a + 1;
    g = b;
}

int q2(int a, int b){
    return a + 1;
    g = b;
}
```

Figure 2.10: The graph-reachability slice of the polyvariant version of the code from Fig. 2.8. The procedure q has been replaced by a new copy for each call-site.

```
int s, d;

void swap(a, b){
  s = b;
  d = a;
}

void rec(k){
  swap(s,d) //cs1
  if(k > 0)
      rec(k-1); //cs2
  swap(s,d); //cs3
}

int main(){
    s = 2;
    d = get_int();//dynamic
    int k = get_int();
    rec(k); //cs0
    return s;
}
```

Figure 2.11: A recursive procedure whose binding-time pattern depends on calling context.

Thus, converting a program to a polyvariant representation will improve the quality of slicing. Given the code in Fig. 2.8, making a copy of q for each call-site improves precision, as pictured in Fig. 2.10. In the absence of recursion, this approach would work. However, it may be the case that an excessive amount of copying would be performed. For a two-parameter procedure like q, at most four copies need to be made, one for each possible subset of the input parameters that are in the slice. If there are more than four call-sites of q, some copies of q are guaranteed to be redundant.

Moreover, as we will see, naive inlining fails in the presence of recursion. Consider the program given in Fig. 2.11, which is an example from [Aung

et al., 2014]. I will use this program as a running example to motivate the core ideas from the work of Aung et al. Before even thinking about how to perform a slice, it is useful to consider what answer would be a desirable result for the slice of this code with respect to the first assignment to `d`.

The procedure `swap` takes the value of the first parameter, `a`, and assigns it to global variable `d`, and similarly `swap` assigns the value of `b` to global variable `s`. Note that `swap` is always called with `s` as the first argument, and `d` as the second. Thus, for the purposes of reasoning about the behavior of the program, it suffices to treat `swap` as a procedure that swaps `s` and `d`.[7]

Now consider the behavior of the procedure `rec` with respect to `s` and `d`. With a small bit of insight, one can see that an inductive argument establishes that for all `k`, `rec(k)` leaves `s` and `d` unchanged.

If `k` is zero, `swap` is called twice in succession, with no recursive call to `rec`. So, after a call to `rec(0)`, `s` and `d` have the same values as before the call. More precisely, `s` immediately after a call to `rec(0)` depends only `s`, and similarly `d` after `rec(0)` depends only on `d`.

Now consider the case where `k` is greater than zero. The first call to swap `k` leaves `s` and `d` swapped. Invoking the inductive hypothesis that `rec(·)` leaves `s` and `d` unchanged, the recursive call `rec(k-1)` leaves `s` and `d` in the swapped configuration, and the second call to `swap` returns them to the original configuration. Again, we can make a stronger statement: the value of `s` after `swap(k)` depends solely on `s` and, symmetrically, `d` after `swap(k)` depends solely on `d`.

Thus, whatever other properties a slice with respect to the initial assignment to `d` in `main` might have, it would be desirable for `return s` to *not* be in the slice. However, a basic graph-reachability slice must include both assignments to `s` and `d`. Moreover, naively making a copy of each pro-

[7] Certainly, we could write `swap` as a procedure with no formal parameters that just explicitly swapped the global variables. However, that version would make it impossible to point out the changing dependence patterns of the input parameters that occur in our running example.

```
void rec_0(k){
  swap(s,d) //cs1
   if(k > 0)
      rec_1(k-1); //cs2
  swap(s,d); //cs3
}
```

```
void rec_1(k){
  swap(s,d) //cs1
  if(k > 0)
      rec_2(k-1); //cs2
  swap(s,d); //cs3
}
```

```
void rec_2(k){
  swap(s,d) //cs1
  if(k > 0)
      rec_3(k-1); //cs2
  swap(s,d); //cs3
}
```

Figure 2.12: Copies of `rec` in the infinite inlining of `rec`

cedure for each call-site in the program clearly cannot be used to resolve the problem, because `rec` is recursive.

Surprisingly, considering the slice of an infinite inlining of copies of `rec` provides a useful insight. Let `rec_i` represent the version of `rec` at call-depth `i`. That is, `main` calls `rec_0`, `rec_0` calls `rec_1`, and so on (see Fig. 2.12). Moreover, let us associate each copy of `rec` with a *stack context*. Let a stack context be defined in terms of a *call-string*:

Let each c_i represent a call-site, and let the string $c_1c_2c_3...c_n$ represent a call-string where c_n is the bottom of the stack. Then, `rec_0` is associated with c_{cs0}, and e.g., `rec_2` is associated with $c_{cs2}c_{cs2}c_{cs0}$. In general, `rec_n` is associated with $(c_{cs2})^n c_{cs0}$. Moreover, we can represent the calling contexts of `swap` in the same manner with, e.g., the second call to `swap` in `rec_1` as $c_{cs3}c_{cs2}c_{cs0}$.

Let us consider what the slice with respect to the assignment `d` in `main` should look like at different depths. First, consider the behavior of `rec_0`. Which of the occurrences of `s` and `d` in `rec_0` are in the slice at call-sites *cs1*, *cs2*, and *cs3* with calling context c_{cs0}?

From our argument earlier, we know that `rec_0`'s call to `rec_1` will leave `s` and `d` unchanged. From this observation, it follows that the second actual parameter of `swap` at *cs1* and the first actual parameter of `swap` at *cs3* should be in the slice.

Next, consider the slice of `rec_1`. Before `rec_0` called `rec_1`, `s` and `d` were swapped. Thus, the *first* actual parameter of `swap` at *cs1* and the *second* actual parameter of `swap` at *cs3* are in the slice.

```
int s, d;

void swap_l(a, b){
  s = b;
  d = a;
}

void swap_r(a, b){
  s = b;
  d = a;
}
```

```
void rec_even(k){
  swap_r(s,d)
  if(k > 0)
      rec_odd(k-1);
  swap_l(s,d);
}

void rec_odd(k){
  swap_l(s,d)
  if(k > 0)
      rec_even(k-1);
  swap_l(s,d);
}
```

```
int main(){
    s = 2;
    d = get_int();
    int k = get_int();
    rec_even(k);
    return s;
}
```

Figure 2.13: A polyvariant version of Fig. 2.11 that yields the desired result.

Clearly, then, because `swap` is called before `rec_2` in `rec_1`, `d` should be in the slice upon entry to `rec_2`. *But, this situation is identical to the situation for `rec_0`.*

Indeed, it is clear that for $n = 2k$, `rec_n` should be sliced identically to `rec_0`, and similarly for $n = 2(k + 1)$, the slice of `rec_n` is identical to `rec_1`. Thus, if we make two versions of `rec`, one encoding the odd-labeled elements of the infinite inlining of `rec`, and the other encoding the even-labeled elements of the infinite inlining, as well as two versions of `swap`, one encoding the case where the first parameter-dependency is in the slice, and one where the second is, and inline them appropriately, we obtain Fig. 2.13. Computing the forward dependence slice over this version of the program yields the desired result, as shown in Fig. 2.13.

Stated in terms of calling contexts, the slice of any member of the infinite unrolling at $(c_{cs2}c_{cs2})^*c_{cs0}$ looks like `rec_even` and any one at $(c_{cs2}c_{cs2})^*c_{cs2}c_{cs0}$ looks like `rec_odd`. Similarly, any version of `swap` at $c_{cs3}(c_{cs2}c_{cs2})^*c_{cs0}$ or $c_{cs1}(c_{cs2}c_{cs2})^*c_{cs2}c_{cs0}$ looks like `swap_l`, and conversely any version at $c_{cs1}(c_{cs2}c_{cs2})^*c_{cs0}$ or $c_{cs3}(c_{cs2}c_{cs2})^*c_{cs2}c_{cs0}$ looks like `swap_r`.

Note that these are all regular languages: this property is what lies behind the key insight of the specialization-slicing algorithm. Whenever, in an infinitely-inlined version of a program, one "performs" a data-

dependence slice like the one depicted in Fig. 2.12, the versions of a procedure P in the infinite inlining can be partitioned into a finite number of equivalence classes, where two elements P_a and P_b are in the same class if they have the same vertices in their slice result. Moreover Aung et al., show that each equivalence class can always be associated with a regular language of calling contexts.[8]

Specialization slicing is performed via a series of automata-theoretic operations [Aung et al., 2014]. First, a program P is transformed into a pushdown system that reflects the calling structure of the program. That is, the stack-manipulations of the system correspond to calls and returns in P. Next, a finite automaton A encoding a regular language of slice sources is constructed. Then, a series of automata-theoretic operations are performed on A, using the structure of P, which construct the sets of regular languages encoding the most-precise partition of each procedure's infinite inlining as described above.

Thus, specialization slicing gives a viable means of performing a more precise binding-time analysis than that afforded by Binkley's Algorithm. The only potential complication is the risk of expnential explosion. It is possible to construct a program with k procedures that has 2^k procedures in the specialization-slicing result. In Chapter 6, we find that such an explosion does not happen in practice. In Chapter 3, I discuss the implementation details of integrating specialization slicing with GenXGen.

2.3 Rabin fingerprinting

The worklist algorithm in §2.1.3 must, when enqueueing a state/block pair, avoid enqueueing the state if the block has already been visited in that

[8]The first part is trivially true: there are only finitely many possible slice results for each procedure. The surprising part is that (1) the partition can always be characterized in terms of regular languages, and (2) a representation of the set of regular languages for the partition is computable.

state. A naive approach to state-comparison entails comparing the current state against all previously seen states, which in the worst case leads to specialization requiring $O(N^2)$ time. If possible, we would like to have this check take constant time, which suggests a hashing-based approach.

However, because GenXGen produces generating extensions for low-level languages, program states consist solely of hardware states: the contents of the CPU registers and memory. Traditional hashing mechanisms have some means of resolving false positives; on a hash collision, the two objects must be compared. Here, that potentially entails comparing all live memory in at least two program states. This issue has significant performance implications for a generating extension, and thus I instead choose to take a probabilistic approach, and forego a collision-resolution mechanism. Thus, it is essential to use a hashing method that guarantees a negligible probability of there being any pair-wise collision between hashed states, over the full execution of a generating extension.

In addition, because states are hardware states, it is infeasible to adopt the naive strategy of hashing all of memory after executing each block (even for a 32-bit architecture). However, in Chapter 4, I show how it is possible to identify specific regions of memory that have changed during an execution of a basic block, and how to make the contents of these regions before and after the basic-block's execution available to the generating-extension runtime.[9] Thus, the most pressing question is, (*) "how do we use the available information from the OS to efficiently update a hash, while guaranteeing that there is a negligible probability of there being any pairwise collision between hashed states."

Avoiding specific implementation details for now, we will operate under the following abstract model: the generating-extension runtime partitions memory into *w*-bit regions, and can (a) identify all regions that

[9]Concretely, Chapter 4 shows how to use `fork` in conjunction with OS-level support for logging events from the Copy-on-Write mechanism of the Linux page-fault handler to collect data about memory changes at 4096-byte page granularity.

had at least one write, and (b) provide the values of changed regions both before and after the writes. For the purposes of this abstract discussion, to decouple the idea from any concrete hardware or OS-level details, I will refer to these regions as "chunks."

Given this model, we consider the following refined version of (*): (**) "given information about what chunks have changed during basic-block execution and their contents before and after, is it possible to obtain an efficient means of computing state summaries? Moreover, can the strategy be implemented in a way that guarantees that there is a negligible probability of there being any pair-wise collision between hashed states?" Question (**) suggests an in-place updatable hash.

What is desired is a hashing algorithm H that, given a state hash S and the following information about changes to a chunk c:

1. $p(c)$, the bit-level offset of c in memory
2. $b(c)$, the contents of c before executing the block.
3. $a(c)$, the contents of c after executing the block.

We would like to be able to compute the post-state hash using only the given information: $S' = H(S, p(c), b(c), a(c))$

Additionally, because we have no mechanism for collision resolution, it is important to ensure that the hash algorithm H guarantees that there is a negligible probability of there being any pair-wise collision between hashed states acceptably low. Specifically, when given a collection M of items, and a hash length of k, we would like the probability of any intra-set collision to be proportional to $\frac{|M|}{2^k}$.

Rabin's "fingerprinting method" [Rabin, 1981; Broder, 1993] provides just such a hashing scheme. The overview here is based on both Rabin's and Broder's papers, and in particular, the mathematical overview and collision-resistance argument are based on Broder's discussion of the material.

2.3.1 Mathematical Preliminaries

Rabin's fingerprinting method uses the mathematical properties of polynomials over the Galois field of order 2 ($GF(2)$) to construct a hashing scheme with the desired collision properties. In particular, Rabin fingerprinting represents bit-strings as polynomials over $GF(2)$, and performs basic arithmetic operations over the polynomials to compute the hash.

$GF(2)$ is the set $\{0, 1\}$ augmented with addition and multiplication defined as follows:

$+$	0	1
0	0	1
1	1	0

$\times$	0	1
0	0	0
1	0	1

Stated in terms of bit-level operations, $+$ is exclusive-or, and $\times$ is logical-and. Moreover, every element is its own additive inverse, so the subtraction operation in $GF(2)$ is the same as the addition operation. Thus, arithmetic over $GF(2)$ is easily implemented in hardware.

Next, we define $GF(2)[t]$, the field of polynomials over $GF(2)$. A polynomial $P(t)$ of degree m has the form

$$P(t) = s_0 + s_1 t^1 + ... + s_m t^m, \tag{2.6}$$

where each s_i is an element of $GF(2)$, and $t^m \neq 0$. Moreover, given a bitstring σ, where $|\sigma| = n$ we can define the polynomial $\sigma(t)$ as the polynomial of degree $m < n$ where m is the largest index of a non-zero bit, and s_i is the i^{th} bit of σ.

It is important to note that we are considering the polynomials merely as *formal polynomials*. That is, we have no interest in evaluating them at any particular t; we are merely interested in the algebraic properties of the polynomials themselves.

Addition and subtraction are extended to polynomials in the usual way. Two particular properties relevant to the hardware-level implementation

of Rabin fingerprinting are:

- Polynomial addition $(+)$ is bit-wise xor.
- Multiplication by t^i can be implemented as an i-bit shift.

These operations are conveniently implemented in hardware, and there is no space overhead for considering a bitstring as $\sigma(t)$: the bitstring itself is a perfectly good representation of $\sigma(t)$.

In addition, with respect to multiplication and division, $GF(2)[t]$is "well-behaved" in the sense that it has algebraic properties analogous to those of the integers. The division $P(t)/D(t)$ is well-defined for all $P(t)$ and non-zero $D(t)$, and hence remainders and congruence modulo $D(t)$ are as well. For example, $(1 + t^1 + t^2)/t^1$ yields a remainder of 1:

$$\begin{array}{r} t^1 + 1 \\ t^1 + 0 \overline{)\ t^2 + t^1 + 1} \\ \underline{+t^2 + \ 0} \\ t^1 + 1 \\ \underline{+\ t^1 + 0} \\ 1 \end{array}$$

Thus, $t^2 + t^1 + 1$ is congruent to 1 mod $(t^1 + 0)$.

For every $m > 0$, there exist *irreducible polynomials*[10] of degree m, which are analogous to prime numbers: they are not divisible by any polynomial other than themselves and the unit polynomial.[11]

[10]Note that while congruence mod $D(t)$ plays the central role in the construction and analysis of Rabin's fingerprinting algorithm, we are never concerned with arithmetic in the ring of polynomials mod $D(t)$. All computations are done in $GF(2)[t]$, and when I speak of irreducibility, I mean irreducibility in the field $GF(2)[t]$.

[11]That is, the polynomial of degree zero, where $s_0 = 1$ (in the notation of Eqn. (2.6)).

Example 2.1. *The only irreducible polynomial of degree* 2 *is* $1 + t^1 + t^2$ *and the two irreducible polynomials of degree* 1 *are* t^1 *and* $1 + t^1$.

Moreover, just as every integer has a unique prime factorization, every polynomial in GF(2)[t]has a unique factorization into irreducible polynomials.

Example 2.2. *The unique irreducible factorization of* $t^2 + t^1$ *is* $(1 + t^1)(t^1)$. *The unique irreducible factorization of* $(1 + t^2)$ *is* $(1 + t^1)(1 + t^1)$.

For the purposes of this dissertation, it will suffice to observe that for reasoning about the operations relevant to Rabin fingerprinting, the algebraic properties of remainders for polynomials over $GF(2)$ are identical to those for the integers.

As we will see, Rabin fingerprinting consists of (1) randomly selecting an irreducible polynomial $P(t)$ of degree n, and (2) for each bitstring σ to hash, computing $H(\sigma) = \sigma(t) \bmod P(t)$. The algebraic properties of mod for polynomials over $GF(2)$ admit an efficient in-place update of H.

In addition, it will be shown that collision-resistance derives from the fact that (1) $P(t)$ was chosen at random, and (2) there are more than $(2^k - 2^{k/2})/k$ irreducible polynomials over $GF(2)$ of degree k.

2.3.2 Rabin's In-Place Fingerprinting Method

Given a bit-string $\sigma = (s_0, s_1, \ldots, s_{m-1})$, and an irreducible polynomial $P(t)$ of degree k, the fingerprint $H(\sigma)$ is defined as

$$H(\sigma) \stackrel{\text{def}}{=} (\sigma(t) \bmod P(t)) \tag{2.7}$$

The choice of the degree k of the irreducible polynomial $P(t)$ gives us control of the size of hash values: $H(\sigma)$ is represented by a bit-string of

length k. The choice of k also allows us to control the probability of any intra-set collision. [Broder, 1993; Rabin, 1981].

For the purposes of the running examples in this section, we will consider a one byte-memory divided into two four bit chunks:

Example 2.3. *Consider an* 8*-bit bitstring* $\sigma = 1010\ 0110$*—corresponding to the contents of a one-byte memory divided into two four-bit chunks—and a* 3*-bit irreducible polynomial* $P(t) = 1 + t + t^2$*. The corresponding polynomial for* σ *is* $\sigma(t) = 1 + t^2 + t^5 + t^6$*. Reducing mod* $P(t)$*, we obtain the hash value* $H(\sigma) = 000$*.*

By properties of mod, Eqn. (2.7) implies two useful properties:[12]

1. Fingerprinting is linear: $H(A + B) = H(A) + H(B)$.
2. The fingerprint of the product of t^i and a polynomial $\sigma(t)$ can be computed via $H(t^i * \sigma(t)) = H(H(t^i) * H(\sigma(t)))$.

Consider these properties in the case of our one-byte memory. The memory consists of two chunks:

$$\sigma = (s_0t^0 + s_1t^1 + s_2t^2 + s_3t^3) + (s_4t^4 + s_5t^5 + s_6t^6 + s_7t^7)$$

Note that for the second chunk, we can factor out t^4, yielding

$$\sigma(t) = (s_0t^0 + s_1t^1 + s_2t^2 + s_3t^3) + t^4(s_4t^0 + s_5t^1 + s_6t^2 + s_7t^8)$$

By property (1),

$$H(\sigma(t)) = H(s_0t^0 + s_1t^1 + s_2t^2 + s_3t^3) + H(t^4(s_4t^0 + s_5t^1 + s_6t^2 + s_7t^8))$$

[12] A convenient way of thinking about these two properties is that H is an additive homomorphism and is "nearly" a multiplicative homomorphism in the specific case shown in property 2. Were it not for the outer application of H in the right-hand side of property 2, it would be one.

By property (2), and having $c_0 = s_0t^0 + s_1t^1 + s_2t^2 + s_3t^3$, then

$$H(\sigma(t)) = H(c_0) + H(H(t^4)H(s_4t^0 + s_5t^1 + s_6t^2 + s_7t^8))$$

From this expression, it is easy to see that if the second chunk is changed, one needs only to update the hash of the affected chunk. Given a change to the second chunk in pre-state σ, the post-state σ' can be derived by subtracting off—i.e., adding—the terms representing the contents of the second chunk in σ, and adding in the terms representing the contents of that chunk in σ'.

More explicitly, representing the contents of the second chunk of σ with t^4 factored out as c_1 and the contents of the second chunk of σ' with t^4 factored out as c_2', we have

$$\sigma = c_0 + t^4c_1$$
$$\sigma' = c_0 + t^4c_1'$$

and thus,

$$\sigma' = c_0 + 0 + t^4c_1'$$
$$\sigma' = c_0 + (t^4c_1 + t^4c_1) + t^4c_1'$$
$$\sigma' = \sigma + t^4c_1 + t^4c_1'$$

and finally,

$$\begin{aligned} H(\sigma') &= H(\sigma + t^4c_1 + t^4c_1') \\ &= H(\sigma) + H(t^4c_1 + t^4c_1') \\ &= H(\sigma) + H(t^4(c_1 + c_1')) \\ &= H(\sigma) + H(H(t^4)H(c_1 + c_1')) \end{aligned}$$

Example 2.4. *Returning to Ex. 2.3, assume that some writes occur in chunk* 1 (*the second chunk*), *yielding the post-state* $\sigma' = 1010\ 0011$. *Computing directly, the post-state polynomial is* $\sigma'(t) = 1 + t^2 + t^6 + t^7$, *and by reducing* $\sigma'(t)$ *mod* $P(t)$, *we obtain the post-state hash* $H(\sigma') = 1 + 0 * t + 0 * t^2 = 100$.
Alternatively, $H(\sigma')$ *can be computed incrementally from* $H(\sigma) = 000$. *For page* 1, *the pre-state polynomial is* $t + t^2$, *and the post-state polynomial is* $t^2 + t^3$. *We have*

$$\begin{aligned} H(\sigma') &= H(\sigma) + H((t^5 + t^6) + (t^6 + t^7)) \\ &= H(\sigma) + H(t^4(t + t^2) + t^4(t^2 + t^3)) \\ &= H(\sigma) + H(t^4(t + t^2 + t^2 + t^3)) \\ &= H(\sigma) + H(H(t^4) * H(t + t^3)) \\ &= H(\sigma) + H(t * (1 + t)) \\ &= H(\sigma) + H(t + t^2) \\ &= H(\sigma) + 1 + 0 * t + 0 * t^2 \\ &= 100 \end{aligned}$$

It is easy to see that this generalizes to any memory and chunk size. By properties (1) and (2), $H(\sigma')$ can be directly computed from $H(\sigma)$ using only the contents of the i^{th} chunks in σ and σ', thereby avoiding the need to examine all of σ' to compute its hash value $H(\sigma')$.

Let w be the chunk size. In addition, let $\sigma_{a,b} = s_a + s_{a+1}t + ... + s_b * t^{b-a}$ denote the bit-string containing the bits of the substring of σ starting at a and ending at b, inclusively, for both a and b. From properties (1) and (2), we have that $H(\sigma')$ equals

$$\begin{aligned} &H(\sigma) + H(t^{i*w} * \sigma_{i*w,(i+1)*w-1}) + H(t^{i*w} * \sigma'_{i*w,(i+1)*w-1}) \\ &= H(\sigma) + H(t^{i*w} * (\sigma_{i*w,(i+1)*w-1} + \sigma'_{i*w,(i+1)*w-1})) \\ &= H(\sigma) + H(H(t^{i*w}) * H(\sigma_{i*w,(i+1)*w-1} + \sigma'_{i*w,(i+1)*w-1})). \end{aligned}$$

For a fixed chunk size of, e.g., 2^{15} bits,[13] the only non-constant-time computation is $H(t^{i*w}) = (t^{i*w} \bmod P(t))$, which can be computed in time $\log_2(i * w)$ using modular-exponentiation-via-squaring. That is,

$$H(t^{2x}) = H(t^{x+x}) = H(t^x t^x) = H(H(t^x)^2).$$

For example,

$$H(t^{16}) = H(H(t^8)^2) = H(H(H(t^4)^2)^2) = H(H(H(H(t^2)^2)^2)^2).$$

Then, letting $A(P) = (H(P))^2$, with $A^n(P)$ denoting n nested applications of A to P in the usual sense, it is easily proved that for any $x = 2^y$

$$H(t^x) = H(A^{\log_2(x)}(t))$$

Because the maximum amount of addressable memory is bounded, $\log_2(i * w)$ is effectively a small constant. Moreover, it is inexpensive to precompute and cache hashes of t^x where $x = 2^y$ for all y up to 64.

In general, the number of chunks that must be hashed to compute the post-state hash is $O(q)$, where q is the number of unique chunks written during the execution of the basic block, which is at most $O(r)$, where r is the number of instructions or statements in the compiled basic block.

2.3.3 Collision Resistance.

We now discuss the collision-resistance property of Rabin fingerprinting. The derivation is based on Broder's treatment of the material [Broder, 1993].

Recall that to ensure that determining whether a state has been seen before takes $O(1)$ time, GenXGen does not perform collision resolution. The following derivation shows that the probability of any intra-set collision can be made arbitrarily small by adjusting the degree k of $P(t)$. Because elements of GF(2)[t]are represented as bit-strings, adjusting k is the same

[13] The size of a page in bits on x86 Linux is 2^{15}.

as adjusting the size of the hash. It can be shown that given a set V of items to fingerprint, the probability of any intra-set collision is proportional to $|V|/2^k$. Thus, given an upper bound on the number of unique states visited in an execution of a generating extension—for our purposes, sixteen million—the probability of any intra-set collision is proportional to 2^{-k}.

Recall that the irreducible degree-k polynomial $P(t)$ was chosen uniformly at random. To reason about collision resistance, we need to compute the probability of at least one collision[14] given a set of values V to fingerprint, where $n = |V|$. More concretely, we are trying to answer the question, "given a fixed V, and a $P(t)$ chosen uniformly at random out of all irreducible polynomials of degree k, what is the probability that there exist at least two items $a, b \in V$ that are both congruent to the same value mod $P(t)$?"

The derivation of the probability bound that answers this question exploits the irreducibility of $P(t)$, along with a key algebraic property of $GF(2)[t]$, namely the fact that every polynomial has a unique irreducible factorization. Every polynomial $A(t)$ has a set S_A of of irreducible factors, and the algebraic structure of $GF(2)[t]$is sufficiently "well-behaved" that the following proposition holds, which is analogous to the same property for divisibility by primes in the integers: a polynomial $A(t)$ is congruent to 0 mod an irreducible polynomial $P(t)$ if and only if $P(t) \in S_A$.

A collision occurs when two polynomials $A(t)$ and $B(t)$ are congruent to the same polynomial $C(t)$ mod $P(t)$. Thus, if there is a collision,

$$H(A(t)) = H(B(t)) = C(t)$$

and

$$H(A(t) - B(t)) = H(A(t)) - H(B(t)) = 0$$

Now, consider Q, where Q is the product of all $(A(t) - B(t))$, where

[14]That is, we're looking to answer the "birthday paradox" problem for a set of bit-strings being hashed.

$(A(t), B(t))$ are unordered pairs over $A, B \in V$, where $A(t) \neq B(t)$, i.e.,

$$Q \stackrel{\text{def}}{=} \prod_{\substack{A(t), B(t) \in V, \\ A(t) \neq B(t)}} (A(t) - B(t)).$$

Then, there is a collision if and only if Q is congruent to 0 mod $P(t)$.

Example 2.5. *Fix* $P(t) = 1 + t^1 + t^2$*, and let the set* V *consist of the following polynomials:*

$$A_1(t) = 1 + t^2,\ A_2(t) = 1 + t^1,\ A_3(t) = t^1 + t^2$$

Their hashes are the remainders mod $P(t)$*:*

$$H(A_1(t)) = t^1,\ H(A_2(t)) = 1 + t^1,\ H(A_3(t)) = 1$$

Thus, there is no collision. Now, consider Q:

$$Q = (A_1(t) - A_2(t))(A_1(t) - A_3(t))(A_2(t) - A_3(t))$$
$$Q = (t^1 + t^2)(1 + t^1)(1 + t^2)$$

The irreducible factorization of each term is:

$$(A_1(t) - A_2(t)) = (t^1)(1 + t^1)$$
$$(A_1(t) - A_2(t)) = (1 + t^1)$$
$$(A_1(t) - A_2(t)) = (1 + t^1)(1 + t^1)$$

$P(t)$ *is not a factor of any term in the product* Q*, and thus* $P(t)$ *does not divide* Q*. Hence* Q *is not congruent to* 0 *mod* $P(t)$*.*

Example 2.6. *Now consider* V' *where* A_2 *is replaced with* $A_2'(t) = t^1$*. We have:*

$$H(A_2'(t)) = H(t^1) = t^1$$

Thus, $H(A_1(t)) = H(A_2'(t))$*, and a collision exists.*
Now consider the term of Q' *corresponding to* $A_1(t) - A_2'(t)$*. We have:*

$$A_1(t) - A_2'(t) = ((1 + t^2) + t^1) = 1 + t^1 + t^2$$

The difference is equal to $P(t)$*, and hence congruent to* 0 *mod* $P(t)$*. Thus,* $P(t)$ *is a factor of* Q'*, and so* Q' *is divisible by* $P(t)$*.*

Thus, by selecting a set of values V to hash, one fixes the product polynomial Q. Let $F_k(Q)$ be the number of irreducible factors of Q of degree k, and let I_k be the number of irreducible polynomials of degree k. Recalling that $P(t)$ was selected uniformly at random, I claim that we have the following upper bound on the probability C of any pairwise collision among the elements of V:

$$C \leqslant \frac{F_k(Q)}{I_k}.$$

To see why it is an inequality, note the following:

1. There is a collision if and only if $P(t)$ divides Q, and hence if and only if $P(t)$ is in Q's unique irreducible factorization.

2. the unique irreducible factorization of a polynomial may contain multiple instances of the same factor.

That is, because the set of *distinct* irreducible factors of Q may be smaller than the number of factors in Q's irreducible factorization, the ratio $F_k(Q)/I_k$ overestimates the probability of any intra-set collision.

Stated dfferently, because $P(t)$ is chosen uniformly at random, the probability of any given factor A of degree k of Q being $P(t)$ is

$$\frac{1}{I_k}$$

Then, if S is the multiset containing the (possibly duplicated) irreducible factors of Q, we have $|S| = F_k(Q)$, and thus obtain the union bound

$$C \leqslant \sum_S \frac{1}{I_k} = \frac{F_k(Q)}{I_k}.$$

Q has at most $\deg(Q)/k$ irreducible factors of degree k, so

$$C \leqslant \frac{F_k(Q)}{I_k} \leqslant \frac{\deg(Q)/k}{I_k}.$$

For every pairwise difference $A(t) - B(t)$ in Q, the degree is bounded from above by the largest degree in the pair $(A(t), B(t))$. Thus, if m is the degree of the largest polynomial in V, the degree of every difference is at most m. In addition, for every $A(t) \in V$, there are at most n non-zero differences involving $A(t)$.

Note that degree of the product of any set of polynomials is the sum of their degrees. Thus, the degree of the product of all differences involving A is at most nm. Then,

$$\deg(Q) \leqslant \sum_{A(t) \in V} nm = n^2 m.$$

Although not proved here, it can be shown that there are at least $(2^k - 2^{k/2})/k$ irreducible polynomials of degree k. Consequently, a bound on

the probability of any intra-set collision can be derived as follows:

$$\begin{aligned} C &\leqslant \frac{F_k(Q)}{I_k} \\ &\leqslant \frac{\deg(Q)/k}{(2^k - 2^{k/2})/k} \\ &\leqslant \left(\frac{n^2 m}{k}\right)\left(\frac{k}{2^k - 2^{k/2}}\right) \\ &= \frac{n^2 m}{2^k - 2^{k/2}} \\ &\approx \frac{n^2 m}{2^k}. \end{aligned}$$

Discussion. Consider the address space of a 64-bit x86 system. Although addresses are 64 bits, only the lowest 48 bits are used. Because x86 is byte-addressable, a virtual address space contains at most $2^{48} * 8 = 2^{51}$ bits. If we use a 256-bit hash, then we select a random irreducible 256-bit polynomial as our modulus. Now suppose that a generating extension explores no more than sixteen million states.[15] Note that 16,000,000 $< 2^{24}$. Then, the probability of there being any pairwise collision during the execution of a generating extension that uses up to 16,000,000 states is bounded by

$$\frac{(2^{24})^2 * 2^{51}}{2^{256}} = \frac{2^{99}}{2^{256}} = \frac{1}{2^{157}}.$$

Noting that the age of the universe is estimated to be approximately $1.13 * 2^{92}$ nanoseconds, we conclude that $\frac{1}{2^{157}}$ is an acceptable probability of intra set collision—i.e., it is far less than the chance of selecting a random nanosecond from the lifespan of the universe.

[15] In our experiments, far fewer than four million states are explored.

2.4 Pointer Symbolization

Lifting values of a simple non-reference type is straightforward. For example, given an arithmetic expression such as `dynamic = static + dynamic`, where all operands are `int` values, and a state [`static` $\mapsto$ 2], a generating extension can emit `dynamic = 2 + dynamic`.

However, when specializing C, lifting address values is more challenging. Due to the source-to-source nature of the specialization algorithm, we do not have direct control over the memory layout of the compiled and linked residual program. It is difficult at best to guarantee that the specialization-time value of an address such as, e.g, `&stack_var` references the same variable in the correct stack frame in the residual program.

Even if one could guarantee perfect control over the linker and loader, heap memory would be even more problematic; one would have to enforce complete equality between the layout of the statically known portion of the heap at specialization-time and at execution time of the residual program.

For these reasons, we cannot simply emit lifted pointers verbatim, unlike, e.g., `int` or `char` values.

Consider a concrete example, where `g` is a static `char` array, `s` is a static `char *` variable, and `d` is a dynamic `char` value, respectively:

```
(1) s = &g[0];
(2) s += 1;
(3) *s = d;
```

Because line (3) computes a dynamic result using the static operand `s`, `s` must be lifted. If the base of `g` is 0x1000, a naive lifting approach would emit `*((char*)0x1001)= d`. However, nothing about our source-to-source transformation enforces any relationship between the memory layout of the generating-extension binary and the memory layout of the residual-program binary. Consequently, we cannot guarantee that 0x1001 refers to the second element of array `g` in the residual program.

Interpretation-based approaches, such as [Srinivasan and Reps, 2015], can resolve this issue by treating addresses as symbolic values: operations such as the reference operator `&` and `malloc` yield a pair, consisting of a symbolic base address and an offset, and subsequent pointer arithmetic operates on the offset. Thus, in our example, at (1), `s` would hold (g_base, 0), and (2) would yield (g_base, 1). At (3), with such an approach, one can *symbolically lift* `s`, emitting `*(&g + 1) = d`.

Because we wish to produce generating extensions that execute natively on hardware, we do not have the luxury of using symbolic values without resorting to instrumenting all instances of address arithmetic. However, the fact that generating extensions are C programs affords us the opportunity to convert addresses to symbolic base/offset pairs *at lifting time*. In §5.1.5, I explain this *lazy-symbolization* process in detail.

Chapter 3

OS-Assisted State Management

For generating extensions produced by GenXGen to be able to correctly produce a residual program, the following two questions from §2.1.5 must be answered:

> How can a generating extension efficiently (1) save and restore native hardware states, and (2) compare them for equality?

A number of different elements are part of our solution. Fig. 3.1 depicts how the different ideas that support our memory-management technique fit together. To address issue (1), we take inspiration from symbolic execution tools such as KLEE [Cadar et al., 2008], and use two OS-level

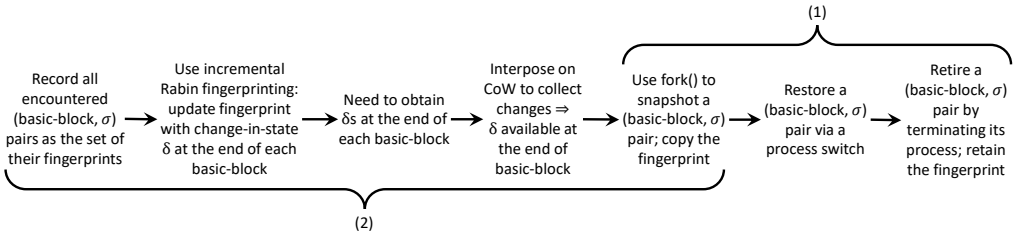


Figure 3.1: Diagram of how the ideas that support our memory-management technique fit together. (1) Ideas used to save and restore program states efficiently; (2) ideas that support an efficient means for determining whether a state has repeated.

mechanisms—(i) `fork()` (with copy-on-write (CoW)) and (ii) process context-switching—to create an efficient mechanism for saving and restoring states. Moreover, the elements in a generating extension's worklist can be represented by a set of process IDs.

For issue (2), the main reason why we could devise an efficient solution was that we changed the requirements slightly. In past work on partial evaluation, two properties of the method for checking equality of partial states have been paramount: given a partial state σ, the state-comparison procedure must determine whether σ has been previously encountered, with *zero false positives*, and *zero* (*or few*) *false negatives*. These properties are used to ensure that the partial evaluator both produces a correct residual program and terminates. A false positive is a soundness issue: it folds together parts of the program that correspond to *different* partial states, and hence can cause the residual program to fail to execute correctly. A false negative, while not desirable, may be tolerable: it would cause extra code to be created in the residual program (which would thus take up more space)—although in the worst case, it could cause the partial evaluator to fail to terminate.

In §3.1, I explain why prior approaches to state-representation are unsatisfactory for solving (1) and (2) in generating extensions for low-level languages. §3.2 explains the how Linux implements the processes abstraction and the associated copy-on-write mechanism by using hardware support for virtual memory. In §3.3, I expand on how my approach to (2) differs from prior implementations of state comparison—particularly my choice to abandon soundness—and show how the process-based snapshot implementation allows me to implement a hash-based approach to state comparison that provides a strong probabilistic guarantee on the absence of collisions. I argue that, despite being unsound, the negligible collision probability is, for any reasonable, physically realizable workload, indistinguishable from soundness (as justified by the bound presented

in §2.3.3 that the probability of there being any pairwise collision during the execution of a generating extension that uses up to 16,000,000 states is bounded by $\frac{1}{2^{157}}$). In §3.4, I explain how GenXGen generating extensions use the OS-level information made available from CoW to update the hash using Rabin Fingerprinting (§2.3).[1] §3.5 concludes the chapter.

3.1 Issues With Prior Snapshot Approaches

In C-Mix-style [Andersen, 1994a; Makholm, 1999] generating extensions, such as the ones given in Fig. 2.4 and Fig. 2.5 from §2.1.5, the state snapshot is explicitly materialized as a structure. However, this approach is potentially problematic from a performance standpoint. For example, in `handle_block_4` from Fig. 2.5, the program's stack state must be explicitly restored:

```
char *p = S.p; char *pat = S.pat;
```

Implementing save and restore via a state structure poses several challenges for C-Mix-style generating extensions, which must address the full memory semantics of C. In particular, the key advantage of a generating extension is that the statically-executing portion of the program is, as much as possible, constrained to the semantics of compiled C code executing natively on hardware. Consider the case where, when taking a snapshot, the stack contains a reference to another stack variable. For example, immediately prior to taking a snapshot, the static statement `p = &stackVar` is executed. Whenever a saved subject program state is restored, the memory layout of the stack must be restored in a manner that ensures all references to objects on the stack are still valid after restoration.

If this statement is implemented verbatim in the static-code component of the generating extension, then—after that code is executed—`p`

[1] In Chapter 4, §4.4 explains how I adapted the traditional C-Mix style of generating extension to use GenXGen's process-based state representation.

contains the concrete address of a location in the generating extension's specialization-time stack. If the generating extension ensures that the stack base of each snapshot is always at the same location, then the simplest way to save and restore a state such that the value of `p` is a valid reference to `stackVar` is to save and restore the entire specialization-time stack. However, this approach is costly in terms of both time and space. A full record of the stack must be recorded for every saved state, the entire stack must be copied at the end of every basic-block execution, and when a block/state pair is de-queued from the worklist, the saved stack must be copied back to the stack base.

In the presence of heap memory,[2] the obligation (1) of saving and restoring states while preserving the correctness of pointers becomes more difficult. One might consider saving and restoring only reachable memory objects by performing a mark and sweep from every pointer-typed variable on the stack. However, due to C's weak typing guarantees, it is difficult to come up with a rational and safe implementation of "reachable from the stack," because an uninitialized reference may contain garbage values, or references to freed memory. The call `free(p)` does not zero out `p`, and even in the absence of undefined behavior, such as use-after-free, the semantics of a mark-and-sweep-based snapshot are ill-defined. Thus, generalizing the stack-copying approach entails copying the entire `malloc()` arena.

The problem becomes even more difficult for specializing stripped binaries. In this case, at best, techniques from tools such as CodeSurfer can extract a very coarse-grained classification of memory regions into reference and non-reference types. Moreover, the lack of debugging symbols means that fine-grained information about the internal structure of `struct` types is lost, and thus it is not clear what mark-and-sweep means in this

[2]Here "heap memory" refers to what is conventionally referred to as "dynamically allocated memory," i.e., memory allocated via procedures such as `malloc()`. However, to avoid confusion with the unrelated notions of "static" and "dynamic" binding times, I will avoid using the term "dynamic allocation" in this dissertation.

context. Thus, to produce a low-level-language-agnostic generating extension runtime, it is best, for the purposes of a snapshot implementation, to treat memory as a collection of of undifferentiated bytes. However in the worst case, this approach entails copying and restoring all live, non-code[3] memory for a state.

Interpretation-based partial evaluators for type-safe high-level languages may be able to take a better approach. Because static instructions are interpreted, and not executed natively, the implementer of a partial evaluator is free to choose a state representation suited to the needs of partial evaluation. For example, a state can be represented as a lookup table mapping symbols to memory objects, along with a collection of memory objects. When specializing a block at a state, the interpreter can simply hold a reference to the current state table. Restoring a state, then, is merely a matter of changing the "current state" reference.

Moreover, research into purely-functional languages[4] has lead to the creation of *applicative data structures*, which are data structures for which updates produce a new data structure, and do not destroy the old data structure [Reps, 1984, Chapter 8][Myers, 1984; Okasaki, 1999]. Thus, a program can insert a value v into an applicative dictionary D, and obtain the root of a new applicative dictionary D′, such that all references to D are still valid references to the old dictionary, while D′ is a new dictionary that (i) contains v, and (ii) shares the bulk of its (tree) structure with D. Applicative dictionaries can be used to implement the state representation for an interpreter for an imperative language.

For a partial evaluator, an applicative-dictionary-backed state representation admits $O(1)$ state swapping, while also reducing space usage by sharing unchanged state between structures. However, applicative

[3] GenXGen does not handle self-modifying code, and in practice assumes all subject programs conform to a `W^X` protection policy.

[4] I.e., functional languages such as Haskell that do not permit side-effects or other forms of imperative state update.

structures do not easily admit O(1) state-equality checking. In practice, one can hash, e.g., an AVL-tree in such a way that the *hash-value* is identical for different AVL-trees that represent the same state, which yields O(1) state-*inequality* tests, but not *equality* tests.

A state-management technique that improves upon applicative-structure based approaches by solving *both* item 1 and item 2 (from the beginning of the chapter) with O(1) operations—while also being usable outside of interpreter-based approaches—is a property that would be desirable in a state-management scheme for GenXGen. However, like C-MixII, the statically-executing portion of the program is constrained to the semantics of compiled C code executing natively on hardware. Thus, for a compiled generating extension, a static statement like `a = a + 1`, the symbol `a` corresponds to some concrete hardware location—either a memory address or a register. This situation precludes a straightforward way of representing state via a swapable lookup table that maps symbols to values, as one would do in an interpreter.

Moreover, the semantics of C and x86 machine code are merely maps from one hardware state to another. That is, a static state in a GenXGen generating extension consists of hardware registers, the instruction pointer, and the contents of memory. Thus, if we wish to produce a language-agnostic generating-extension runtime, the state-management problems become a matter of saving and restoring full hardware states. However, by design, widely-adopted operating systems, in conjunction with hardware, already provide a means of saving and restoring full hardware states. The desired behavior is precisely what the process abstraction in, e.g., Linux on x86-64 provides. We thus take an approach inspired by symbolic execution engines such as KLEE [Cadar et al., 2008], which represent states with OS processes.

3.2 The Process Abstraction on x86 Linux

Effectively, by working in concert, the OS and the CPU ensure that every process has an isolated address space. From the perspective of machine-code-level load and store semantics, each process has exclusive access to a full, nearly-unbroken[5] 32 or 64-bit *virtual address space*. More succinctly, the CPU and OS present a view of hardware that allows a process to run as if it were the sole program running on the system. In its ordinary execution, the operating-system kernel is constantly changing which programs are actively executing on hardware, and, using special hardware-level mechanisms provided by the CPU, the OS is able to swap active processes in constant time.

The key hardware construct that provides this process abstraction is *virtual memory*. Virtual memory decouples the logical[6] addresses (referred to as *virtual addresses*) manipulated and accessed by a program from the actual hardware locations these virtual addresses refer to (these hardware locations are referred to as *physical addresses*). For example, although a program may execute `v = *(0x40000)`, the virtual address `0x40000` may correspond to the physical address `0x1000`. When executing load and store instructions, the CPU translates virtual addresses to physical addresses using lookups into an in-memory address-translation table managed by the OS, aided by an on-CPU cache of virtual-to-physical mappings.

On x86, virtual memory is implemented by dividing memory into *pages*. Both virtual and physical memory are partitioned into 4096-byte regions called pages; every "live" page in a process maps to a underlying physical page. For each process, the OS maintains a *page table*, which contains a

[5] A small virtual region contains OS-kernel-reserved memory, which is marked as inaccessible to programs. Ordinary reads and writes to these regions are not permitted. This region is necessary for the OS to be able to function correctly while also providing the process abstraction. A given process's virtual address space is not similarly cluttered with regions belonging to other processes

[6] Here "logical address" means "visible to a program at the semantic level," and not "logical address" in the sense of the (largely obsolete) x86 segmentation model

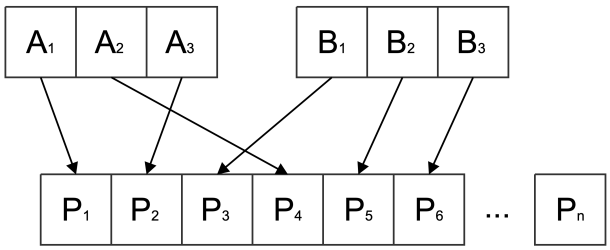


Figure 3.2: An example virtual-memory configuration for two three-page processes, A and B in a system with n physical pages.

list of translations from virtual to physical addresses. The CPU holds the address of the current page table in a special-purpose register; when executing loads and stores, the table is used to translate virtual addresses to physical addresses, and the CPU caches recent virtual-to-physical page translations to avoid excessive page-table lookups.

For example, Fig. 3.2 depicts a system with n pages of physical memory, and two processes. Note that adjacent virtual pages, such as A_2 and A_3 need not be physically adjacent: virtual addresses are only guaranteed to be physically contiguous inside pages.

Swapping from process P to P′ on x86 CPUs thus consists of saving P's CPU registers, restoring P′′s registers, and changing the CPU's "current page table" register to reference P′′s page table, and is thus an $O(1)$ operation.

Moreover, Linux identifies every process with a unique numeric ID, termed a *process id* or PID. This PID serves as a handle to a process, and Linux exposes several APIs that use PIDs to set up communication and synchronization channels between processes, and to send signals to other processes.

3.2.1 Using Processes to Implement State Snapshots

The generating extensions created by GenXGen use different OS processes to represent different state snapshots. A set of snapshots is represented by a set of PIDs. Thus, the generating extension is a multi-process program; a single *controller process* manages a collection of *state processes*. The worklist consists of block/PID pairs. From the perspective of the controller, specializing a block consists of de-queuing a pair, and sending an IPC signal to the process associated with the state-PID, instructing it to perform specialization at the specified block. The controller then halts and awaits a signal from the post-state-process. Upon completion, the post-state-process sends a message to the controller process, consisting of the PID of the post-state-process, along with the ID of all successor blocks. The `snapshot()` operation used in Fig. 2.4, thus consists of spawning a new process prior to executing a basic block, and using the newly spawned process to execute the block.

To allow programs to create new processes, Linux provides the `fork()` system call. When a process A calls `fork()`, a new process A' is spawned, and both processes return from fork. At this point both A and A' have identical register values, memory contents, and value of the instruction pointer, except for the return value from `fork()`, which is 0 in the child, and the PID of the child in the parent.

To understand the implementation of `snapshot` via fork, consider the execution of the process `P` representing state σ at some block `b`. Process A receives a message from the controller telling it to perform specialization of `b`. `P` calls `fork()`. Upon return from `fork()`, A checks the return value, and upon seeing that it is non-zero, suspends execution, awaiting further signals; the child process A' is reserved as the process to represent the post-state σ' of the execution of `b` on σ. A' then executes block `b`, and sends a completion signal to the controller, along with its PID, which is registered as the canonical representation of σ', if σ' has not yet been

visited. Then A' suspends execution and, like A, awaits further execution signals.

It is important to note that while GenXGen is implemented as a multi-process program, it is *not* a parallel application. At any given point, either the controller process is executing, or exactly one of the state processes is executing. Thus, there is no need for, e.g., non-trivial synchronization constructs, and no dependencies that may lead to deadlock.

The call to `snapshot()` is an $O(1)$ operation, because `fork()` is implemented in a time and space-efficient fashion through *copy-on-write* (CoW). Rather than allocating new physical pages for A', every page $F_{(A',i)}$ in A' is mapped to the same physical page G as the corresponding to $F_{(A,i)}$ in A. However, the virtual-to-physical mappings for A' are flagged as CoW using hardware support. When A' writes for the first time to a page $F_{(A',i)}$ inherited from A, a hardware fault occurs, the changed version of the page is allocated its own page G' in physical memory, and the hardware state is updated so that $F_{(A',i)}$ is mapped to G'.

For example, consider the state of memory before and after the CoW fault pictured in Fig. 3.3. In Fig. 3.3(a), process A has just returned from `fork()`, and has spawned a child process A'. Later, A' writes virtual page A'_3, triggering a CoW fault. Before the write, the CPU interrupts A', and the OS copies the contents of P_3 to a free physical page, P_5, and updates the page table of A' so that A'_3 now references P_5, and the write completes, this time targeting the physical page P_5.

In addition, the copy-on-write mechanism provides a means of implementing a more efficient means of checking state equality, based on an incrementally-updatable hash algorithm, as discussed in §3.3 and §3.4.

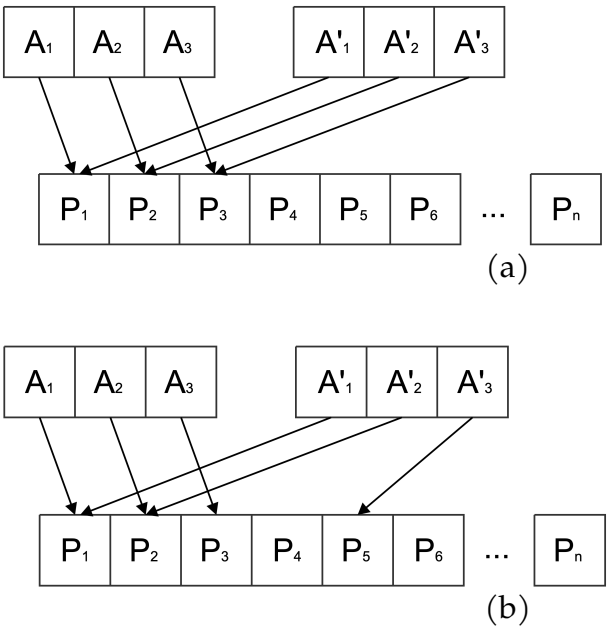


Figure 3.3: The state of virtual memory before (a) and after (b) a CoW fault. Configuration (a) denotes the state of memory immediately after A calls `fork`, and (b) denotes the state of memory immediately after child process A' writes to virtual page A_3.

3.3 Using OS Mechanisms to Implement Incremental State Hashing

In our work, partial states are native hardware states, which caused us to reconsider the ground rules for checking partial-state equality. Clearly, determining state equality by full comparison of two process address spaces—even if constrained to global variables, stack, and heap—is prohibitively expensive. Every state process A must be retained, even if no instances of A are referenced by the worklist. Moreover, because equality is determined via full state-comparison, a new state process A_σ must be checked against all other state processes. Thus, an end-to-end execution of a generating extension entails $O(N^2)$ state comparisons.

One could take a conventional hash-based approach, and retain a hash table of previously-seen states. However, in the conventional approach, false positives in the equality check are eliminated by performing direct comparisons if a collision occurs in the hash table. Thus, if A and A' have the same state hash, a full comparison of all live pages in A and A' must occur.

Thus we further relaxed the constraints on checking partial-state equality, and settled on the following properties:

1. The procedure must be space-efficient and time-efficient:
   a) We want to store at most a few hundred bits per state.
   b) State-equality checks should take constant time.
2. There should be no false negatives.
3. The false-positive rate must be kept acceptably low.
4. One must be able to incrementally update the value that characterizes a visited state.

Hashing—with the exception of collision-handling—satisfies (1b): the contents of a previously seen state can be represented as a 256-bit number. Moreover, item 2 is obtained for free with hashing.

Item 3 deviates from the conventional approach to state management in partial evaluation. In our approach, *we do not insist that there be a mechanism to resolve collisions*, as long as we have control over parameters that ensure that *the probability of a collision* ***ever*** *arising is below a value of our choosing*. In other words, we allow the use of a hash, as long as the parameters of the hash function can be tuned to keep the collision probability below an acceptable threshold. As described in §2.3, by choosing a hash-size k, we can tune the collision probability, which is proportional to $1/2^k$. Moreover,

as we will see in §3.4, by exploiting available OS-level information, we can use the efficient-incremental-update property of Rabin fingerprinting to satisfy 4.

Moreover, because we do not need to resolve collisions, we automatically satisfy 1a: to record the set of previously encountered partial states, during specialization it is only necessary to keep *the set of hash values of partial states*, rather than the partial states themselves. In our implementation in GenXGen[C], a hash value is a 256-bit value, so checking equality of partial states is performed by comparing 256-bit values.

Why is a Probabilistic Guarantee Acceptable? Although it is possible for GenXGen[C] to produce an unsound residual program due to a hash collision, by choosing appropriate values for parameters of the hashing scheme, the probability of a collision can be made arbitrarily small. The counting arguments from Rabin [Rabin, 1981] and Broder [Broder, 1993] discussed in §2.3.3 establish that with the parameter values used in our implementation, for any single run of GenXGen[C] in a 64-bit byte-addressable address space that produces a residual program with up to 16,000,000 basic blocks, the chance of an incorrect program being generated due to a hash collision is 2^{-157}, and hence negligible.[7]

The reliance on a probabilistic guarantee runs counter to principles generally accepted in the PL community, and may be seen as heretical by some. Nevertheless, there are good reasons for considering the approach. In particular, similar ideas are used in other systems—some with even higher risks than the ones we have chosen to accept in GenXGen[C].

One example is the Vesta build-automation tool created at DEC SRC [Heydon et al., 2006]. Vesta has a sophisticated function-caching system to avoid rebuilding components whenever possible. Each cached object has

[7]Note that we are not tripped up by a "birthday-paradox" situation. The circumstances are roughly as if we have arranged for a year to have the right number of days so that, with any randomly chosen group of 16,000,000 people, the chances of winning a birthday-paradox bet is less than 2^{-157}.

an identifying fingerprint that is computed according to the object's build provenance. A fingerprint collision would lead to the reuse of an out-of-date object, and hence Vesta provides only a probabilistic guarantee that a component is built correctly. In particular, "Vesta uses 128-bit fingerprints. Based on an overall system size of 20 million source lines ... and some conservative estimates about the number of versions of each source file, the probability of a collision occurring over the expected lifetime of the Vesta system is much less than 2^{-42} [Heydon et al., 2006, p. 117]." Moreover, with Vesta's 128-bit fingerprints, the probability that a given build reused something incorrectly must be at least 2^{-128}. Hence, if your invocations of GenXGen[C] were under control of Vesta, an erroneous residual program is at least $\frac{2^{-128}}{2^{-157}} = 2^{29}$ ($\approx$ 537 million) times more likely to occur because of a Vesta issue than a GenXGen[C] issue—and probably more like $\frac{2^{-42}}{2^{-157}} = 2^{115}$ ($\approx$ 42 decillion) times more likely to occur because of a Vesta issue.

Another system that relies on the chance being negligible that there is ever a hash collision is the Solidity programming language, running on the Ethereum Virtual Machine (EVM) [Marx, 2018]. Conceptually, the storage model for a smart contract running on the EVM is a map from 256-bit indexes to 256-bit values. Whereas fixed-size Solidity values are located in slots at the low end of memory, the base of a dynamically-sized Solidity array A is located in a slot computed by hashing on a quantity associated with A. Similarly, the slot for an element in the range of a Solidity mapping M is computed by hashing on a quantity associated with M, together with the value of the key `k` whose element is to be accessed. (That is, the value is located in slot `hash`$(M, \texttt{k})$.)

If any hash collision were to occur, the operation of the smart contract could be at odds with the intended (non-hash-based) semantics of Solidity. However, because the range of `hash` is a 256-bit integer, the probability of any hash collision occurring during the execution of a Solidity smart contract is negligible (at least for the kind of smart contracts being written

today).

3.4 Incremental Updating of State Hashes

To create efficient generating extensions, incremental updating of hash values of states—property 4 from §3.3—is crucial. With an appropriate choice of hash algorithm, it is possible to satisfy property 4 by taking advantage of CoW at each end-of-block worklist update. As discussed in §3.2, each partial state is a separate Linux process. An additional "controller" process oversees the generating extension's worklist of (partial-state, basic-block) pairs, and serves as a dispatcher for the specialization phase. The worklist of unprocessed (basic-block, partial-state) pairs is implemented as a set of process IDs. Each time the dispatcher selects a pair (σ, b) from the worklist, it signals the process P_σ that represents σ, and P_σ begins executing. P_σ immediately calls `fork()`, creating a child process P'. Initially, the logical address space of P' contains σ, and its program counter is set to b. After P' finishes executing block b, its logical address space contains state σ'. However, only the pages that *changed* during the execution of block b on σ are specific to P'; the rest are shared with process P_σ.

An invariant of the system is that, except for the current process P', the system has in hand a hash value for the state of each process. To compute the hash value for P' incrementally and efficiently, the system uses (a) the known hash value for P_σ, (b) state σ of P_σ, (c) state σ' of P', and (d) a record of which pages of σ and σ' differ.

Item (d) is obtained by interposing on CoW to collect the list of pages dirtied during the execution of process P'. While P' executes, the first write to each page induces a CoW fault. We used eBPF [Cassagnes et al., 2020] to interpose on each CoW fault to intercept invocations of the kernel's page-fault handler and collect a record of all pages dirtied during the execution of a basic-block process, such as P'. This record—together with

states σ and σ'—represents the "delta" between P and P'. Using the method described in §2.3.2, this information allows the hash value of P' to be computed *in time proportional to the size of the delta*. Thus, executing a basic-block b on a given state σ incurs a cost that is linear in the number of memory-writing instructions in b: each such instruction is executed only once, incurs at most one CoW fault, and contributes a constant-size entry to the record of CoW faults (which, in turn, causes a constant amount of computation in the incremental computation of the hash value for P').

3.5 Discussion

Using an approach inspired by symbolic-execution engines such as Klee [Cadar et al., 2008], we represent states with processes. By this means, we are able to improve upon the state-of-the-art for classical C-Mix-style generating extensions, and implement a state representation that allows generating extensions in GenXGen to both (1) save and restore native hardware states and (2) identify repeated states in $O(1)$ time. By exploiting the copy-on-write feature of Linux processes, we can interpose on CoW faults, and identify page-level changes, which are used to incrementally update state hashes. In Chapter 1, I described one of the key advantages of generating extensions: the fact that they are *shallow embeddings* of a program specializer into the subject language. That is, the static portion of the subject program need not be interpreted: it can simply execute as is. The process-based state-management mechanisms used by GenXGen allow GenXGen's generating extensions to perform a worklist-marshalled specialization algorithm while remaining a shallow embedding. The key insight is that, in practice, the semantics of a low-level language extends beyond the language itself to the runtime environment provided by the hardware and operating system. That is, in practice, the process abstraction and IPC faculties provided by the OS and standard libraries are, for

all intents and purposes part of the language semantics for a program executing on real hardware. It is precisely by taking this less abstract, more pragmatic view of language semantics that GenXGen is able to exploit OS features to implement specialization.

In Chapter 6, experimental evaluation show that these technique allows real-world programs to be specialized in a reasonable amount of time, with the majority of feature-removal tasks completing in under one minute.

Chapter 4

The Ge-Gen Algorithm

In this chapter, I describe the Ge-Gen phase of GenXGen. The standard approach from the literature[Jones et al., 1993; Andersen, 1994a], as described in Chapter 2, uses binding-time-analysis results to instantiate the classical specialization algorithm in a single program-specific specializer. This construction rewrites the subject program in a basic-block-by-basic-block manner, with each block b transformed into a procedure that takes a partial state σ as an argument, and produces a b_σ, a version of b specialized on σ. These specialized blocks are combined with a top-level control procedure that marshals the state-space exploration.

Although GenXGen produces generating extensions that are similar to classical C-Mix-style generating extensions in many ways, GenXGen differs in a key way: the binding-time-analysis results are polyvariant. That is, in a classical generating extension, like the one discussed in §2.1.5, there is one specialization procedure for each basic block, the BTA results only contain one partition of the vertices into static and dynamic. However, because GenXGen uses specialization slicing [Aung et al., 2014], there may be multiple result partitions for every procedure, and hence for every basic block. In this chapter, I show how GenXGen produces generating extensions that are *binding-time polyvariant*. Although polyvariant binding

times have been used before in program specialization, for example to partially evaluate a Scheme dialect [Jones et al., 1993], my work represents the first time it has been done for a generating-extension-generator for C or machine code.

In §4.1, I provide a condensed review of the salient ideas from §2.1.2. In §4.2, I discuss two notions of polyvariance: the first is the classic notion of *data polyvariance*—namely, that a procedure or block can be specialized with respect to multiple states, and contrast that with *binding-time polyvariance*. In §4.3, I show how, given a program P and a specialization-slice result, P can be transformed into a program P′ that is data polyvariant, but no longer binding-time polyvariant. (For one thing, multiple replicas of each procedure in P can be materialized in P′.) In §4.4, I discuss GenX-Gen's gegen algorithm for specializing programs that are data polyvariant, but not binding-time polyvariant. Thus, the pieces fit together as follows:

BTA via specialization slicing (§2.2.3 and §4.1)
→ Transformation to eliminate binding-time polyvariance (§4.3)
→ Creation of process-based generating extension (§4.4)

4.1 Summary of Slicing as a BTA Algorithm

In §2.1.2, we discussed Binding Time Analysis (BTA), the process of taking an initial set of binding times for a program P's inputs, which partitions the inputs into static and dynamic sets, and propagating this partition to every program point in P. To produce a correctly-behaving generating extension, a BTA algorithm must be congruent; any value computed at a static program point must not depend on values computed by program points marked as dynamic.

In §2.1.2 and §2.2.1, we discussed the use of forward slicing [Weiser, 1984; Reps et al., 1994] to implement a BTA algorithm. Given a set V of program points in program P, a forward data-dependence slice of P with

respect to V computes the set of all program points in P that may depend on V. For a single procedure, the standard forward slice from dynamic inputs produces a congruent BTA result: the slice result is the dynamic set; the complement of the slice is the static set; and no program point p in the static set depends on the dynamic inputs (otherwise, p would be in the slice).

However, due to the parameter-mismatch problem described in §2.2.2, this result is not congruent for multi-procedure programs. To obtain congruent results, one must use Binkley's less precise reslicing algorithm [Binkley, 1993].

More precise results can be obtained by using specialization slicing [Aung et al., 2014], discussed in §2.2.3. The issue with standard forward slicing arises from the monovariance of the underlying SDG program representation—for every procedure F in P, there is one representation of F, and thus the slice results for F must incorporate information from all callsites at which F is called. The key insight of the specialization-slicing algorithm is that more precise results can be obtained by creating a copy of F for every callsite, then performing a forward slice.

In the presence of recursion, this inlining is infinite, but the specialization-slicing algorithm uses an automata-theoretic construction that only uses finite representations of the entities involved. Program P is encoded as a pushdown system, and the slice results are returned as a deterministic finite automaton (DFA) R. R encodes the result of a forward slice of the infinitely inlined program. In particular, R encodes a representation of the minimum set M(F) of variants of each procedure F, such that the structure of the variants and the calling relationship between all procedure variants encodes a congruent slice result over the infinite inlining. This produces the most-precise—i.e., coarsest—*polyvariant* binding-time result: every procedure may have more than one set of binding-time results.

```
blk_4_4h:                blk_4_4a:                blk_4_4t:
  if('h' != *s1)           if('a' != *s1)           if('t' != *s1)
    goto blk_6_6h;           goto blk_6_6a;           goto blk_6_6t;
  goto blk_5_5h;           goto blk_5_5a;           goto blk_5_5t;
```

[p ↦ "hat"] [p ↦ "at"] [p ↦ "t"]

Figure 4.1: The data-polyvariant residual versions of block 4 from Fig. 1.1(b) produced by the generating extension in §2.1.5, along with the state that produced the variants.

4.2 Polyvariance Overview

In the literature for classical partial evaluation, there are two distinct forms of polyvariance. The first is what I refer to in this dissertation[1] as *data polyvariance*. Given a single subject-program basic-block `b`, a data-polyvariant partial evaluator may produce multiple instances of a given block. For example, the generating extension in §2.1.5, which performs the specialization described in §2.1.3, produces multiple instances of the body of the inner loop of the program from Fig. 2.1(a) by unrolling the inner-loop body with respect to the target string "hat", as depicted in Fig. 4.1.

Conventional C-Mix-style generating extensions are data-polyvariant, and GenXGen is as well.

The other use of the term "polyvariance" refers to what I will call *binding-time polyvariance*. In a binding-time-polyvariant partial evaluator, a procedure may have multiple BTA-result partitions. In particular, this property is true of specialization slicing, as described in §2.2.3. For example, Fig. 4.2 shows the specialization-slicing results for the procedures `rec` and `swap` from Fig. 2.11.

Prior C-Mix-style generating-extension tools for C have only used

[1] In works such as [Jones et al., 1993], the term "polyvariance" is used fairly freely, and there, it is generally clear from context which meaning is intended.

```
void rec(k){
  swap(s,d) //cs1
  if(k > 0)
      rec(k-1);
  swap(s,d); //cs3
}
```

$$S_{\texttt{rec\_odd}} = \{\texttt{swap.a.actual\_in@cs1}, \texttt{swap.b.actual\_in@cs3}\}$$
$$S_{\texttt{rec\_even}} = \{\texttt{swap.b.actual\_in@cs1}, \texttt{swap.a.actual\_in@cs3}\}$$

```
swap(a,b){
  s = b;
  d = a;
}
```

$$S_{\texttt{swap\_l}} = \{\texttt{swap.a.formal\_in}, \texttt{d} = \texttt{a}\}$$
$$S_{\texttt{swap\_r}} = \{\texttt{swap.b.formal\_in}, \texttt{s} = \texttt{b}\}$$

Figure 4.2: The specialization slice results for `rec` and `swap` from Fig. 2.11. When considered as BTA results, these results are binding-time polyvariant. Each set $S_{\texttt{variant\_name}}$ represent a single procedure-variant slice result encoded in the result automaton R.

monoviariant binding-time analyses, and without significant modification, cannot handle polyvariant result sets represented in an abstract form, as in Fig. 4.2. Moreover, GenXGen's ge-gen pass expects a monovariant BTA result. However, given an appropriate program transformation, one need not modify the ge-gen algorithm itself to handle binding-time polyvariance. If one can convert a program with polyvariant BTA results—that is, a program with multiple distinct BTA result sets for each procedure—into a semantically equivalent program with a single BTA result set for each procedure, then one can then pass the resulting program and BTA result set to a standard ge-gen algorithm and obtain the desired results.

Concretely, given the slice results pictured in Fig. 4.2, we would like to be able to produce the `materialized slice` shown in Fig. 4.3. In this dissertation, I refer to producing such a program as *slice materialization*.

Given appropriately structured polyvariant binding-time results, Jones

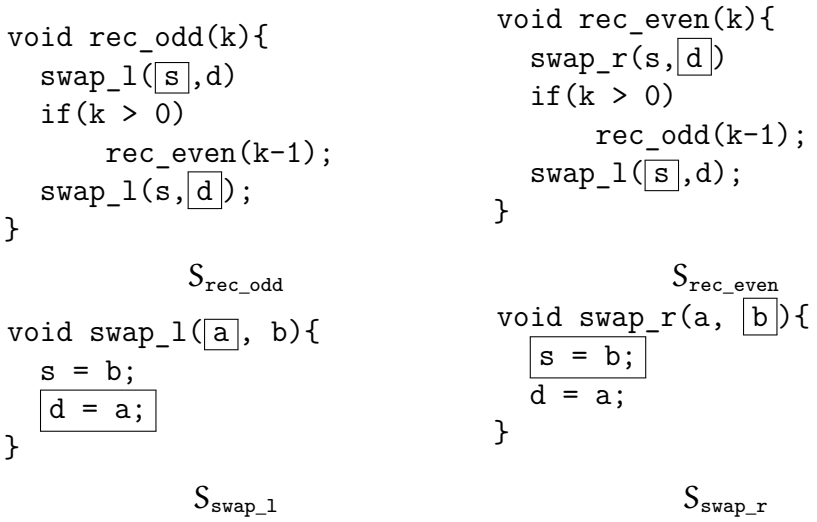


Figure 4.3: The materialization of the slicing results in Fig. 4.2, pictured along with the result set corresponding to each variant.

et al. [Jones et al., 1993], suggest an approach to performing slice materialization.

Let F be the set of procedures in the original program P, and let F' be the set of procedure variants.

- V: Maps each procedure $f \in F$ to to the set of BTA result sets for f. For example, $V(\texttt{rec}) = \{S_{\texttt{rec\_odd}}, S_{\texttt{rec\_even}}\}$.
- I: Maps each BTA result set to a unique identifier.[2] E.g., $I(S_{\texttt{rec\_odd}}) = \texttt{rec\_odd}$
- C: Given a unique identifier corresponding to a result set for some procedure $f \in F$, and a callsite c in f, C maps c to g', where g' is the unique identifier of the callee associated with the appropriate variant of c's callee g in f. For example, $C(\texttt{rec\_odd}, \texttt{cs1}) = \texttt{swap\_l}$ and $C(\texttt{rec\_even}, \texttt{cs1}) = \texttt{swap\_r}$

[2]For clarity, I use the variant names from the example, but in practice the identifiers are merely the name of the original procedure with a unique integer concatenated to it.

Given these three sets, Jones et al. assert that one can convert P to an equivalent *monovariant* program P'. However, he does not provide an explicit algorithm for doing so.

In §4.3, I give an algorithm for performing this transformation. However, Jones's algorithm sketch is formulated in terms of result *sets*. Specialization slicing yields an automaton R that encodes the result sets of the slice, adding another layer of indirection. Although one can extract the majority of the information to construct V from R, there is one case that is a notable exception. To rectify this, I show how to slightly adjust the approach suggested by Jones to produce the desired program.

4.3 A Slice-Materialization Algorithm

Given a program P and a slice-result automaton R, the slice-materialization algorithm produces P' by the following steps:

1. Extract functions V, I, and C from R.

2. Produce P_V, a rewritten version of P containing n identical versions of each $f \in F$, where $n = |V(f)|$. Each copy of f is associated with S, one of the result sets in $V(f)$, and renamed using $I(S)$, and is associated with the result set corresponding to the identifier used. These copies are the set of *variants* f' of f. Moreover, construct an $\hat{I}$, such that for all f', $\hat{I}(f')$ maps to the corresponding S.

3. Produce P' as follows: For each procedure variant f', map every callsite s by replacing s with an s' that calls $C(\hat{I}, c)$.

4. Create V', where $V'(f') = \hat{I}(f')$ is the slice result for f'. Thus, $V'(f')$ is the set of dynamic program points of f' when V' is used as the result of binding-time analysis of P'.

That is, to materialize the slice results, GenXGen construct a new program P_v with a sufficient number of copies to reflect the structure of the slice result. For example, in transforming the program fragment in Fig. 4.2, the materialization algorithm obtains two identical copies each of `rec`, and `swap`.

However, the copies still contain callsites referencing the original procedures in P. Thus, step 3 performs a fix-up to the callsites reflect the calling structure of the variants in the slice results, yielding P'. E.g., the first callsite in `rec_even` is mapped to `swap_r`, and the first callsite in `rec_odd` is mapped to `swap_l`.

Note that V' maps each resulting f' in P' to a single result set, and is thus binding-time monovariant.

As stated in §4.2, the automaton R returned by specialization slicing does not quite contain enough information for the standard slice-materialization algorithm to produce the desired V, I, C. Let $\hat{S_f}$ be the *null slice result* of a procedure f, where the null result is the empty slice result, and let the term *null variant* refer to the version of f in P' corresponding to a null result. R does not explicitly encode $\hat{S}_f$, and because of this, it is not clear whether a given procedure has a null slice result. Thus at step (2), it is not possible to determine if a copy of f corresponding to the null variant $\hat{S}_f$ needs to be made. Stated differently, given V_R, where V_R is the version of V extracted from R, it is not clear whether $|V_A(F)|$ (in the case where $\hat{S}_f \notin V(F)$ or $|V_R(F)| + 1$ (in the case where $\hat{S}_f \in V(F)$) copies of f need to be made. For example, in Fig. 4.4, a null variant of p is necessary, while a null variant of q is not.

However, the structure of P coupled with the relational information in C is enough to recover the correct materialization, because of the following property of specialization-slicing results (not proven here): whenever a null result for f exists, and the corresponding null variant is called by

```
int g1,g2;
main(){
  a = 0;
  p(a)
  a = 1;
  q(a);
}

void p(int x){
  g = x + 1;
}

void q(int y){
  g2 = y;
}
```

Figure 4.4: A program in which p does not have a null slice result, while q does.

a non-null variant, or the null variant calls a non-null variant[3], that call relation is recorded in C as a call to or from, respectively, the original version of f. This property is sufficient to ensure that if the materialization algorithm adds a copy of f corresponding to $\hat{f}$ for every $f \in F$ in (2), then $\hat{f}$ corresponds to an extant null result if and only if $\hat{f}$ is reachable from `main`.

For example performing the un-modified materialization algorithm, creates a variant p_1 of p, and main is rewritten using C, yielding.

```
main(){
 a = 0;
 p_1(a);
 a = 1
 q(a);
}
```

Now consider augmenting the materialization algorithm by creating null copies p and q at the end. Because the call to p in `main` is replaced by the call to `p_1`, p is unreachable from `main`. However q is reachable from `main`, and the final reachability pruning step removes p, but not q.

Consequently, the GenXGen slice-materialization phase performs the modified step 2, and then prunes all unreachable procedures. This ap-

[3]To see why this is possible, note that the source of a slice can occur inside a procedure p, and code within p can be in the slice, but it is entirely possible for there to be no formal-out parameters of p to be in the slice. Thus, callers of p need not have any transitive dependences on the slice results for p.

proach yields a monovariant P′ that materializes the specialization-slicing results, and hence is suitable to be handed to the Ge-Gen algorithm.

4.4 The Ge-Gen Algorithm

Given a program P and a monvariant BTA result, the ge-gen phase of GenXGen produces a generating extension in a manner largely analogous to the transformation described in §2.1.5. That is, for every procedure F in program P, ge-gen traverses the program's interprocedural CFG—constructed by CodeSurfer [codesurfer, 2018; Anderson et al., 2003], a program-analysis tool that constructs program SDG, PDGs, and CFGs—and produces a basic-block-procedure for every basic block in F.

Within a basic block, the non-control-flow vertices are transformed based on their binding time, and gegen also adds appropriate code to generate residual control-flow constructs, and to perform state-management bookkeeping.

In this section, I will describe how the transformation is implemented in GenXGen, providing more concrete implementation details for control-flow, and describing how GenXGen's approach to lifting differs from the classical approach discussed in §2.1.5. I will also note several subtleties, the full analysis of which will be deferred to Chapter 5. The

GenXGen's gegen phase is, as described, a transformation of each procedure's basic-block-level CFG, which transforms each basic block into a basic block procedure. This basic-block-level CFG is a representation of a program as basic blocks connected by control-flow edges. Each block contains multiple assignment statements, and optionally a final control-flow statement. Moreover statement-level structure is normalized. If a statement has multiple assignments, as in `*p++ = 5`, CodeSurfer decomposes it into multiple statements, with one assignment for each statement. Moreover, the statement that computes the condition of a conditional branch

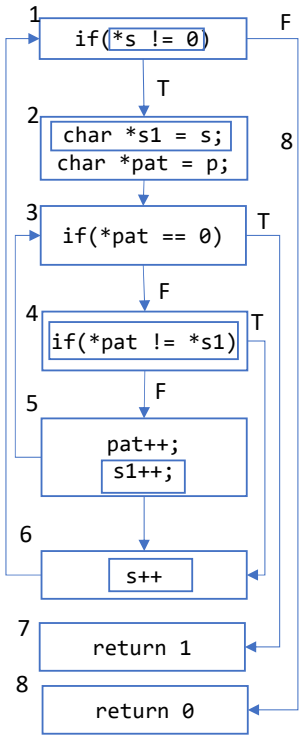


Figure 4.5: the CFG of `match` from Fig. 2.1

statement is separated from the control-flow construct. For example, the if statement from block 4 in Fig. 4.5 is converted to the following code:

```
cond = *pat != *s1
if(cond)
```

4.4.1 Block-Procedure Structure and Utility Macros

The conventional Mix-style block-procedure structure. In §2.1.5, each basic-block procedure in Fig. 2.4 and Fig. 2.5 contains calls to several utility procedures:

1. `snapshot()`, which records the current state.
2. `printf(fmt_str, ...)`, which is used to emit residual code

3. `contains(visited, block, state)`, which checks if the generating extension has already visited `block` at `state`, by looking up (`block`,`state`) in `visited`.

4. `insert(visited, block, state)` inserts (`block`,`state`) into `visited`

5. `worklist_enqueue(worklist, block, state)` puts (`block`,`state` into the worklist.

Each basic block procedure takes a worklist and state object as arguments, and has the following overall structure: first, static state is restored from the state parameter, and a block label is generated. Next, comes the code corresponding to the subject program's basic-block body. This code consists of static code interleaved with emit statements. Finally, the block contains procedure calls that snapshot and record the post-state, emit residual control flow, and enqueue successor block/state pairs, if they have not been seen yet.

The GenXGen block-procedure structure. In Chapter 3, I described how GenXGen represents states as processes. Because of this, we reconsider the structure of basic block procedures, exploiting the advantages of the new state representation, and implement the block structure using a slightly different and smaller set of *utility macros*. The implementation of these macros (and the rationale for them being macros and not procedures) entails the need to handle several subtleties relating to the process-based state representation and the full-memory hashing. These implementation concerns are discussed in more detail in Chapter 5.

The main difference from the overall C-Mix-style implementation is the fact that GenXGen generating extensions represent each state as a unique process (a "state process"), managed by a single controller process. Moreover, as described in Chapter 3, the worklist manipulations are handled

by the controller process. The controller sends a basic block ID to the appropriate state process, and the state process executes the procedure corresponding to the block ID.

Because the state process already contains the relevant state, and also performs no worklist manipulations, there is no need for an explicit, per-variable/memory-location restoration of state. Thus, the block procedure does not take either a state object or the worklist as a parameter. Moreover, when the controller requests that the process corresponding to state σ perform specialization at block `b`, the state process calls `fork()` before calling the block procedure for `b`. The new child process executes the block procedure. The OS copy-on-write mechanism implicitly performs the snapshot, by making copies of altered pages in the child process.

Thus, GenXGen has a simplified set of utility macros compared to the C-Mix-style generating extensions depicted in §2.1.5:

1. `emit(fmt, ...)` is a macro that emits a residual statement, and is parametric like printf.

2. `worklist_enqueue` combines the Mix-style utility procedures (3), (4), and (5) into a single macro. The `worklist_enqueue` macro submits an IPC request to the controller, telling it to enqueue the block/successor-state if it has not been enqueued before, and to record the pair in the controller process's visited set if it is successfully enqueued.

3. `getPreStateID()` obtains a unique numeric ID corresponding to the pre-state's hash value.

4. `getPostStateID()` obtains a unique numeric ID corresponding to the post-state's hash value. The first time it is called in a post-state, `getPostStateID()` also computes the hash of the post-state, and requests the unique numeric ID corresponding to the hash.

The numeric IDs in `getPreStateID()` and `getPostStateID()` are primarily used for generating user-readable labels, and mapped in a one-to-one manner to state hashes.

4.4.2 Producing Residual Block Labels

In §2.1.5, we noted that the residual program is a collection of basic blocks "stitched together" with `gotos`. That is, if a block b is specialized at state σ, producing residual block b_σ there needs to be a label that identifies b_σ as the residual version of b at σ. The state-management approach described in Chapter 3 ensures that every state has a single canonical representation: its hash. Thus a GenXGen generating extension can produce a numeric label[4] for every state.

Block-label generation is thus analogous to the version discussed in §2.1.5. For example, to produce the label for the residual version of block 1, we simply have the following emit statement:

```
emit("block_1_%d:", preStateID());
```

4.4.3 Handling Non-Control-Flow Statements

With the exception of lifting, the assignment statements are handled as described in §2.1.5. Static statements are placed verbatim in the basic block procedure. Dynamic statements are wrapped verbatim in an emit statement.

However, dynamic statements that contain static subexpressions that can be lifted into the residual code, like `*pat` in the condition `*pat != *s1`, are *not* parameterized on the static portion, unlike the generating extension

[4]We could use the 256-bit hash as-is. For readability reasons, every canonical state representation is instead granted a numeric label, starting from 0. The runtime maintains a map from hash to state label.

discussed in §2.1.5. That is, instead of `emit("%d != *s1;", *pat)`, the generating extension contains `emit("*pat != *s1;")`.

Lifting in GenXGen. GenXGen instead performs lifting at the reaching-definition level. When producing the IR used to construct the generating extension, CodeSurfer associates with every statement s the *may-use* set U_s, which contains the set of all memory locations that s might use—and for every memory location l in U_s, CodeSurfer provides a backwards edge from the statement to all statements s_l that (a) assign to l, and (b) have a path from s_l to s that does not contain an intervening assignment to l. Each static s_l is marked as "lifted" in GenXGen's binding-time analysis.

For each lifted statement s_l, GenXGen's gegen phase places the following into the generating extension:

1. a verbatim copy of s_l
2. a parameterized emit statement that emits residual code that sets l to the static value of l after executing s_l in the generating extension.

For example, consider the following code, where the boxed statement is dynamic, and the unboxed statement is static.

```
s = 10;
d = d + s;
```

The generating extension contains:

```
s = 10;
emit("s = %d", s);
emit("d = d + s");
```

This code is functionally identical to the equivalent parameterized emit statement one would obtain in classical lifting. This approach does have one subtlety due to the fact that the left-hand side of an assignment in C

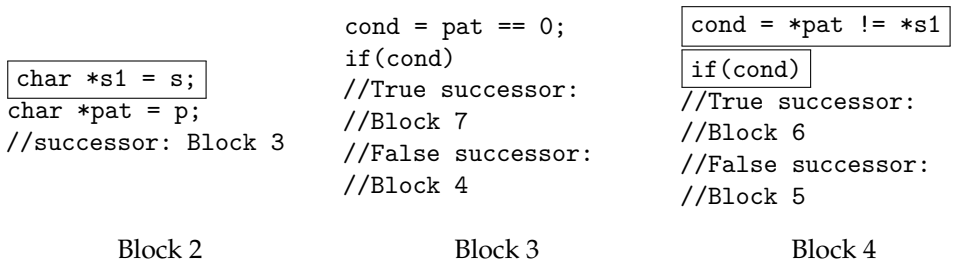


Figure 4.6: Blocks from Fig. 2.1.

may be an arbitrary expression. Thus, the left-hand side of the expression may also be computed from static values, and thus the left-hand side of the statement may need to be lifted as well. A full discussion is provided in Chapter 5, but for now it suffices to note that one can avoid infinite regress by using the observation that it is always legal to take the address of the left-hand side of a C assignment.

4.4.4 Handling End-of-Block Control-Flow

When a generating extension executes, handling end-of-block control-flow entails two disjoint but related actions: (1) updating the worklist, and (2) emitting residual code that produces the desired control-flow in the residual program.

Blocks Without a Final Control-Flow Statement, or With an Unconditional Control-Flow Statement. For a block that does not have a final control-flow statement, producing generating-extension code to handle control flow is straightforward. For issue (1), the ge-gen algorihm produces a worklist-update statement, which takes the successor block ID (which is known at ge-gen construction time), and the successor state id (known at specialization time—i.e., ge-gen execution time), and checks to see whether the state/block pair has been enqueued before. If it hasn't,

the pair is enqueued in the worklist. For example, for block 2 in Fig. 4.6, the following code is produced:

```
update_worklist(3, getPostStateID())
```

Moreover, as observed in §4.4.2, the state-hashing technique ensures that GenXGen can obtain canonical labels for all program states, and specifically, after executing all static statements in a block, the post-state ID can be produced. Thus, ge-gen produces a similarly simple block-final goto statement:

```
emit("goto block_3_%d;", getPostStateId());
```

Blocks With a Static Control-Flow Statement. For a block with a static control-flow statement, although the successor cannot be determined at ge-gen time, the generating extension will be able to determine the successor using the static state. Thus, ge-gen produces an `if` statement that checks the control-flow condition and selects the appropriate successor value. The worklist update and goto-emitter are produced in a manner similar to the unconditional case, except it is now parameterized on the statically successor block variable. For example, for block 3 in Fig. 4.6

```
cond = *pat == 0;
if(cond){
  static_successor = 7;
}else{
  static_successor = 4;
}
update_worklist(static_successor, getPostStateID());
emit("goto block_%d_%d;", static_successor, getPostStateID());
```

Blocks With a Dynamic Control-Flow Statement. As noted in Chapter 2, for a block with a dynamic control-flow statement, the generating extension must execute both successor blocks, and thus ge-gen must visit both successor blocks. Thus, to handle issue (1), ge-gen must produce two worklist updates. Moreover, to handle issue (2), code to emit an if statement must also be produced. However, the if statement must respect the basic-block structure of the residual program. Thus, the only action in each branch of the if statement must be a goto targeting the appropriate residual block. For example, given block 4 in Fig. 4.6, ge-gen places the following code in the generating extension:

```
update_worklist(6, getPostStateID());
update_worklist(5, getPostStateID());
emit("cond = *pat != *s1");
emit("if(cond){");
emit("  goto block_6_%d;", getPostStateID());
emit("}else{")
emit(" goto block_5_%d;", getPostStateID());
emit("}")
```

Blocks that End With a Procedure Call. Consider the following basic block for some block c in a program:

```
prod = s * d;
s++;
p(s, prod)
//callee block is some block n
//start of block c + 1
```

where p has formal parameters a and b.

There are two main ways of handling code generation for called procedures in a generating extension:

1. Inlining: One way of specializing procedures is to perform automatic inlining. That is, when a procedure is called during the execution of a generating extension, the procedure call can be removed and replaced with a jump to the specialized version of the first block.

2. Call a specialized procedure: Produce a call to a version of the callee procedure specialized with respect to the post state of the current block.

Both are acceptable approaches, but each poses its own set of challenges. In practice GenXGen[C] uses approach 2 and GenXGen[MC] uses approach 1. A full discussion of the implementation of each is deferred to Chapter 5, because there are several subtleties to each implementation. In particular, in both cases, the immediate textual successor of the block is not the successor during the execution of ge-gen. That is, the immediate textual successor to the pictured block c is block c + 1, but the immediate successor in terms of execution is block n, the first block of p.

If the generating extension finishes executing block c in state σ, then in approach 2 the generating extension will emit a call to $\texttt{p\_I}_\sigma$, where $\texttt{I}_\sigma$ is the numeric ID of post-state σ. However, in the emitted block text, the emitted call to p needs to be followed with a goto targeting the version(s) of block $c + 1$ specialized with respect to one or more post-states of the specialization of p, which cannot be known yet. I call this problem the *exit-splitting problem*, which will be covered in Chapter 5.

Conversely, if we were to use the inlining approach,[5] the generating extension will emit a `goto` targeting $\texttt{block\_n\_I}_\sigma$. However, consider what happens at some block m that is a return block for p. For p's return block, block m, there is no immediately available successor in the CFG that represents the successor to m. The return block for the call to p is encoded

[5]Ignoring various variable naming and scoping issues inherent to using this approach for C code.

in the calling context, and no successor can be produced at ge-gen-time; it can only be determined at ge-execution time.

This information must be made available to the generating extension in some way. The solution to this problem will also be covered in Chapter 5.

However, the callsite control can be handled in a straightforward way in either approach. For approach (1), ignoring variable renaming issues that are deferred to Chapter 5 (i.e., assuming here that procedure `p` is non-recursive, and that formal parameters `a` and `b` are globally unique names and can be globally defined), the following generating extension code is emitted:

```
emit("prod = s * d;")
s++;
a = s;
emit("b = prod")
update_worklist(n, getPostStateID());
emit("goto block_n_%d", getPostStateID());
```

Assume that, again, `a` is a globally unique name identifying a formal parameter of `p`, and it is correctly handled as a parameter to `p` in the static state. Then, for approach (2), everything is identical, except the final `emit`. Note that because the first argument is part of the static state, the residual version of `p` specialized with respect to σ only needs the value for parameter `b`:

```
emit("p_%d(b)");
```

4.5 Discussion

In Chapter 5, I discuss concrete implementation details of each of the utility macros, and subtleties relating to state representation and residual-

program construction.

In Chapter 6, I discuss the experimental evaluation of the implementation of specialization slicing. In particular, as part of RQ1, I investigate the time taken to perform specialization slicing. In practice, the specialization-slicing algorithm takes a negligible amount of time compared to the overall time taken by ge-gen. In particular, the construction of the program representation used for slicing dominates the actual specialization-slicing time. Because specialization slicing can produce an exponentially large number of result sets, and because BTA materialization converts slice results into a program, I also examine the size of the programs produced by materializing specialization slices. In practice, exponential blowup does not occur.

Chapter 5

Pragmatics

This chapter covers various practical concerns that must be addressed in implementing a generating extension-based specialization system, emphasizing aspects related to GenXGen's low-level, language-agnostic runtime, and process-based state representation. In §5.1, I discuss issues relating to implementing the generating-extension runtime. In §5.2, I discuss issues relating to the construction of generating extensions, as well as the main differences in constructing GenXGen[mc] generating extensions. §5.3 elaborates on the subtleties that must be addressed to generate residual code for procedure calls (and thus is related to §4.4.4).

5.1 Generating-Extension Runtime

In §5.1.1-§5.1.4, I describe how GenXGen state-representation processes implement IPC and residual-code-generation without contaminating the subject-state hash with the contents of memory related to these meta-tasks. §5.1.1 presents the meta-state isolation issue, and the mechanisms for passing information between the subject state and the wrapper context that maintains the meta-state. §5.1.2 discusses how the subject-program state is partitioned and isolated from the wrapper, so that CoW faults can correctly

be identified for the subject program. §5.1.3 describes the mechanism for switching between the subject-program state and the wrapper context. §5.1.4 discusses how information is passed between the subject-program and wrapper context.

The remainder of the chapter discusses other issues that arise in the implementation of the generating-extension runtime. §5.1.5 discusses the lazy symbolization technique used to lift pointers, and §5.1.6 explains how GenXGen[C] lifts arbitrary non-pointer data-types. §5.1.7 describes transition-compression, a technique that improves the performance of the generating extension and the residual program. §5.1.8 discusses the need to zero out stack frames when a procedure call returns, and explains how GenXGen performs this action.

5.1.1 State-Representation Process and Meta-State

In Chapter 3, the runtime environment for GenXGen generating extensions was described as a collection of state-representation processes managed by a single controller process. Recall that each block procedure in GenXGen performs, e.g., worklist and code-gen operations via a set of utility macros, as described in §4.4.1. Because the GenXGen runtime is a multi-process system, these operations require performing IPC between the state processes and the controller. In addition, the state process needs to perform various bookkeeping operations, and set-up and tear-down of various data structures in the process of being shepherded by the controller process.

Moreover, the GenXGen runtime identifies changed memory and updates by identfying CoW faults since fork. This task poses a challenge: each state process necessarily maintains important state that is *not* part of the subject program state. This *meta-state information* is part of of the same address space, but it must not be included in the state-hash. Thus, care must be taken to partition the address space so that only the static-state-relevant CoW faults are used to update the hash.

To avoid contamination of the hash with meta-state information, the state-representation process is partitioned into two contexts: the subject-program context, which represents the static state of the subject program, and the wrapper context, which is the meta-state that implements the utility macros and IPC needed to perform specialization. The binary for the state-representation process is implemented so that the memory regions are partitioned with no overlap—the subject program and wrapper context have their own global memory regions, stack, and heap. In effect, to achieve isolation, inside each state-representation process, the GenXGen runtime implements a special-purpose version of *green threading*,[1] where the threading library only ever manages two threads: one for the subject-program state, and one for the single wrapper program state that an individual state process represents.

5.1.2 Subject-Program State Isolation

Because every state process includes wrapper code, the address space must be partitioned to ensure that meta-state does not contaminate the hash of the subject-program state.

Specifically the subject program's stack, heap, and globals, must be placed in locations completely disjoint from the meta-state. Because we hash memory at the page granularity, each region must be page-aligned, and the size of the region must be a multiple of the system page size. Each region is allocated as follows:

Stack Isolation. To implement the subject program's stack, we `mmap` an appropriately sized, and non-file-backed, region into the address space, supplying the appropriate flags to ensure page-alignment. The mechanism for swapping between the subject program's stack and the meta-state stack is described in §5.1.3.

[1] User-space threads that do not use any OS facilities (see [Sung et al., 2002]).

Heap Isolation. We implement a subject-program version of `malloc` backed by a page-size-multiple sized and page-aligned `mmap` region, as with the stack. We chose to use our own allocator instead of, e.g., multiple `malloc` arenas because we need to inspect allocator state to support lifting of heap references, as described in §5.1.5

Global Isolation. To isolate the subject `.global` and `.bss` regions, we link the code for the subject-program representation in its own object file. When we link the state-process program, we use a custom linker script, which places the subject-program representation's globals in their own `.subject_globals` and `.subject_bss` regions, with the appropriate parameters set to ensure suitable sizes and alignments.

5.1.3 Switching Between Subject-Program State and Meta-State

Because the wrapper and the subject program have separate stacks and instruction pointers, the state-process program needs to be able to switch stacks and jump to a saved instruction pointer on demand. This requirement is similar to the functionality provided by `setjmp` and `longjmp`. However, in GNU libc, `setjmp` and `longjmp` are implemented as procedures that wrap architecture-specific assembly code. Thus, if either is called while executing in the subject-program context, some amount of subject-program-irrelevant information will be written to the subject program stack, violating our state isolation.

To prevent this contamination, we use inline assembly to implement a *pseudo-context-switch* macro that acts as a combined `setjmp` and `longjmp`.

1. Save all general purpose registers.
2. Save the stack pointer and base pointer.

3. Save the instruction pointer. Adjust it to point to the first instruction after the macro.
4. Restore all general purpose registers stashed during the previous pseudo-context-switch.
5. Restore the stack and base pointer stashed during the previous pseudo-context-switch.
6. Jump to the instruction pointer saved during the previous pseudo-context-switch.

This macro is crafted to avoid contaminating the subject stack with any meta-context-data, and only uses registers and global variables that are located in the meta-state, but are known to both the subject and wrapper code.

Note that because every basic block is implemented as a procedure in the GenXGen[mc], it suffices to only save the registers and stack context, and in fact, saving the instruction pointer would be incorrect, because the context switch would simply cause execution to jump to the exit point of the previous basic block executed by the state process. Thus, GenXGen[mc]'s version of the macro does not save and restore the instruction pointer.

5.1.4 Communication Between Subject-Program State and Wrapper State

To implement the utility macros described in §4.4.1, information needs to be passed between the subject-program state and the wrapper state. This section concentrates on the implementation of `emit`, because every other operation is a simpler instantiation of the same basic principles.

Recall that a state-representation process must avoid leaving any meta-state-relevant information in the subject-program stack. Thus, the macros must not invoke any procedure calls, and in particular `emit` cannot, e.g.,

invoke `printf` inside the subject-program context. Instead, the `emit` statement communicates necessary information from the subject-program state to the wrapper.

To appreciate when such communication is necessary, consider the following block, where the boxed instructions are dynamic, and all variable types are `char`:

```
dynamic--;
static1++;
static2++;
dynamic += static1 + static2;
```

Noting that static reaching definitions are lifted, the generating extension contains the following code:

```
emit("dynamic--;\n")
static1++;
emit("static1 = %d\n", static1);   // A lifting operation
static2++;
emit("static2 = %d\n", static2);   // A lifting operation
emit("dynamic += static1 + static2");
```

Only the two emits that perform lifting need to pass values back to the wrapper context. Moreover, other than the lifted values, the code emitted for a basic block is fixed, regardless of the state in which it executes, because basic blocks, by definition, contain no control flow. Thus, if the lifting-operation emits simply pass the associated values to the wrapper context, the remaining portion can be performed in the wrapper context.

The scheme sketched above is how code generation is implemented in GenXGen[C]. In effect, the print portion of emits—and by extension the entirety of emits with no lifted value—are no-ops that are deferred until after the pseudo-context-switch back to the wrapper context takes place.

To pass the lifted values to the wrapper context, a region of memory outside of the hashed subject program memory is reserved. The base of this region is stored in a variable known to both the subject and wrapper context, `lift_loc`. Because the number of lifts in a block are known at ge-gen time, each lift in a block can be assigned a specific offset in the region. For example:

```
static1++;
*(lift_loc + 0) = static1;
static2++;
*(lift_loc + 1) = static2;
```

After switching back to the wrapper context, the lifted values are simply inserted into a residual-block template and emitted:

```
printf("dynamic--;\n");
printf("static1 = %d\n", *(lift_loc + 0));
printf("static2 = %d\n", *(lift_loc + 1));
printf("dyamic += static1 + static2");
```

In general, whenever information, such as the numeric state-id, is passed between the subject and the wrapper contexts, a privileged, location outside hash-influencing memory is used. Moreover, to avoid excessive pseudo-context-switching, "heavyweight" operations that involve significant IPC, such as the worklist update, are implemented as deferred no-ops performed in the wrapper context, after executing the basic block.

5.1.5 Lifting Pointers

As described in §2.4, because GenXGen[C] performs a source-to-source transformation, static addresses cannot simply be used verbatim in the lifted program—there is no guaranteed way to preserve memory layouts

```
(1) s = &global_arr[0];
(2) s += 1;
(3) *s = d;
```
(a)

```
(1) s = malloc(10);
(2) s += 1;
(3) *s = d;
```
(b)

```
(1) s = &stack_arr[0];
(2) s += 1;
(3) *s = d;
```
(c)

Figure 5.1: The three classes of lifts.

between the original and residual program. Interpretation-based specialization approaches can simply treat all addresses as symbolic-base/offset pairs. However, because the GenXGen[C]'s representation of the static portion of the subject program is C code compiled to execute natively on hardware, without any instrumentation of pointer arithmetic, we do not have the ability to implement address arithmetic in this manner.

Instead GenXGen performs *lazy symbolization*. I make the simplifying assumption that programs specialized by GenXGen are sufficiently correct as to lack undefined behavior,[2] and that all lifted pointer values are a valid reference to some memory object, either on the stack, on the heap, or in the global regions. Thus, GenXGen[C] can simply introspect on the subject program's static state to identify which memory object a pointer references. More concretely, when constructing a generating extension, GenXGen[C] can examine debugging symbols, stack configuration, and memory-allocator state to obtain a "reverse map" from memory ranges to the corresponding symbols or allocated regions in the residual program.

Consider the three classes of lifts shown in Fig. 5.1. For all three cases, the lifting code placed in the generating extension for statement (3) is

```
base_t base_info = get_base(s);
record_base_decl(base_info);
printf("*(%s + %d) = d;\n", base_info->name, get_offset(base_info, s));
```

[2]Moreover, I assume that subject programs do not engage in any forms of type-punning or pointer forgery that violate surface-level assumptions about whether or not a given value is a pointer or non-pointer type.

The `get_base` procedure obtains information that uniquely identifies the variable or heap region referenced by `s`. The `name` field is the name of a global variable that, in the residual program, will contain the appropriate base address. It also contains the information necessary to compute the offset of `s` in the region or variable.

The `record_base_decl` procedure places a declaration of the global variable named by `base_info->name` in a special header, if the declaration has not yet been recorded.

The procedure `get_base` handles the pointer differently depending on whether it is a global, heap, or stack reference. It checks whether `s` is within the bounds of the subject's global, heap, or stack regions, and acts in the appropriate manner. As explained below, for global and stack references, the generating extension can obtain the base by using a combination of debugging symbols and program state; for heap variables, the generating extension uses a special `malloc` implementation to determine which allocated region the pointer falls into.

Identifying Global-Variable Bases

The generating extension's subject-wrapper process contains a lookup table initialized with the debug information for all of its global variables. The `get_offset` procedure simply searches the information to find the variable referenced by `s`, and produces the specification for the appropriate global variable. In the case of Fig. 5.1(a), we would obtain `char **_base_global_arr`, which would be initialized in the residual program with the base of `global_arr`.

Identifying Heap-Region Bases (Dynamically Allocated Storage)

In the case of a heap pointer, like the one lifted in Fig. 5.1(b), all calls to `malloc` in the subject program are replaced in the generating extension with calls to a special `subject_malloc` procedure. This procedure is a conventional memory allocator extended with additional bookkeeping that allows us to implement a procedure `lookup(s)`, which identifies the memory region within which `s` falls.

Each region is uniquely identified by the (program point, state) pair at which the region was allocated. The `base_info->name` field is uniquely determined by this pair. When the residual program begins execution, each malloc-region-base variable associated with the base of a lifted value is initialized by a call to `malloc`, to produce an appropriately sized region.

For example, in Fig. 5.1(b), if the program point in (1) is `point_1`, and the state is `0`, we will have a global region-base-variable named `_malloc_base__1_0`, which will be initialized in the residual program by a call to `malloc(10)` when the residual program starts execution.

Identifying Stack-Variable Bases (Local Variables)

Similar to the global-variable case, debugging information about symbols is used to obtain the variable associated with a given pointer. However, each process needs to track a small amount of additional information to allow the use of debugging information. Each state-wrapper process tracks the name of every procedure call on the call stack, the ID of the state at which each call occurred, and the frame-base pointer for each call on the stack.

When the generating extension performs the lift associated with statement (3) in Fig. 5.1(c), it scans the stored stack information for the current state, until it finds the stack frame that `s`'s value falls within, obtaining

both the name of the procedure P and the state S associated with the call to P. To determine which variable V with which s is associated, it then uses s's offset within the stack frame to search the debugging symbols for that procedure. It uses the (P, S, V) triple to produce the name of the global variable associated with the base of the reference to that "instance" of V in the residual program. For example, assume that in Fig. 5.1(c) one has P = p, and S = 1. The generating extension will emit "`*(_base_stack_arr_p_1 + 1) = d;`", where `_base_stack_arr_p_1` is a variable that holds the base of variable `stack_arr` from Fig. 5.1(c) in the version of p associated with state 1 in the residual program. Variable `_base_stack_arr_p_1` is initialized with the base address of `stack_arr` upon entry to the residual version of p associated with state 1.

5.1.6 Lifting Arbitrary C Types

A particular challenge specific to GenXGen[C] is the handling of compound data types, such as structures. For example, consider the following code, where `dyn` is marked as dynamic in the BTA:

```
struct s {
  int n;
  char c[2];
};
struct s v[3]
void p(){
  int dyn; /*dynamic*/
  struct s orig, lifted;
  orig.n = 1;
  orig.c[0];
  orig.c[1];
  lifted = orig; // This assignment is lifted.
```

```
  v[dyn] = lifted; // dynamic assignment
}
```

As described in §4.4.3, lifting is performed for all reaching definitions of a dynamic expression. Thus, the assignment `lifted = orig` is lifted. Recall that all lifted objects are transferred to the meta-state region by writing them to an unhashed location visible to the wrapper-context code. The region is essentially untyped, so for structs, a lift could be performed in the following manner:

```
*(struct s *)(lift_loc + orig_offset) = orig;
```

However, due to difficulties with CodeSurfer's AST representation (described in §8.2), obtaining the type specifier for the cast is challenging for compound data types. Moreover, the corresponding emit statement is also challenging to implement. C99 permits compound literals, so the following would work:

```
int  lifted_n = *(int *)(lift_loc + lifted_offset)
char lifted_c0 = *(char *)(lift_loc + lifted_offset + 4)
char lifted_c1 = *(char *)(lift_loc + lifted_offset + 5)
emit("lifted = (struct s){%d, %x, %x};\n", lifted_n ,
                              lifted_c0 , lifted_c1);
```

This operation also requires a cast, which, again runs up against the same limitations of CodeSurfer's AST representation.

CodeSurfer, however, does provide ready access to the size of all compound types in the program representation. To expedite the implementation of GenXGen[C], I chose to use this information and perform lifting at byte-level granularity. Thus, for example, instances of `struct s` are lifted as follows:

```
*(lift_loc + lifted_offset + 0) = *(((char *)lifted) + 0);
```

```
*(lift_loc + lifted_offset + 1) = *(((char *)lifted) + 1);
*(lift_loc + lifted_offset + 2) = *(((char *)lifted) + 2);
*(lift_loc + lifted_offset + 3) = *(((char *)lifted) + 3);
*(lift_loc + lifted_offset + 4) = *(((char *)lifted) + 4);
*(lift_loc + lifted_offset + 5) = *(((char *)lifted) + 5);
```

The corresponding emit statements are:

```
emit("*(((char *)lifted) + 0) = %x\n", *(lift_loc + lifted_offset + 0));
emit("*(((char *)lifted) + 1) = %x\n", *(lift_loc + lifted_offset + 1));
emit("*(((char *)lifted) + 2) = %x\n", *(lift_loc + lifted_offset + 2));
emit("*(((char *)lifted) + 3) = %x\n", *(lift_loc + lifted_offset + 3));
emit("*(((char *)lifted) + 4) = %x\n", *(lift_loc + lifted_offset + 4));
emit("*(((char *)lifted) + 5) = %x\n", *(lift_loc + lifted_offset + 5));
```

In GenXGen[C], all lifting, even for scalar types is performed at byte granularity. This approach produces correct code; however, as seen in §6.3.4, the approach can have detrimental effects on a residual program's performance. Thus, in future versions of GenXGen[C], it would be desirable to find a way to rectify the issues with CodeSurefer's AST representation and implement the compound-literal lifting scheme.

5.1.7 Jump Compression

One of the primary advantages of partial evaluation is the elimination of branches controlled by static predicates. For example, consider the specialization of the following code:

```
if(static_cond){
  dyn++;
}else{
  dyn--;
}
```

```
dyn *= 2;
```

If `static_cond` is true when the block is evaluated (and the worklist is empty other than the entries for the specialization of the pictured code) the emitted code will have this structure, with the if statement eliminated:

```
block_m_s1:
dyn++; //The true branch
goto block_n_s2;
block_n_s2:
dyn *= 2;
```

However, this code still contains a spurious goto, because the target label immediately follows the goto. In practice, residual code often contains long chains of these goto/target pairs; when a program has large runs static basic blocks, this situation often results in sections of the residual program that consist solely of chained jumps with no intervening static code.

To eliminate these chains of gotos, GenXGen[C] performs *online jump compression* [Jones et al., 1993]. Jump compression (also known as *transition compression* in the partial-evaluation literature), can be performed either *online* or *offline*, depending on whether it is done as part of the partial-evaluation process, or as a transformation of the residual program. I chose to implement the online jump-compression method used in C-MixII[Makholm, 1999], because this technique has the added advantage of significantly improving the time taken to specialize a program.

The key idea of C-MixII's jump-compression technique is that at every block with a statically-known successor, the generating extension need not enqueue a state/block pair in the worklist. In fact, the generating extension can simply continue producing residual blocks until a dynamic condition is reached. Only upon reaching a dynamic branch does the generating extension halt specialization and enqueue the necessary block/state pairs.

The potential downside of this approach is that there may be some amount of duplicated code and work. For example if in two different unrollings of a loop with a single block in its body, specialization reaches static state `s` after the first iteration, jump compression will cause the loop to be unrolled completely, and the duplicate state will not be recognized until the first dynamic control after the unrolled loop. In practice, such duplication is uncommon, and the advantage in improving specialization time and performance of the residual program outweigh the small amount of code duplication that may occur.[3]

5.1.8 Stack Zeroing

Due to the use of full-memory hashing, care must be taken to ensure generating extensions converge in a reasonable amount of time in the presence of procedure calls. When procedures return, data that is no longer semantically relevant is left on the stack. Consider specializing the code in Fig. 5.2 when `d` is dynamic. Assume that all stack memory below[4] the stack frame for `do_loop` prior to loop entry is zero.

Because the loop is controlled by the dynamic parameter `d`, the generating extension will repeatedly evaluate the loop body until the state exploration converges. The if statement is also governed by `d`, and so the generating extension will evaluate all three branches of the if, each of which calls one of `p`, `q`, and `q` with a statically known parameter. Note the call relationship of the three procedures: `p` calls `q`, which calls `r`, and assume `p`, `q`, and `r` have identically sized stack frames. Then, after statically executing the three branches in the loop body, we have performed

[3]Because of internal implementation details not discussed here, it is currently not easy to deactivate this jump compression, and no formal evaluation was performed in Chapter 6 to compare specialization time with and without jump compression, but in my work developing GenXGen, I observed substantial improvements in specialization times and residual-program execution times using this technique.

[4]The x86 stack grows downwards.

```
int d2 = 0;//make global;
void do_loop(int d){}
  while(d > 0){                        p(int d,int i){
     if(d % 3 == 0){                     q(i);
       //branch 1                      }
       r(1)
     }else if(d % 3 == 1){             q(i){
       //branch 2                        q(i);
       q(2);                           }
     }else{
       //branch 3                      r(i){
       p(3)                              d2 += i;
     }                                 }
     d--;
  }
```

Figure 5.2: Without stack zeroing, specialization of procedure `do_loop` explores many semantically redundant states.

all possible specializations in the loop body—the static parameters are hard-coded.

However, the generating extension ends up enqueueing the three distinct post-states shown in Fig. 5.3, none of which have been seen before (because the stack below `do_loop's` call frame was all zeros).

Thus, the generating extension must specialize the loop body again, with respect to all three post-states. Consider specializing the body with respect to the state resulting from specializing branch 1. This post state contains the three stack frames for `p`, `q`, and `r` with their parameters set to 3. Branch 1 has never been specialized with respect to this state, and thus, it is not until the generating extension returns from the call to `p` that the generating extension encounters a (state, block) successor that has been seen before.

Worse, consider the post states for branches 2 and 3 in Fig. 5.4. The loop body has never been visited in these post-states either, and thus all three paths through the loop body must be specialized yet again with respect to each of these. Doing some scratch work, it can be seen that in

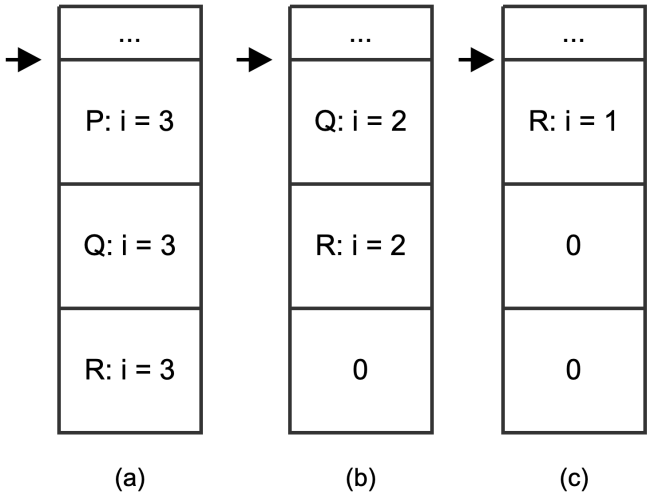


Figure 5.3: The post-states after executing branches 1, 2, and 3, respectively. The arrows denote the stack-frame pointers after returning from the respective procedure calls in each branch. The x86 stack grows downward, so the contents of memory below the pointer are no longer valid.

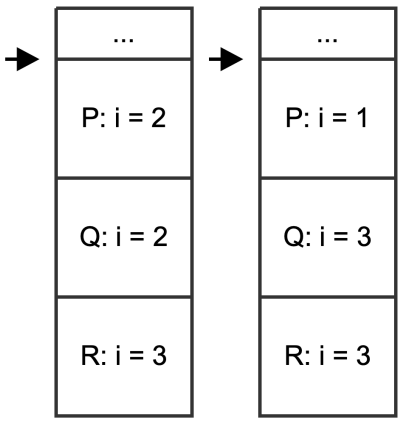


Figure 5.4: The post-states enqueued after specializing the loop body with respect to Fig. 5.3(a). The arrows denote the stack-frame pointer after returning from the procedure calls in each branch. The x86 stack grows downward, so the contents of memory below he pointer are no longer valid.

total there will be 21 specializations of loop branches, seven times as many as are necessary.

The problem is that despite not being semantically relevant (referencing the contents of a stack frame after returning is undefined behavior in ANSI C) old stack frames are still incorporated into the state hash. If, on return from the calls in each path, the generating extension had zeroed out everything below the `do_loop` stack frame, the post-state of each path would have been identical to the state at loop entry.

Thus, we extend our generating-extension infrastructure to zero out all memory below the address referenced by the stack pointer `rsp` after a procedure returns. We implement this by tracking `rsp` in the meta-state, along with the identity of all return blocks. Upon returning from a procedure, the range between the old and new `rsp` is zeroed out.

Because our representation of the static portion of the subject program is compiled at `-O0`, it is safe to zero out that portion of memory, as long as the use of the "red zone" is disabled.[5]

5.2 Ge-Gen

This section covers issues related to constructing generating extensions for real-world C programs built using tools such as `make` (§5.2.1), and also describes how GenXGen[mc] produces generating extensions for x86 binaries, emphasizing the ways that the binaries for generating extensions created by GenXGen[mc] and GenXGen[C] differ (§5.2.2).

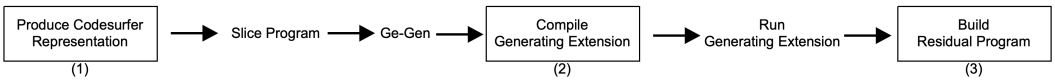


Figure 5.5: The sequence of steps for specializing a program with GenXGen[C]. The boxed items, (1), (2), and (3) require the ability to replay the build of a C project.

5.2.1 Build Tracing for C Generating Extensions

One primary goal in our implementation of GenXGen[C] was the ability to specialize non-trivial projects in, as much as possible, a turn-key manner. Specializing a subject program using generating extensions requires producing three key artifacts, the structure of each of which is intimately related to the structure of the subject program. Fig. 5.5 shows the steps that GenXGen[C] goes through when specializing a C program. The boxed items (1), (2), and (3) are the steps that build the key artifacts. Artifact (1) is the CFG and SDG representation of the subject program, which is used to perform the slicing-based BTA, and to perform the ge-gen phase. Artifact (2) is the generating extension itself, which is a transformed version of the static portion of the subject program, intermingled with program-specialization code. The generating extension executes, producing residual code in the form of transformed versions of the original subject program's source files. To produce the residual program binary, these files must be compiled in a manner consistent with the build process for the original program.

Performing step (1) requires identifying all source files used to compile[6] the subject program. Because the generating extension's subject representation contains, e.g., all global variables and procedures relevant

[5]By default, the System V AMD64 ABI reserves the 128 bytes below the stack frame as scratch space, particularly for leaf functions. This feature can be disabled in GCC via the `-mno-red-zone` flag [gcc, 2019, §3.1].

[6]Note that I am not, in general, concerned with the binary that is built for the original subject program *per se*; as will be seen, to goal is to capture the *process* by which it is built.

to the execution of the static portion of the subject program, the compilation in step (2) needs to be done in a manner that respects the overall source-code structure of the subject program. For the generating-extension source code produced by GenXGen[C], the simplest way of achieving this goal is just replaying the compilation of the subject program, with the original subject-program source files replaced by corresponding source files in which each original procedure is replaced by appropriate basic-block procedures for its constituent basic blocks (à la Figs. 2.4 and 2.5). In step (3), the residual source code consists of transformed versions of the source-code files from the original program; thus, step (3) also requires replicating the original program's build process, with the original subject program's source-code files replaced with the residual source-code files.

In GenXGen[C], performing step (1) is also carried out by replicating the build process of the subject program. CodeSurfer, the tool used to implement slicing and ge-gen, produces the CFG and SDG of complex projects by performing a *hook build*. A hook build is a CodeSurfer feature: when invoked in hook-build mode, and given a command that constructs a C program, such as a single compiler invocation, or a bash script, or a makefile, CodeSurfer traces every compiler invocation, and collects a list of all source files used to produce the resulting program. CodeSurfer parses these files to construct its program representations.

In practice, large C programs built to run on Linux, such as the BusyBox applets used to evaluate GenXGen[C], are built using a build-automation tool such as *make*. The build for a large project generally consists of a series of compilation commands, such as `gcc -b -o f.o f.c`, where, rather than producing a standalone executable binary, `gcc` is invoked with the `-b` flag to produce a static object file (here named `f.o`), which is linked into the final program. Individual object files are sometimes linked together into *archive* files, which themselves are not stand-alone programs, but larger compilations of static objects intended to be linked into a final binary.

Thus, the final build command in a large project is generally a (nominal[7]) compiler invocation, `gcc -o p f.o g.o h.a`, which produces the output program[8] (here named `p`, the item after the `-o` flag) by linking together object and archive files into an executable.

Thus, the builds for large C projects tend to consist of a partially ordered set of dependencies. Build-automation tools, like `make`, take a declarative list of objects to produce, along with rules to produce them, and the objects those rules depend on. These systems automatically resolve dependencies and produce the appropriate output program.

Prior classical C specializers, such as C-MixII[Makholm, 1999] cannot handle these builds in a turn-key fashion. To operate on programs that are built using `make`, a user of C-MixII must provide three different manually edited makefiles for, respectively, steps (1), (2), and (3) from Fig. 5.5. A user of C-MixII manually selects source files of interest within a program's build process to specialize, and alters the original program's makefile accordingly [Makholm, 1999].

Existing classical specializers, such as LLPE, that handle multi-file projects more organically generally operate at the IR level [Smowton, 2014]. For example, LLPE interposes on the compilation of software projects, and extracts LLVM IR, and specializes the LLVM IR using an interpreter-based specializer. The residual program is emitted in LLVM IR form, and the emitted IR is used to produce the residual program. Because the goal with GenXGen[C] was to implement a classical Mix-style source-to-source partial evaluator, an approach similar to LLPE was not an acceptable solution

Instead, I chose to design GenXGen[C] with an emphasis on being able to correctly parse and transform large multi-file projects. Specifically,

[7] In practice, a compiler command such as `gcc` is actually a wrapper that provides a unified interface to a system's linker, loader, and assembler, and the appropriate program is invoked based on the flags passed to `gcc`. Here `gcc` actually invokes the linker.

[8] Note that the lack of a -b flag here causes `gcc` to produce an executable.

my goal was to be able to produce generating extensions for projects that contain hundreds of source files, such as BusyBox. Performing this task correctly requires performing a program build that, for steps (2) and (3) reproduces the build process, but with various source files substituted with either generating-extension or residual-program source code, and for step (1) creates the SDG and CFG for the correct set of source files of the subject program. In practice, one cannot simply invoke `make` on, e.g., an automatically transformed version of the subject program to perform steps (2) or (3), and moreover, for step (1) performing a CodeSurfer hook build on complex projects frequently causes spurious information to be included in CodeSurfer's representation of the subject program. Instead, GenXGen performs *build tracing*, which traces an invocation of the subject program's build command, and produces the sequence of commands that produce the subject program.

To see why build tracing is necessary, first we consider (1): producing the CodeSurfer SDG and CFG for the subject-program.

Recall that makefiles define a partially-ordered collection of actions that are performed to build the output program. These actions need not merely consist of compiler and linker invocations. In the builds of many large programs, such as BusyBox and GNU Coreutills[9], other ancillary C programs are compiled, and various scripts are run to produce, e.g., header files derived from various host-system configuration settings. For example, during the compilation of a BusyBox applet, several ancillary C are compiled, such as a small program that parses portions of an applet's source code to produce files containing documentation about a program's usage. These ancillary C programs are essentially disposable parts of the build process that are only used during compilation of the target program.

This approach to program construction is problematic, because

[9]Even in the case where a single BusyBox applet or program from the GNU Coreutils collection are built, these ancillary C programs are still built, as they are considered a necessary component of the build process..

CodeSurfer's hook-build feature, when used to trace the evaluation of a makefile that builds multiple distinct programs, does not differentiate among the programs being built: CodeSurfer's hook build mode cannot determine which compiler invocations produce objects relevant to the final target program of a makefile is, and hence which source files are truly relevant. Codesurfer simply traces all C-compiler invocations, recording all source files used, and incorporating them into its SDG and CFG. Thus, the representations of multiple disjoint programs are merged into a single program representation. Thus, the resulting SDG and CFG represents a single "conglomerate" program with multiple distinct entry points named `main`.[10] Thus, manual intervention is required to select the appropriate entry point and build a generating extension in many cases. Even disregarding the need for manual intervention, CodeSurfer projects are quite large, over a gigabyte for BusyBox applets, and the incorporation of spurious programs into the SDG and CFG structure constitutes needless bloat.

In addition, running a complex project's makefile to perform steps (2) or (3) does not work correctly. BusyBox's makefile also executes several ancillary scripts to parse the source code and produce several internal tables that are used to construct various header files. In practice, these scripts are not compatible with specialization: if they are executed on specialized code, they produce incorrect tables, and the residual program's build will fail. Moreover, from the view of GenXGen[C], the programmatically generated files are already part of the program's source code, having been incorporated into CodeSurfer's SDG that is used to perform ge-gen, and thus are part of the subject-program representation. The files will be emitted when the generating extension produces residual code, and consequently, they do not need to be generated again.

Because of these issues, GenXGen needs to collect an executable script

[10]The representations of `main` are given numerical suffixes to disambiguate the representation, but manual intervention is required to select the correct one to produce the generating extension.

that produces the desired binary without producing or running unnecessary binaries and scripts. One potential approach would simply be invoking `make` with the `-t` flag and extracting all necessary compiler and linker invocations, and discarding unnecessary ones. However, the `-t` flag does not capture the behavior of subsidiary, non-`make` scripts.

Instead, I implemented a `make`-tracing-and-filtering script. Before GenXGen[C] is run, the trace script invokes the project's makefile, which produces the binary as usual.[11] The trace script attaches to the build via `strace`. `Strace` is used to record every call to `exec` by `make` and its descendants. From this trace, every `exec` call that launches `gcc`, `ar`,[12] or `ld` is extracted, along with their arguments.

After the trace is collected, the invocations are used to construct a DAG. An invocation A has an edge to another invocation B if and only if A uses B's output. For example, A = `gcc -o p f.o g.o`, which produces `p` by linking together `f.o` and `g.o`, would have an edge to B = `gcc -b -o f.o f.c`, which produces object file `f.o` from `f.c`, because A depends on the output of B.[13] Then, all items in the DAG that are not reachable from the target binary are removed from the trace. This approach ensures that the trace contains only the invocations needed to produce the target binary. The resulting trace is suitable for use with CodeSurfer's hook-build mode (to accomplish step (1)), and is also used to compile the generating extension (step (2)) and to crate the residual binary (step (3)).

[11] It is important that `make` be invoked on the subject-program makefile exactly once. All programatically-generated source code needs to be produced, or we cannot construct a correct generating extension.

[12] `ar` is an archival program that bundles multiple .o files into a single archive that can be passed to the linker at a later time

[13] Recall that gcc produces an object file, and not a binary file, when invoked with `-b`.

5.2.2 Constructing Machine-Code Generating Extensions

In this section, I describe how GenXGen[mc] produces generating extensions for x86 machine code. The core ideas and architecture are essentially identical to that of GenXGen[C], with only slight changes in implementation. The utility macros from §4.4.1 are generally implemented as *pseudo-instruction* macros that are functionally identical to the C equivalents. In general they function as described in §5.1.4, with values communicated to the wrapper state via privileged, unhashed memory locations, and "heavy-weight" operations such as emitting code and communicating worklist updates to the controller being deferred until the end-of-subject-block pseudo-context switch back to the wrapper.

Thus, the description of generating extensions for x86 code presented in this section focuses on the salient differences between GenXGen[C] and GenXGen[mc], which are primarily related to lifting. The key difference between GenXGen[C] and GenXGen[mc] (besides the obvious difference in languages) is that GenXGen[mc] processes *stripped* binaries; i.e., binaries with all debugging symbols removed. That is, GenXGen[mc] does not expect to have access to fine-grained symbol or type information.

Like GenXGen[C], GenXGen[mc] performs BTA using slicing[14] —in this case, slicing as supported by CodeSurfer/x86 [Balakrishnan et al., 2005b; Anderson et al., 2003; codesurfer, 2018]. CodeSurfer/x86 can recover information about variable-like locations, as well as information about structural properties for compound data-types such as structures through an analysis known as Value Set Analysis (VSA) (see §8.2 for a discussion of the implementation of VSA and its limitations). Thus, it is generally possible to to identify variables to slice from, and to identify a division between pointer and non-pointer types for the values held by variables and CPU registers at a given program point.

[14]However, the initial implementation of GenXGen[mc] predates the use of specialization slicing in GenXGen[C].

```
L3:  mov dl, [ebx]      -- dereference pat
     cmp dl, 0          -- check first if condition
     jz L7              -- if(*pat == 0) return 1
L4:  mov cl, [eax]      -- dereference s1
     cmp cl, dl         -- check second if condition
     jne L2             -- if(*pat != *s1) break;
L5:  incr eax           -- s1++
     incr ebx           -- pat++
     jmp L3             -- while(1)
```

Figure 5.6: Body of the naive string matcher's inner loop. Boxed instructions are dynamic, double-boxed instructions have their destination operands lifted, and the remainder are static.

The partitioning between static and dynamic occurs at the instruction level. Thus, a standard forward slice is computed to determine the set of dynamic instructions. To identify lifted values, reaching-definitions analysis is performed for the operands of each instruction I in the dynamic set. Any static instructions that define an operand used by I must have associated code to lift the value of the operand.

In Fig. 5.6, BTA results are illustrated for the code that implements the innermost loop of `match` from Fig. 1.1.[15] Register `eax` contains the address of the current offset in the string that is being searched, and `ebx` contains the address of the current offset in the pattern to be matched. The registers `cl` and `dl` contain the current characters in the string and

[15]The 32-bit Intel x86 instruction set (also called IA32) has six 32-bit general-purpose registers (`eax`, `ebx`, `ecx`, `edx`, `esi`, and `edi`), plus two additional registers: `ebp`, the frame pointer, and `esp`, the stack pointer. In Intel assembly syntax, which is used in the examples in this paper, the movement of data is from right to left (e.g., `mov eax,ecx` sets the value of `eax` to the value of `ecx`). Arithmetic and logical instructions are primarily operand instructions (e.g., `add eax,ecx` performs `eax := eax + ecx`). An operand in square brackets denotes a dereference (e.g., if `a` is a local variable stored at offset -16, `mov [ebp-16],ecx` performs `a := ecx`). Branching is carried out according to the values of condition codes ("flags") set by an earlier instruction. For instance, to branch to `L1` when `eax` and `ebx` are equal, one performs `cmp eax,ebx`, which sets `ZF` (the zero flag) to 1 iff `eax` $-$ `ebx` $= 0$. At a subsequent jump instruction `jz L1`, control is transferred to `L1` if `ZF` $= 1$; otherwise, control falls through.

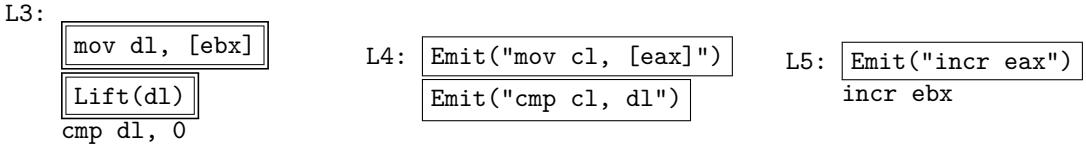


Figure 5.7: The machine-code generating extension block-procedures produced for the code in Fig. 5.6, with label-generation, jump-generation, and worklist-manipulation pseudo-instructions elided.

pattern, respectively. Basic blocks `L3`, `L4`, and `L5` correspond to the inner loop blocks in Fig. 1.1. `L7` is reached only if a match is found. `L2` is the target of the inner loop's break statement, which starts another iteration of the outer loop.

In lifted instruction `mov dl, [ebx]`, register `dl` must be lifted, because `cmp cl, dl` in block `L4` compares the static pattern character in `dl` to a character from the dynamic string.

Given the partitioning of instructions in Fig. 5.6, the generating-extension-generator emits x86 code augmented with *pseudo-instruction* macros analogous to their C counterparts. We elide the state snapshotting as well as jump and label-generation code, because it is analogous to the C equivalents.

Fig. 5.7 illustrates the machine-code generating extension produced from `match`.

The lifted instruction I = `mov dl, [ebx]` is executed, just like a static instruction. After I is executed, `Lift` emits code that sets the value of `dl` in the residual program to the value that `dl` holds immediately after the execution of I.

Note that the approach taken in GenXGen[mc] differs from that used in GenXGen[C] in a significant way: in the example above (for GenXGen[mc]), what is lifted is a register, and consequently a memory access is eliminated. In contrast, in GenXGen[C], the entire branch condition for the comparison of a character in the pattern with a character in the string

being searched in the C program is tagged as dynamic:

```
if(*pat != *s1) goto 1;
```

The definition of `pat` that reaches the branch condition is the increment statement `pat++;`, so that statement (as a whole) is what is lifted. When compiled, the unoptimized[16] residual program would contain a load from memory (to obtain the value of `pat`).

Even if the value of `*pat` were stored in an intermediate variable, e.g.,

```
char c = *pat;
if(c != *s1) goto 1;
```

the behavior of GenXGen[C] would be the same: when compiled, the (unoptimized) residual program still contains a load from memory—in this case, to obtain the value of `c`.

As these examples illustrate, GenXGen[mc] is capable of more fine-grained lifting than GenXGen[C]. That is, at the machine-code level, because a source-code statement is typically broken into a sequence of finer-granularity instructions, reaching-definition-based lifting can improve the performance of residual programs in ways that are not available to GenXGen[C].

Lifting Pointers. Like the `Lift` macro used in GenXGen[C], the implementation of the `Lift` macro in GenXGen[mc] needs to handle correctly pointers to stack and heap objects. However, GenXGen[mc] processes stripped binaries and produces a residual program in machine-code form. This situation differs from the one we face with in a significant way: GenXGen[mc] is not performing a source-to-source transformation that needs to preserve source-level symbols. Thus, GenXGen[mc] does not need to

[16] In our experimental evaluation in Chapter 6, to avoid conflating the effects of GCC's optimization with the effects of GenXGen, the original and residual programs are compiled at `-O0`.

perform the symbol-preserving lift techniques described in §5.1.5. Instead, GenXGen[mc] performs a more light-weight form of lazy symbolization in a manner analogous to the eager symbolization performed by WiPEr [Srinivasan and Reps, 2015].

For every lifted operand, I assume CodeSurfer/x86's value set analysis can reliably[17] recover information about whether or not a given operand holds a pointer type. Making the simplifying assumption that the subject program being specialized is correct, I assume that all lifted pointers reference a valid memory region. At a coarse-grained level, such a pointer either references the heap, the stack, or a global variable. Thus, every pointer can be treated as an offset into a heap object, the stack, or the heap, simply by examining the address to see which region it falls into.

Because GenXGen produces machine-code, I have sufficiently fine-grained control over the residual code that I can ensure that when the residual program is assembled and linked, all global references are still valid in the residual program.

Heap references are handled identically to GenXGen[C], as described in §5.1.5. Each heap address is checked using the data held by the special purpose `malloc` implementation; the base/offset pair is recorded; and the residual code is produced in the same way as the C version.

For stack references, I take the same approach as WiPEr, except that the symbolization is performed lazily. When generating residual code, WiPEr preserves the stack layout between the original and residual program. Thus, WiPEr can treat all references to the stack as a symbolic `stack_base+offset` pair. Then, when the residual program is executed, the lifted value in the residual program is recomputed by adding the residual program's stack base to the recorded offset.

Thus, when a stack pointer is lifted in GenXGen[mc], the offset into the stack is recorded, and code in the residual program is emitted to obtain

[17]The VSA implementation reliably identified whether a memory location or register holds a pointer to the stack or heap at a given program point in all of the programs tested.

```
int g = 0;
void f(int s, int d){
prod = s * d;
  s++;
  p(s, prod); //end of block 1.
  printf("%d", g); //end of block 2
}
```

```
void p(int a, int b){
   g++; //block 3
  if(b > 0){
     g = a; //block 4
   }else{
     g = a + 1; //block 5
  }
     return; //block 6
}
```

Figure 5.8: Code that illustrates several subtle issues with code generation for procedure calls. Note that the printf statement in block 2 is tagged as dynamic even though its arguments are static, so that an occurrence of the printf statement will appear in the residual program.

the residual program stack base and add it to the offset.

5.3 Handling Procedure Calls

In §4.4.4, I noted that there are two ways of emitting residual code for calls to a procedure p. In particular, a specializer can (i) inline the specialized version of p, eliminating the call, or (ii) emit a specialized version of p, and replace the original call with a call to the specialized version. I noted that both approaches require solving several subtle code-generation issues.

To appreciate the subtleties, consider specializing procedure `f` in Fig. 5.8 with respect to the initial state `s = 1`. Note that the first basic block in `f` consists of all code up to and including the call to p, and that the call to p is the end-of-block control construct for block 1.

Regardless of whether we choose to inline the specialized version of p, or emit a call to a specialized version of p, a peculiar non-locality issue arises. The immediate textual successor of the callsite that calls p in the residual program will be block containing `printf("%d", g)`. However, the immediately-executed successor in the specialization of the program in Fig. 5.8 is block 3, the first block of p. In fact, block 2 will not be reached

```
int g = 0;
void f(int s, int d){
prod = s * d;
  p_1(prod); //end of block 1.
  //What goes here?
}
```

```
void p_1(int b){
  if(b > 0){
     goto block_4_2;
   }else{
     goto block_5_2;
   }
block_4_2:
  goto block_6_3;
block_5_2:
  goto block_6_4;
block_6_3:
   //What goes here?
block_6_4:
   //What goes here?
}
```

Figure 5.9: Incomplete code for the specialization of `f` on `s = 1`.

at all until the return statement in block 6 has been specialized, and its successor enqueued.

This situation poses several problems. Consider the case where GenX-Gen[C] emits a call to a specialized version of `p`, say, `p_1`. Assume that the state at the end of block 1 has state id 1. Then, the next block to execute is block 3, the first block of `p`. The variable `g` is incremented, so the successor state has id 2. Block 3's successor is dynamically determined, so both block 4 and block 5 must be specialized on state 2, yielding empty residual blocks for both. Moreover, because `g` has different values at the end of block 4 and block 5, their post-states differ, and thus block 4 ends in post-state 3, while block 5 ends in post-state 4. Thus, block 6, the return block for `p` must have two residual versions, one specialized on state 3 and one specialized on state 4.

This situation leaves us with a problem: how is the return from `p_1` handled in the residual program? The static states at the return differ; in particular, block 2 contains a residual `printf` whose output is governed by the static value `g`, and `g` differs in the two post-states of the call to `p_1`.

Thus, somehow, after the call to `p_1`, some code needs to be added that can select the appropriate post-state block to execute. Moreover, somehow, `p_1`, at each return block, needs to communicate which static post-state state the return block corresponds to. This problem is referred to as the *exit-splitting problem*[Bodík et al., 1997]. As pictured in Fig. 5.9, this poses a challenge because a procedure returns to a single point, control needs to flow to two different control points, depending on the return.

To solve the exit-splitting problem, we augment the residual program with a global variable, `state_tag`, which contains the static post-state id at the return site. Thus, `blocks_6_3` and `block_6_4` are emitted as:

```
block_6_3:
  state_tag = 3;
  return;
block_6_4:
  state_tag = 4;
  return;
```

Each call to a specialized procedure in the residual program is followed with specialized exit-splitting code. For example, each call to `p_1` would be followed by code that dispatches control to the appropriate residual version of block 2:

```
p_1(s);
switch (state_tag){
 case 3:
  goto block_2_3;
 case 4:
  goto block_2_4;
}
```

Characterized more generally, due to a dynamic branch in a called procedure, there can be multiple static return states from a call-site associated

with a given static state. However, in classical partial evaluation, every residual program point is associated with a *single* static state. Thus, a residual procedure called at a given callsite can return to one of several different program points in the caller. This situation is non-standard, because in a conventional program, every call-site has exactly one return point (typically the location following the call instruction). Thus, call-return linkages in the residual program need *exit splitting*[Bodík et al., 1997], where control can be returned to one of several return points in a caller. In the residual programs created by GenXGen[C], this is done by having each exit from a procedure pass, by means of a global variable, an exit number denoting the static state associated with the residual procedure's exit point. At the corresponding (standard) control-site in the caller, a switch statement dispatches control based on the exit number to the appropriate point in the residual caller for the matching static state.

In addition, there is a more subtle problem that occurs in both the inlining and specialized-call approaches. At the end of block 6, a successor block needs to be identified. But block 6 is a *return*, and there may be many callsites for a procedure. In all other cases, the set of successors is known from a combination of immediately available syntactic information from the subject program in combination with static state that is visible in program variables—thus, it is relatively straightforward to produce generating-extension code that enqueues the correct state/block pair (see §4.4.4). However, the block that a procedure call returns to is encoded in the *call stack*, which is not (within the bounds of most language standards) explicitly available in the program text.[18]

Thus, for both procedure-specialization approaches, we need to make call-stack information available to the residual-code-emission procedures. Specifically, GenXGen maintains, for each state, in the wrapper context (i.e., outside of hashed memory), a shadow-stack that contains the block to

[18]Even if it were, mapping from code-section-address to the basic-block ID produced from the CodeSurfer representation of the subject program is, in practice, a difficult task

return to for each procedure call. This approach allows the next successor to be enqueued correctly, and also in the case where the residual procedure is inlined, to determine the goto target for the jumps that replace the returns in the inlined residual procedure.

Chapter 6

Experimental Evaluation

The promise of generating extensions is their potential to specialize real-world programs, by optimizing performance through simplification and unrolling, and debloating programs by removing unreachable code. Thus, there are two main concerns: first, given GenXGen's new OS-assisted state-management techniques, are the generating extensions produced by GenXGen capable of specializing real-world programs in a timely manner, and second, does GenXGen perform reasonable debloating and optimization in practice? Thus, we evaluate GenXGen on a set of microbenchmarks and real-world programs.

The majority of the experiments, covered in §6.1-§6.3, focus on GenXGen[C]. GenXGen[mc], for x86 machine code, was the earliest iteration of GenXGen, but does not implement all of the features described in this thesis; of particular importance, GenXGen[mc] does not use specialization-slicing for its BTA, only regular forward-dependence slicing. Moreover, for reasons described in more detail in Chapter 8, various limitations on CodeSurfer's x86 library models and static binary-analysis tools made it significantly more difficult to obtain slices of useful precision on real-world programs. Thus, I discuss GenXGen[mc]'s experiments separately from those for GenXGen[C].

§6.1 explains the six research questions aimed at addressing these concerns. §6.2 explains the design of the experiments, and the specific hardware and software platform used for the experiments. §6.3 discusses the evaluation for each experimental question, and gives an in-depth discussion of the results. §6.4 discusses the evaluation of GenXGen[mc].

6.1 Research Questions

The goal of the state-management and lifting techniques presented in §3 is to enable GenXGen[C] to specialize real-world programs that use stack, heap, and global memory. A partial evaluator should be able to specialize non-trivial programs in a "reasonable" amount of time, while also producing meaningful reductions in program size, as well as residual programs that are faster than the original. To evaluate our tool with respect to these criteria, our experiments were designed to answer the following research questions:

Research Questions

- **RQ1.** How long does it take to produce a generating extension? How much does specialization slicing contribute to the time?
- **RQ2.** What increase in program size occurs from slice materialization (§4.3) when binding-time analysis is performed via specialization slicing?
- **RQ3.** What are the execution-time characteristics of a generating extension produced by GenXGen[C]?
- **RQ4.** How does specialization affect the run-time performance of the residual program? In programs for which it is relevant, how does the difference in run-time performance change as the size of the dynamic input increases?
- **RQ5.** How does specialization affect the size of the residual program?
- **RQ6.** How much code does specialization remove from a program?

RQ1. *How long does it take to produce a generating extension? How much does specialization slicing contribute to this time?*

Rationale. Our base slicing-based BTA relies on the use of CodeSurfer to perform, e.g., points-to-analysis and reaching-definitions analysis to construct an SDG representation of the subject program. The phase of BTA during which specialization slicing is carried out (to sidestep the parameter-mismatch problem) requires non-trivial automata-theoretic operations, and in the worst case the output program size is exponential in one of the parameters that characterizes the size of the input program.

Moreover, given these slicing results, producing the generating ex-

tension requires a non-trivial source-to-source transformation. Thus, it is reasonable to investigate the time to produce a generating extension, particularly in terms of the individual tasks that comprise the process.

Metrics. To evaluate this cost, we track the amount of time required to produce a generating extension for every BusyBox applet in our evaluation suite. We break these times down by the seven phases of the generating-extension generator.

The specialization process is an eight-phase process:

1. The recording of compiler and linker invocations in the program's makefile. This process produces a repeatable script that can be used in later ge-gen phases.

2. The C-preprocessor expansion of all source files identified in (1), to simplify the source-to-source transformation in later phases.

3. The construction of the CodeSurfer representation of the program. This phase constructs the PDG and SDG used for specialization slicing, as well as internal AST representations used for source-to-source rewriting.

4. Invocation of the specialization-slicing algorithm itself to perform binding-time analysis.

5. An additional code-rewriting pass based on the specialization-slicing results, to create variants of each procedure identified by specialization slicing. This pass is necessary due to limitations in CodeSurfer's internal program-rewriting facilities.

6. Build a new CodeSurfer representation of the rewritten program.

7. The source-to-source transformation that produces the generating extension's source code.

8. Compilation of the generating extension.

RQ2. *What increase in program size occurs from slice materialization (§4.3) when binding-time analysis is performed via specialization slicing?*

Rationale. GenXGen[C]'s binding-time analysis uses specialization slicing to sidestep the parameter-mismatch problem that can occur with conventional slicing. However, specialization slicing produces programs that are, in the worst case, exponential in one of the parameters that characterizes the size of the original program—namely, the maximum number of formal-in or formal-out vertices in any procedure's PDG.[1] Such blowups would carry over from binding-time analysis to the generating extension: each procedure variant found during specialization slicing would yield a different variant of the procedure in the generating extension. Moreover, before the generating extension is created, the slice-materialization step (§4.3) creates an elaborated version of the program that, in general, contains multiple replicas of the program's procedures—i.e., a replica of each procedure q for each different binding-time pattern of q's formal-in vertices. If such blowups occur in practice on the real-world programs we specialize, it could unacceptably increase the time required to construct generating extensions.

Metrics. As described in §4.3, GenXGen[C]'s use of specialization slicing adds a full copy of procedure P for every unique result set for P in the slice. Thus, the most important metric for assessing the code-size effects—particularly the presence or absence of exponential blow-up—is simply the number of procedures before and after specialization slicing. However, even in the absence of exponential growth in code size, specialization slicing might, e.g., produce an unacceptable number of copies of very large procedures. Therefore, we also include other code-size metrics, such as lines of code and number of basic blocks. In principle,

[1] Recall from §2.2.1 that formal-in vertices capture the passing of parameters from caller to callee when a procedure is invoked (including the "passing" of global variables as a kind of extended set of parameters), and formal-out vertices capture the passing of return values (including globals) from callee to caller when the procedure returns.

it would have been natural to compare specialization-slicing-based BTA with BTA based on Binkley's reslicing algorithm. However, on the Busybox programs Binkley's Algorithm generally produced overly conservative results that admitted no useful specification, largely due to global variables such as `errno` being included in the slice.

RQ3. *What are the execution-time characteristics of a generating extension produced by GenXGen[C]?*

Rationale. Generating extensions created by GenXGen[C] execute in a complex runtime environment, as described in §3 and §4.4. Thus, it is reasonable to investigate the performance of the generating extensions produced by GenXGen[C] on microbenchmarks and real-world programs.

Metrics. For each program, we record the execution time of the generating extension on a single representative static input. We also record the average time taken to perform a hash-update operation in each program's generating extension.

RQ4. *How does specialization affect the run-time performance of the residual program? In programs for which it is relevant, how does the difference in run-time performance change as the size of the dynamic input increases?*

Rationale. Program specialization is useful to the extent that it can "improve" the program in some observable way. Partial evaluation can eliminate the computation of values known at specialization time, allowing for, e.g., loop-unrolling and elimination of large parts of the program. Such transformations can speed up the residual program. On the other hand, transformations such as full unrolling can have negative effects on instruction-cache locality, and the reaching-definition-based lifting can introduce unnecessary lifts into the program, both of which can have negative effects on performance. To determine how much improvement (or degradation) occurs in practice, we compare the performance of the original and residual programs.

Moreover, for many programs, the size of the input that is tagged as *dy-*

namic is not necessarily constant—e.g., the dynamic input could be a file of arbitrary size. Thus, it is natural to inquire if, given a program specialized on a fixed static input s, whether the difference in performance between the original and residual program grows as the size of the dynamic input increases. Thus, we also investigate how the execution times of the original and residual programs scale as a function of the size of the dynamic input.

Metrics. For each program, we measure the execution time of the original and residual program for a single representative static and dynamic input pair. For a fixed *static* input, we measure how the performance scales with respect to changes in the size of the dynamic input.

RQ5. *How does specialization affect the size of the residual program?*

Rationale. The other way partial evaluation can improve programs is by *decreasing* program size by eliminating code that is impossible to reach, given a specific static input. On the other hand, many of the potential speed-ups available through specialization come at the cost of increasing the size of various parts of the program. Thus, we measure the effect of program specialization on program size.

Metrics. For each program, we record the change in size between the original and specialized versions.

RQ6. *How much code does specialization remove from a program?*

Rationale. There is a subtlety to the size-change metrics discussed in RQ5. Program specialization can be used to remove features. For a given static input, if code corresponding to a given feature is unreachable, that feature will not be present in the residual program. However, the specialization produced for a feature that is present can involve, e.g., loop unrolling, which creates multiple copies of one or more basic blocks. Consequently, a residual program can have fewer features than the original program, despite being larger than the original. Thus, we also investigate the amount of code from the original program that has been completely eliminated from the subject program.

Metrics. To determine what code has been removed, we record the percentage of blocks from the original program that do not occur (in any specialized form)[2] in the residual program.

6.2 Experimental Setup

We evaluated GenXGen[C] using six microbenchmarks, an additional seventh benchmark for evaluating the worst-case (blow-up) effect of specialization slicing, and ten BusyBox applets. These programs are described below.

Four of the microbenchmarks were previously used to evaluate WiPER [Srinivasan and Reps, 2015]. The fifth, `str_match`, is the $O(|s||p|)$ string matcher given in Fig. 1.1. The sixth is another string matcher, specially structured to produce the Knuth-Morris-Pratt string matcher, as described in [Consel and Danvy, 1989].

Microbenchmarks.

- `power` computes x^n
- `dotproduct` computes the dotproduct of two n-dimensional vectors.
- `interpreter` is an interpreter for the minimalist language "Brainf*ck"
- `filter` applies a convolutional filter to an image.
- `str_match` is an $O(|s||p|)$ substring matcher, where `s` is the subject string, and `p` is the pattern string being searched for.

[2]There is one subtlety about the concept of "does not occur." Consider the case where a generating extension executes a block that consists entirely of static code. The code from the block will not occur overtly in the residual program. However, because the block affects static program state, the block's effects may be materialized in the residual program at some other location via lifting. Thus, for our purposes "does not occur in any specialized form" means "never visited by the generating extension."

- `KMP` is an $O(|s||p|)$ substring matcher structured such that the specialized version's residual code implicitly encodes the finite automaton produced in the Knuth-Morris-Pratt substring-matching algorithm. The residual program's runtime is $O(|s|)$.

Microbenchmark Specialization. For `power`, the static input is the exponent, leaving the base as the dynamic input. For `dotproduct`, the static input is the set of coefficients of one vector, as well as the dimension of both vectors, n. The other vector is dynamic. For `Interpreter`, the static input is the source code of a Brainf*ck program. For `filter`, the static inputs are a parameter that selects one of four fixed $m \times m$ filter matrices, and n, which specifies the width and height of the $n \times n$ image the chosen filter is applied to. The dynamic input is the array containing the image pixels. The static input for both `str_match` and `KMP` is the pattern string, and the dynamic input is the subject string.

Specialization-Slicing Exponential-Case Microbenchmarks. To evaluate the worst-case effect of specialization-slicing, we use an additional family of programs that produce the worst-case exponential blow-up in program size when specialization slicing is performed. We separate these programs from the other microbenchmarks because they have no semantically meaningful utility; they are merely programs that are structured to induce the worst-case slice result.

Each program P_n in the worst-case family contains a single `main` procedure and n auxiliary procedures, which have a nested calling relationship. That is, each P_i calls P_{i-1}, and the exponential blowup is a function of nesting depth n. Specialization slicing yields a program with 2^{n+1} procedures.

```
unsigned int g1, g2;
void R0(){
  t1 = g1;
  g1 = t1;
}

void R1(){
 v = 2;
 if(v > 0){
   g1 = v;
   P0();
 }else{
   P0();
 }
}

int main(){
    g1 = 1;
    R1();
    return 0;
}
```

$R0_0 = \emptyset$

$R0_1 = \{\texttt{R0.g1.formal\_in}, \texttt{t1} = \texttt{g1}, \texttt{g1} = \texttt{t1}\}$

(a)

```
unsigned int g1, g2;
void R0(){
  t1 = g1; t2 = g2;
  g1 = t1; t2 = g2;
}

 void R1(){
  v = 2;
  if(v > 0){
    g1 = v;
    R0();
  }else{
    R0();
  }
 }

 void R2(){
  v = 2;
  if(v > 0){
    g2 = v;
    R1();
  }else{
    R1();
  }
 }

int main(){
    g1 = 1; g2 = 2;
    R2();
    return 0;
}
```

$R0_0 = \emptyset$

$R0_1 = \{\texttt{R0.g1.formal\_in}, \texttt{t1} = \texttt{g1}, \texttt{g1} = \texttt{t1}\}$

$R0_2 = \{\texttt{R0.g2.formal\_in}, \texttt{t2} = \texttt{g2}, \texttt{g2} = \texttt{t1}\}$

$R0_3 = \{\texttt{R0.g1.formal\_in}, \texttt{t1} = \texttt{g1}, \texttt{g1} = \texttt{t1},$
$\texttt{R0.g2.formal\_in}, \texttt{t2} = \texttt{g2}, \texttt{g2} = \texttt{t1}\}$

(b)

Figure 6.1: The first two programs (a) P_1 and (b) P_2 in the worst-case-slice program family, along with the sets of slice results for the different variants of procedure R_0 (at the bottom of the call hierarchy) produced by specialization slicing. The boxed lines denote the source statements for the specialization slice.

Fig. 6.1 shows the first two programs in the family. The key idea underpinning the exponential blowup in P_n is that the program is structured such that the slice results for the assignments to g1 through gn in R0 encode the binary representation of every integer from 0 to $2^n - 1$. Stated differently, the presence or absence of gi's assignments in R0 encodes the

```
R2(){
  v = 2;
  if(v > 0){
    g2 = v;
    R1_0(0)
  }else{
    R1_1(0);
  }
}
```

```
R1_0(){
  v = 2;
  if(v > 0){
    g1 = v;
    R0_00();
  }else{
    R0_01();
  }
}
```

```
R_00(){
  t1 = g1; t2 = g2;
  g1 = t1; t2 = g2;
}
```

Figure 6.2: Three example procedures from the materialization of the specialization slice of P2 in Fig. 6.1

i^{th} bit of an n-bit binary number.

Each `Ri` contains a single `if` statement where, in one path, `gi` is assigned a value not in the slice, thus ensuring that there will be slice results for all `gj`, $j < i$, that do not have `gj` in the results. Conversely, the other path does not contain an assignment to `gi`, so there must also be slice results for all `gj`, $j < 1$, that do contain `gi`. Thus, the size of the polyvariant result set is exponential in the size of P_n. Moreover, this structure also ensures that every Rj has 2^{n-j} slice results. Thus, the materialized slice results contain 2^{n-j} copies of each R_j. In total, with the inclusion of main, the results contain $1 + \sum_{i=0}^{n} i = 2^{n+1}$ procedures.

Discussion of the Materialized Results and Their Specialization. We include the worst-case microbenchmarks to illustrate the worst-case blow-up issue with specialization slicing, and to see whether there are similar effects in the sizes of the materialized slice results obtained in practice. However, we do not produce generating extensions for these results—and we do not include them in any other experiment—because semantically each P_n, in and of itself, is not very interesting; the programs exist solely to force the "binary encoding" in the slice results, which induces the blow-up.

Moreover, the specialization of the materialized slice results is straightforward. Consider the two procedures from the materialization of the

results for P2 pictured in Fig. 6.2. The dynamic "inputs" are the assignments to g1 and g2 (in main, which is not shown). The local variables v are necessarily static "inputs," because they are used to kill the dynamic slice variables and induce the appropriate control flow that yields the exponential blow-up. Because every local v is static, the specialization simply follows a single chain of calls: R2, to R1_0, to R0_00, yielding a program that contains only those procedures and main. Because the variables v *must* be static, every specialization of the materialized slice result for P_n yields a program with $n + 2$ procedures. Thus, we do not consider the specialization of the materialized slice results for P_n further.

BusyBox Applets. We use ten applets from BusyBox, which perform the following computations:

- yes repeatedly prints its argument to stdout (y if no argument). There is no dynamic input.
- base64 encodes/decodes a base64 string depending on whether a flag is absent or -d, respectively.
- dos2unix converts line endings of a file read from stdin from DOS-to-Unix convention or Unix-to-DOS convention, depending on whether the -u or -d flag is given, respectively.
- cat prints the standard input to standard output. Cat provides additional behavior, such as numbering lines, depending on what flags are supplied.
- env takes a list of assignments to environment variables and then invokes a specified program. The flag -i clears all other existing environment variables before invoking the specified program.

- `shuf` computes random shuffles based on specified input. The -i flag takes a numeric range `m-n` as an argument and produces a shuffled list of all numbers from `m` to `n`.
- `od` is a binary-dump program that supports a variety of output formats. The -x and -X flags are standard hex dump formats, with 16 bytes per line divided into two or four byte clusters, and -b dumps the input as octal bytes. The -e flag dumps the input as 64-bit IEEE floating-point numbers. The flags can be combined arbitrarily; in this case, each "chunk" of the input stream is printed multiple times, once in each format.
- `tr` "translates" a character stream based on the provided parameters and character sets. When given a pair of character sets, such as `ab cd`, `tr` maps the `n`th character in the first set to the `n`th character in the second. The -d and -s flags delete and squeeze a specified character set, respectively, where "squeezing" is the deletion of all but one instance of a character `c` in every contiguous run of a repretition of `c`.
- `cut` removes specified items from each line of its input. The -b flag takes a comma-separated list of integers, and removes the n^{th} character in a line unless `n` is one of the positions specified in the list.
- `xxd` is a binary-dump program that also supports patching. `xxd` provides a more limited set of dump-formatting options than `od`, always producing hexadecimal output. The `-g` flag specifies the size of bytes per group in a line, and the `-c` flag specifies the number of bytes per line. The `-r` flag supports patching, however, we did not with respect to this capability.

BusyBox applet Specialization. `dos2unix`, `cat`, and `base64` represent *feature-removal tasks*, in which a single operation mode is chosen from a set

of potential modes, and the code for other modes is removed.

`yes`, `shuf`, and `env` are *in-lining-and-unrolling tasks*, unrolling the inner loop of the program, and inlining specific actions based on the specified flags.

The programs `od`, `tr`, `cut`, and `xxd` have an *interpreter-like structure* and combine elements of both feature-removal and in-lining-and-unrolling tasks. In particular, the main loop of the program iterates over the dynamic input stream, and the inner loop iterates over a static structure specifying a set of operations to be performed on the stream. For example, the two hex-dump programs `od` and `xxd` take their command-line arguments, and construct a linked list of structures containing, e.g., format specifiers for printing "chunks" of the input stream in the specified format(s).

Many Unix command-line utilities can be run as filters, typically documented as follows:

> < `command` > [`OPTION`]...[`FILE`]...
> ...
> `With no FILE, or when FILE is -, read standard input.`

Consider such a program P, where the static parameter set S is {`argc`, `argv`}. Suppose that P's generating extension $ge_{P,S}$ is supplied with an assignment $A(S)$ in which the `argv` value either (i) contains zero or more options and no file name, or (ii) contains zero or more options followed by '-'. The specialized program $P_{A(S)}$ produced by GenXGen[C] is a filter program that receives input from `stdin` and behaves according to the options that were specified in the static `argv` value. In our experiments, the application of GenXGen[C] to `dos2unix`, `cat`, and `base64` carries out such a specialization; e.g., the specialization of "`base64 -d -`" creates a decoding command-line filter.

`yes`, `base64`, `dos2unix`,`cat`, `env`, `shuf`, `od`, and `tr` are BusyBox analogs of coreutils command-line utilities that we isolated from BusyBox prior to

specialization. The `xxd` command was also isolated from BusyBox, but is an analog of a Vim component.

BusyBox is a single binary that provides the functionality of coreutils and other standard Unix utility packages in a single binary. Ordinarily, the top-level `main` procedure of BusyBox dispatches into the entry procedure of a given applet based on flags passed to BusyBox (or the binary name). However, for each experiment, we created a modified version of the BusyBox source that only enters the desired applet.

6.2.1 Scaling Experiments

To answer the second part of RQ4—*how does the difference in run-time performance change as the length of the dynamic input increases?*—we select the relevant subset of the subject programs for which GenXGen[C] can specialize the program with respect to a static input and then accept an arbitrarily large dynamic input. For each of these programs P, we select a static input s and specialize P, yielding P_s. Given P and P_s, we run the pair of them on dynamic inputs of various sizes, and compare their performance. (For each of the dynamic inputs, P also receives s as the value of the static input.)

The seven programs for which we can perform the scaling experiment are `base64`, `dos2unix`, `cat`, `od`, `tr`, `xxd`, `str_match`, and `KMP`. All of these programs take as static input their command-line arguments—the arguments used for each experiment are shown in Fig. 6.10—and take `stdin` as their dynamic input which can be scaled arbitrarily.

For these programs, we scale the size of the input from 5,000 bytes to 95,000 bytes in 5,000-byte increments. For each size, we perform the 100-trial experiment described in §6.2.2.

6.2.2 Experiment Timing

We timed the end-to-end execution time of each program on an input, collecting the 10% trimmed mean of 100 executions: i.e., we ran the program 100 times, and discarded the 10 shortest and 10 longest execution times. To time the programs, we instrumented the beginning and end of `main` in each program with calls to a `rdtscp`-based timer. `rdtscp` is an x86 instruction that returns the time since boot as the number of clock cycles *in terms of the CPU's base frequency* (i.e., the rate that timestamps increase is invariant, and not tied to changes in the processor's operating frequency due to, e.g., power-saving or thermal factors). In practice, `rdtscp` provides ~40-clock-cycle resolution [Paolini, 2010].

Evaluation platform. We used a VMWare Fusion 13.0.1 VM on a MacOS 12.1 computer with 16 GB of RAM, and a 2.5 GHz, 4-core, Intel Core i7 CPU (model 4870HQ). The guest OS is Fedora Linux 31, running the 5.3.7 Linux kernel, and the VM is allocated two cores and 8 GB of RAM. The times collected via `rdtscp`, as described above, are not virtualized: they sample the host OS timestamp counter. Thus, times may include host-OS context-switching overhead. To mitigate this effect, the results for the end-to-end execution of the original and residual programs are collected when the host and guest OS have a minimal process workload. When done this way, and when 10% trimmed means are used, the experiments are repeatable, and fine-grained timing information can be extracted from the results.

6.3 Evaluation

This section discusses the evaluation of each of the experimental questions, and presents our findings.

6.3.1 Answer to RQ1: How long does it take to produce a generating extension? How much does specialization slicing contribute to the time?

Fig. 6.3 and Fig. 6.4 show the time required for each phase of GenXGen[C]'s generating-extension generator. The plot in Fig. 6.4 is the time required to produce a generating extension from specialization results, and corresponds to the top bar in the stack plot in Fig. 6.3

For the BusyBox applets, tracing compiler and linker invocations, expending source code, and producing the Codesurefer program representation dominates the time taken, largely owing to the complexity of BusyBox's build system, and the large number of compiler invocations performed in each of these steps.

As shown in Fig. 6.4, for the BusyBox applets, the amount of time required to perform both BTA via specialization slicing and generate the generating-extension source code is comparatively negligible. The worst-case complexity of specialization slicing does not occur in these (real-world) programs, and the automata-theoretic operations performed during specialization slicing comprise a negligible proportion of the overall ge-gen time. The longest specialization-slicing pass was for `cut`, which required 17 seconds.

For the microbenchmark experiments, we do not perform specialization-slicing. No programs in the microbenchmark suite have polyvariant BTA results, and the specialization-sliced and forward-dependence slice results are identical.

For all six of the microbenchmark programs, ge-gen takes less than two seconds, nearly half of which is taken up by the construction of the generating extension.

Specialization Slicing. The worst-case-blowup experiment, the timings for which are shown in Fig. 6.6, contrasts sharply with the BusyBox exper-

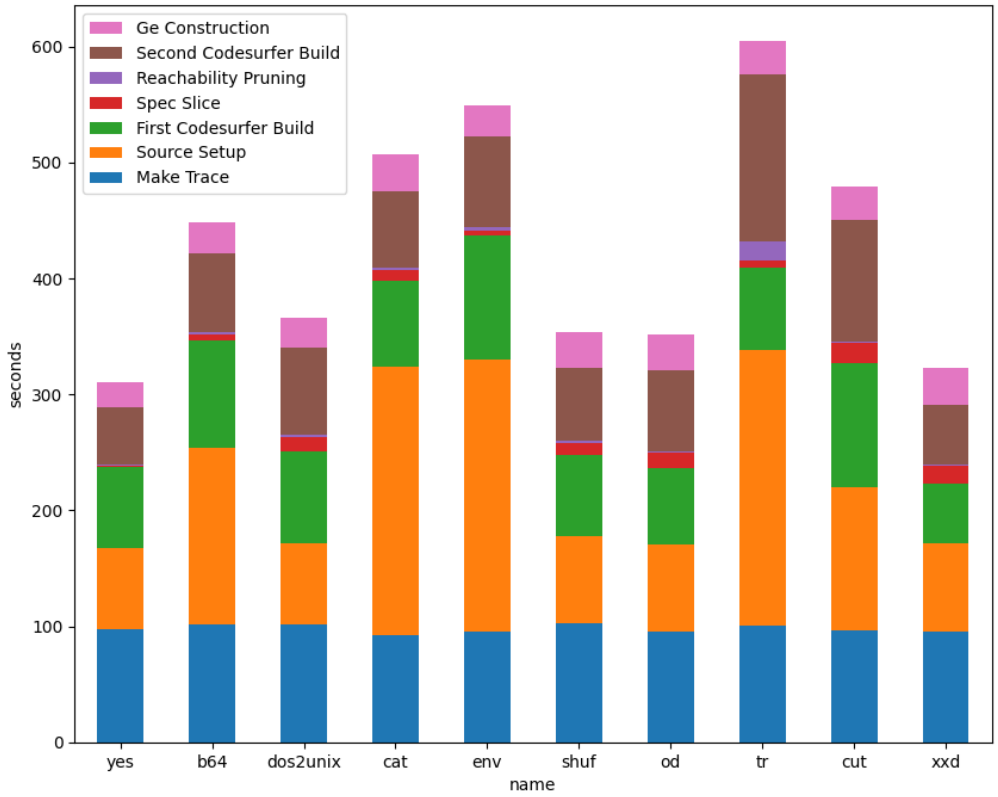


Figure 6.3: Time to perform a full ge-gen on the BusyBox applets. The colored regions break the time down into the eight phases described in RQ1 in §6.1: The bottom bar is the make tracing step (step 1). The orange bar is the preprocessor expansion (step 2). The green bar is construction of the SDG and PDG (step 3). The red bars are for specialization slicing (step 4). The purple bar is reachability pruning (step 5). The brown bar builds a new CodeSurfer project for the materialized slice (step 6). Steps 7 and 8 are folded into a single light pink bar, and the times for those two steps are shown in Fig. 6.4.

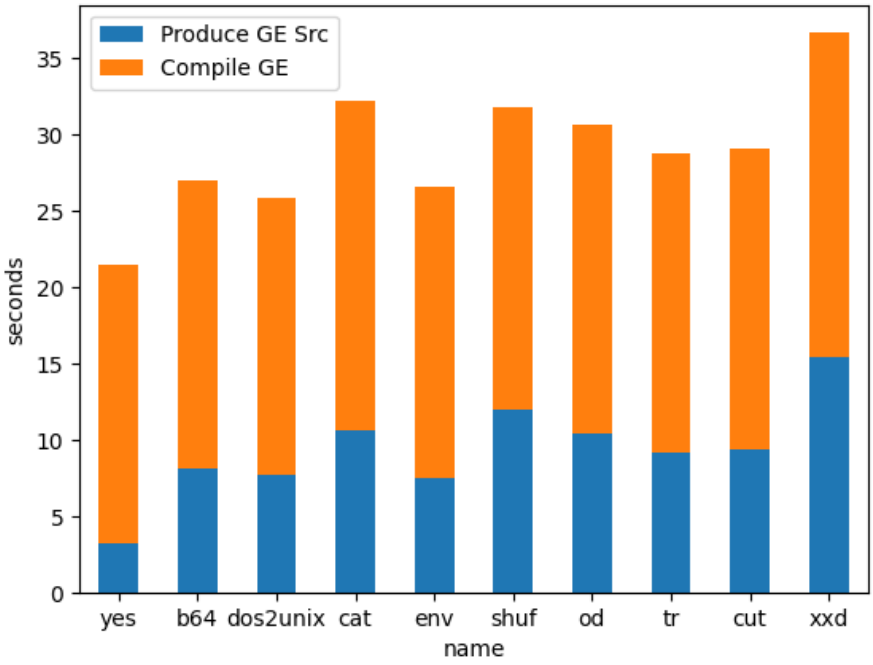


Figure 6.4: Time to produce generating extensions for the BusyBox applets from specialization-slicing results.

iments; a ten-procedure program required 45 seconds for a specialization slice; over twice as long as the 17 seconds for `cut`, a 13-procedure program. Moreover the exponential trend is immediately clear from the plot.

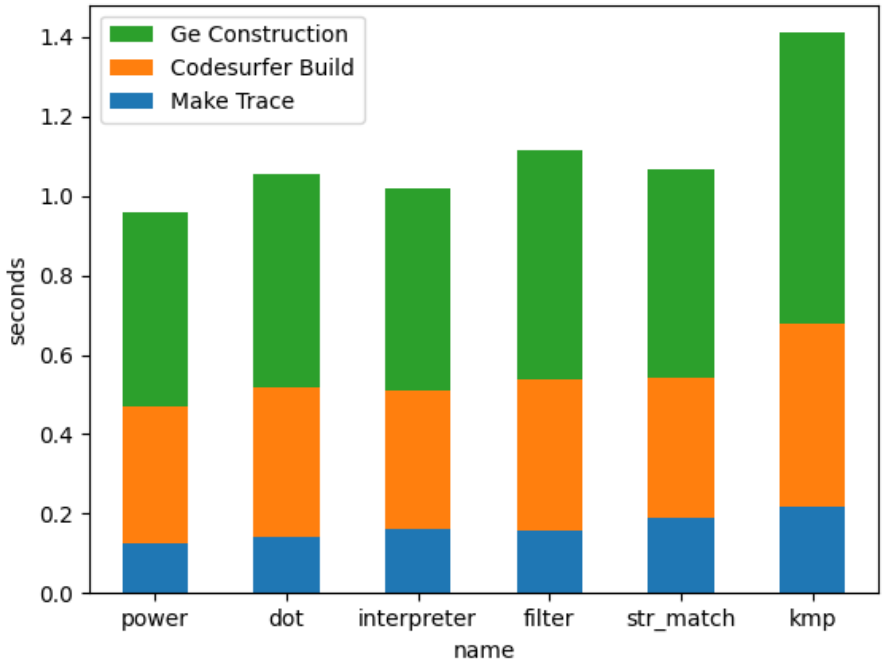


Figure 6.5: Time to produce generating extensions for the microbench programs.

Findings

The time required to produce a generating extension is largely due to the time required to trace compiler and linker invocations, as well as the time required to produce an SDG. The time for traversing the SDG (e.g., BTA), generating source code for the generating extension, and compiling the generating extension contribute comparatively little to the overall time.

For the BusyBox applets, the specialization-slicing algorithm contributes little to the overall ge-gen time, and the performance does not exhibit the characteristic time explosion seen in the micro-benchmark that provokes worst-case behavior for specialization slicing.

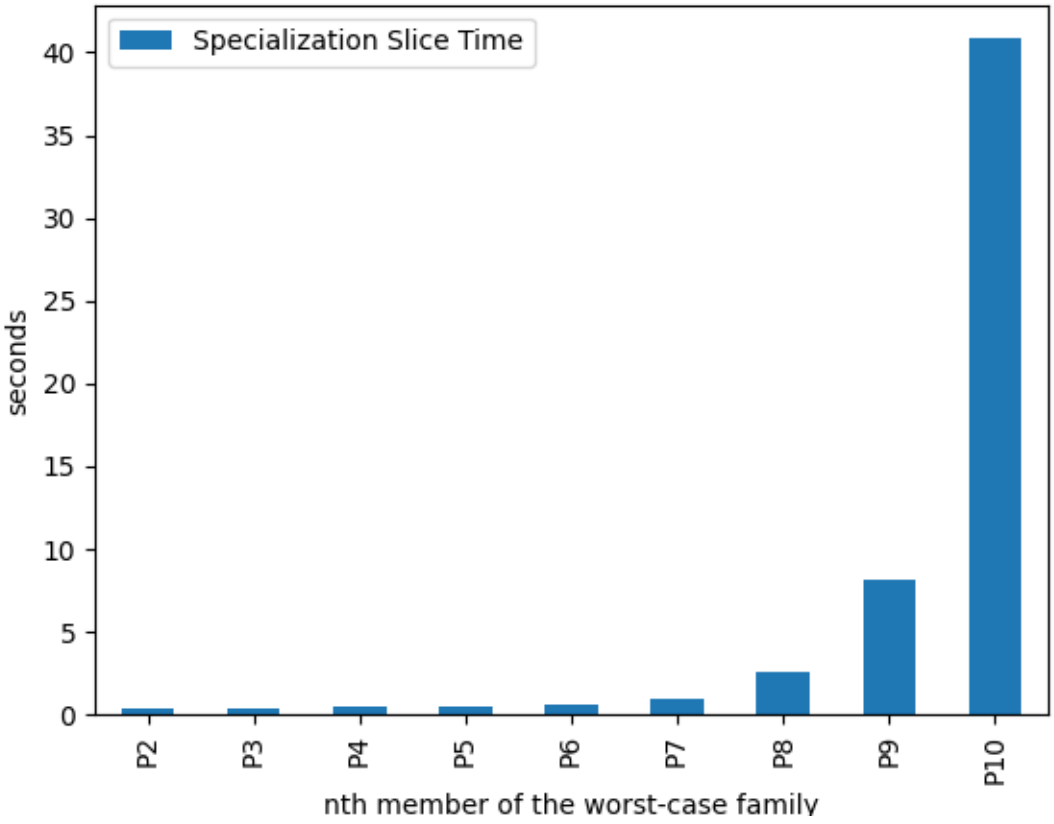


Figure 6.6: Timing results for the worst-case microbenchmark family.

6.3.2 Answer to RQ2: What increase in program size occurs from slice materialization when binding-time analysis is performed via specialization slicing?

Fig. 6.7 shows the effects of specialization slicing on program size from slice materialization (§4.3). The overall code-size increase caused by slice materialization was modest, amounting to 31% in the worst case for `tr`. In the worst case for procedure duplication—`xxd`—specialization slicing increased the number of procedures by 32.6%; however, the size of the

Name	Nodes	Blocks	Edges	Procedures	Callsites	Branches	Stmts
yes org	77	20	34	11	11	14	41
spec-slice	77	20	34	11	11	14	41
% change	0.0	0.0	0.0	0.0	0.0	0.0	0.0
base64 orig	599	99	268	42	89	169	299
spec-slice	615	101	275	44	88	174	309
% change	2.7	2.0	2.6	4.8	-1.1	3.0	3.3
dos2unix orig	604	136	293	50	128	157	269
spec-slice	623	147	304	54	132	157	280
% change	3.1	8.1	3.8	8.0	3.1	0.0	4.1
cat orig	684	117	304	51	112	187	334
spec-slice	718	127	321	56	116	194	352
% change	5.0	8.5	5.6	9.8	3.6	3.7	5.4
env orig	444	78	203	33	74	125	212
spec-slice	463	85	213	36	77	128	222
% change	4.3	9.0	4.9	9.1	4.1	2.4	4.7
shuf orig	651	139	311	56	130	172	293
spec-slice	776	179	376	67	160	197	352
% change	19.2	28.8	20.9	19.6	23.1	14.5	20.1
od orig	876	227	492	47	158	245	426
spec-slice	987	264	550	58	185	266	478
% change	12.7	16.3	11.8	23.4	17.1	8.6	12.2
tr orig	731	86	293	41	95	207	388
spec-slice	935	91	357	49	111	266	509
% change	27.9	5.8	21.8	19.5	16.8	28.5	31.2
cut orig	619	92	274	39	100	182	298
spec-slice	657	110	297	47	111	187	312
% change	6.1	19.6	8.4	20.5	11.0	2.7	4.7
xxd orig	775	198	426	43	138	208	386
spec-slice	893	240	489	57	168	229	439
% change	15.2	21.2	14.8	32.6	21.7	10.1	13.7

Figure 6.7: The effects of specialization slicing on code size.

replicated procedures comprised a smaller proportion of the overall code, because the change in code size was closer to 15%.

Four of the five programs that exhibit the largest amount of procedure duplication are the four interpreter-like programs, `od`, `tr`, `cut`, and `xxd`. The core behavior in each of these is implemented in a dynamically-controlled inner loop that interprets instruction objects stored in a list

structure. The bodies of these loops tend to be nested conditional statements that dispatch to a variety of auxiliary procedures. Because dynamic input data can flow through the loop body via a multitude of instances of auxiliary procedures, there are more opportunities for varied binding-time patterns in their call-sites compared to the non-interpreter-like programs. Even in these cases, the increase in the number of procedures is modest, ranging from 19.5% to 32.6%.

Worst-Case Slice-Materialization Microbenchmark. As previously described, the worst-case slice-materialization microbenchmark consists of programs P_n, where P_n is the n^{th} member of a family of programs that contain 2^{n+1} procedures in their materialized specialization slice results. By $n = 6$, the size of the materialized slice exceeds the size of the largest specialization-sliced version of a real-world program, `od`, despite the base worst-case slice-materialization benchmark being less than 10% the size of `od` before slicing.

Fig. 6.8 shows the sizes of the base program and the materialized slice results for programs P_3 through P_{10} from the slice-materialization microbenchmark. The columns labeled "Procedures" and "Callsites" illustrate the exponential growth particularly clearly.

This growth pattern is strikingly in contrast to the modest changes in size of the BusyBox applets. No specialization-sliced version of a BusyBox applet introduces more than thirty procedure copies. The worst-case slice-materialization benchmark exceeds that increase at $n = 5$.

P_n	Nodes	Blocks	Edges	Procedures	Callsites	Branches	Stmts
P_3	31	9	12	5	7	3	16
P_3 Materialized	104	24	31	16	15	7	66
P_4	40	11	15	6	9	4	21
P_4 Materialized	241	48	63	32	31	15	163
P_5	49	13	18	7	11	5	26
P_5 Materialized	546	96	127	64	63	31	388
P_6	58	15	21	8	13	6	31
P_6 Materialized	1219	192	255	128	127	63	901
P_7	67	17	24	9	15	7	36
P_7 Materialized	2692	384	511	256	255	127	2054
P_8	76	19	27	10	17	8	41
P_8 Materialized	5893	768	1023	512	511	255	4615
P_9	85	21	30	11	19	9	46
P_9 Materialized	12806	1536	2047	1024	1023	511	10248

Figure 6.8: The size of the base program and the materialized slice results for programs P_3 through P_{10} from the slice-materialization microbenchmark described in §6.2.

Findings

In practice, the code-size increase from slice materialization when binding-time analysis is performed via specialization slicing is small, and the real-world BusyBox programs exhibit at most a 32.6% increase in procedure count.

Similarly, the overall effect on code size is small—at most 31.2%.

This finding contrasts sharply with the effects of specialization slicing on the worst-case slice-materialization benchmark. For P_6, the program experiences a 16× increase in procedure count, with more than 100 procedures added, and a 21× increase in code size.

6.3.3 Answer to RQ3: What are the execution-time characteristics of a generating extension produced by GenXGen[C]?

Time to Specialize a Program. The columns labeled "Generating-extension execution time" in Fig. 6.9 and Fig. 6.10 show the amount of time taken for a generating extension to produce a residual program for a given static input.

The four microbenchmarks also exhibit sub-second specialization times, though all of them can in principle be scaled arbitrarily[5] Moreover, `power`, `dotproduct`, `interpreter`, and `filter` unroll a static loop. As described in §5.1.7, the generating extension only takes state snapshots at dynamic control points or procedure calls. Thus, `power` and `dotproduct` only traverse 14 unique states, despite unrolling 10,000 and 15,000 iteration loops, respectively.

[5]Other than `str_match` and `KMP`, the microbenchmarks are taken, with a few small modifications, directly from the WiPER experiments[Srinivasan and Reps, 2015]. In those experiments, the static input size can in principle be modified, but was hard-coded into the program, along with the size of any relevant arrays.

	static args	dynamic input size	states visited	Generating-extension execution time	Execution time		Resid speedup
					Original	Residual	
power	10,000	4	14	0.13 s	37.56 ± 0.12 µs	38.72 ± 0.2 µs	0.97x
dotproduct	15,000	4	14	0.53 s	107.82 ± 1.24 µs	387.98 ± 9.79 µs	0.28x
interpreter	<program text>	44	125	0.29 s	19.7 ± 0.05 µs	18.34 ± 0.25 µs	1.07x
filter	2	9	41	0.19	27.19 ± 0.23 s µs	26.15 ± 0.26 µs	1.04x
str_match	abababc	100,000	61	0.13 s	19869.39 ± 79.52 µs	16549.94 ± 95.14 µs	1.2x
kmp	abababc	100,000	384	0.9	17442.16 ± 91.83 µs	25255.53 ± 147.35 µs	0.69x

Figure 6.9: Run times for the generating extensions for the microbenchmarks, along with run times for the original and residual programs (with 95% confidence intervals).

	static args	dynamic input size	states visited	Generating-extension execution time	Execution time		Resid speedup
					Original	Residual	
yes		0	25	0.05 s	N/A	N/A	N/A
yes	hello	0	27	0.06 s	N/A	N/A	N/A
base64		100,000	77	0.17 s	6207.83 ± 49.22 µs	6276.57 ± 76.92 µs	0.99x
base64	-d	100,000	66	0.15 s	9930.76 ± 97.47 µs	11020.74 ± 114.02 µs	0.9x
dos2unix	-d	100,000	48	0.13 s	5011.09 ± 109.31 µs	5429.47 ± 133.38 µs	0.92x
dos2unix	-u	100,000	45	0.13 s	5347.41 ± 89.28 µs	4583.73 ± 87.64 µs	1.17x
cat	-n	100,000	56	0.13 s	29600.89 ± 207.99 µs	29371.85 ± 130.77 µs	1.01x
cat		100,000	41	0.09 s	2535.07 ± 60.46 µs	2622.58 ± 83.05 µs	0.97x
env	a=a b=b env	0	45	0.12 s	N/A	N/A	N/A
shuf	-i 1-1000	N/A	141	0.34 s	913.07 ± 14.21 µs	901.94 ± 15.69 µs	1.01x
od	-x	100,000	5865	41.0 s	53844.4 ± 296.42 µs	57340.55 ± 409.66 µs	0.94x
od	-X	100,000	5865	29.13 s	35573.47 ± 191.94 µs	36377.33 ± 244.6 µs	0.98x
od	-b	100,000	5865	29.88 s	87148.08 ± 407.73 µs	90177.03 ± 396.03 µs	0.97x
od	-e	100,000	5868	28.88	176695.98 ± 503.58 µs	179524.77 ± 591.82 µs	0.98x
tr	-d a	100,000	63	0.2 s	15315.0 ± 897.16 µs	12650.28 ± 400.51 µs	1.21x
tr	-s a	100,000	67	0.19 s	13116.92 ± 447.17 µs	14092.35 ± 388.56 µs	0.93x
tr	a b	100,000	65	0.19 s	9903.48 ± 276.48 µs	15881.39 ± 1459.4 µs	0.62x
cut	-b 3	100,000	81	0.23 s	18130.5 ± 140.11 µs	19916.3 ± 137.11 µs	0.91x
xxd[3]	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Figure 6.10: Run times for the generating extensions for the BusyBox applets, along with run times for the original and residual programs (with 95% confidence intervals). `yes` is a program that prints repeatedly until killed, so times are marked N/A.

[3]Due to an undiagnosed divergence issue with the specialization of `xxd`, I was not able to collect timing data for the residual program.

	static args	average hash time	total hash time	average hashes per block	average hashes per snapshot	min hashes per snapshot	max hashes per snapshot
power		112.38 µs	2.02 ms	0.01	1.29	1	2
dotproduct		97.82 µs	2.05 ms	< 0.01	1.50	1	3
interpreter	<none>	126.95 µs	19.93 ms	.126	1.03	1	2
filter	2	91.70 µs	7.24 ms	0.05	1.93	1	3
str_match	abababc	142.05 µs	9.38 ms	0.35	1.08	1	2
kmp	abababc	133.51 µs	59.01 ms	0.64	1.15	1	2

Figure 6.11: Average time to hash a single page, average number of page hashes per block, and minimum and maximum hashes computed in a single block for the generating extensions for the microbenchmarks.

	static args	average hash time	total hash time	average hashes per block	average hashes per snapshot	min hashes per snapshot	max hashes per snapshot
yes	y	143.7 µs	4.74 ms	0.37	1.20	1	3
yes	hello	135.1 µs	4.46	0.32	1.22	1	2
base64		145.18 µs	19.92 ms	0.47	1.16	1	3
base64	-d	140.73 µs	11.12 ms	0.40	1.20	1	3
dos2unix	-u	132.71 µs	7.30 ms	0.21	1.22	1	3
dos2unix	-d	129.37 µs	7.24 ms	0.21	1.22	1	3
cat		127.78 µs	6.26 ms	0.21	1.26	1	3
cat	-n	132.57 µs	7.95 ms	0.22	1.22	1	3
env	a=a b=b env	127.200 µs	7.37 ms	0.27	1.20	1	3
shuf	-i 1-100	144.63 µs	22.27 ms	0.36	1.09	1	3
od	-x	148.58 µs	947.81 ms	0.75	1.09	1	2
od	-X	148.64 µs	948.15 ms	0.74	1.09	1	2
od	-b	148.75 µs	948.86 ms	0.76	1.09	1	2
od	-e	148.75 µs	948.86 ms	0.76	1.09	1	2
tr	-d	116.64 µs	6.77 ms	0.08	1.45	1	4
tr	-s a	125.37 µs	11.03 ms	0.12	1.31	1	4
tr	a b	123.51 µs	10.75 ms	0.12	1.34	1	4
cut	-b 3	135.42 µs	13.00 ms	0.18	1.18	1	3
xxd[4]		n/a	n/a	n/a	n/a	n/a	n/a

Figure 6.12: Average time to hash a single page, average number of page hashes per block, and minimum and maximum hashes computed in a single block for the generating extensions for the BusyBox applets.

[4]Due to an undiagnosed divergence issue with the specialization of `xxd`, I was not able to collect size data for the residual program.

For the three feature-removal programs, `base64`, `dos2unix`, and `cat`, i.e., those programs whose specialization times do not scale with the size of the input, specialization times are under a second. Overall, specialization times are reasonable, with every specialization taking under one minute, except for `od`. Due to an interaction, described in §6.3.5, between structural properties of `od` and slicing, the generating extension explores a large number of state-block pairs that are functionally redundant, and explores far more states than the other programs tested.

In addition, due to a divergence issue that I have not been able to diagnose the cause of, specialization of `xxd` does not terminate after a five-minute timeout. For the other interpreter-like programs, specialization times are more in line with a feature-removal task. These results suggest that specialization times in practice are reasonable, because even though arguments can be combined arbitrarily in a large list, programs like `od` and `tr` tend to most frequently be used with a small set of parameters to perform simple tasks.

Unlike the other BusyBox programs, which accept arguments from a finite set of flags, the BusyBox programs `yes` and `env` accept a list of strings as input; both the set of strings and the individual strings can become arbitrarily large. Each of the programs loops over the list of strings, passing each to a library call. Thus, in practice the time consumed by specialization scales with the size of the input. For the examples tested, the specialization times are fast, both completing in under a second.

`Shuf`'s static inputs can grow arbitrarily as well, but as discussed in §6.3.5, in practice, static loop unrolling does not occur, so specialization behaves more like a pure feature-removal task.

Hash Times. Fig. 6.11 and Fig. 6.12 show various hashing statistics for the BusyBox programs and microbenchmarks, respectively. For the microbenchmarks, the times range from 91.70 to 142.05 microseconds, and

similarly for the BusyBox applets, the average time to hash a single page is 116.64-148.75 microseconds. For every program tested, even `od` (which runs for over forty seconds in one case), the overall amount of time spent on hashing is less than a second.

A comparison of the proportion of total hash time from Fig. 6.12 relative to overall generating-extension execution time in Fig. 6.10 reveals that for the BusyBox applets, the total time spent computing hashes constitutes 3.8% to 11.7% of the total generating-extension execution time The generating extension for `filter` spends the smallest proportion of execution time on hashing, and `base64`'s generating extension with no static arguments spends the largest proportion of execution time on hashing. Thus, the bulk of the specialization time is taken by other factors, such as IPC, process-management, and generation of residual code.

We also track several metrics that measure the relative volume of hashing performed per basic block/state pair de-queued by the worklist. In particular, we track several per-snapshot-operation metrics, including the number of pages hashed and used to update the state hash. At least one page was always updated per hash, which is due to how the switch between subject and meta-state is implemented, as described in §5.1.3, and in all cases, the average number of pages per snapshot is less than two. At most, four pages were hashed in a single snapshot operation. This number is reasonable, because individual basic blocks tend to be small, and handle a fairly focused set of operations.

Because a snapshot may occur after several basic blocks have been specialized, due to our on-line jump compression, we also measure the average number of pages hashed per block. For programs that consist of a simple statically-controlled loop, like power and dotproduct, the average hash count is extremely low. For all programs, the average is less than one.

Findings.

For all the programs tested, specialization times are reasonable, taking less than a minute on inputs corresponding to typical use cases.

Moreover, page hashing is a comparatively small part of execution time, and other state-management and IPC factors contribute over 70% of the execution times in the BusyBox applets.

6.3.4 Answer to RQ4: How does specialization affect the run-time performance of the residual program? How does the difference in run-time performance change as the size of the dynamic input increases?

Execution on Representative Inputs. The last three columns of Fig. 6.9 show the execution times of the original and residual microbenchmark programs on representative dynamic inputs, along with the speedups of the residual programs. The non string-matching programs `power` and `dotproduct` both exhibit some degree of slowdown. In the case of `power`, we believe that this is largely due to poor instruction-cache locality: the residual code is 15,000 repetitions of `acc *= acc`, and we believe that the cost of instruction-cache misses outweighs the elimination of the evaluation of the loop condition.

In the case of `dotproduct` the original program is over three times as fast as the residual. We believe that this is partially an artifact of the byte-granularity lifting performed in §5.1.6. Both the original and residual programs are compiled at `-O0`, and are thus completely unoptimized, so there will be four memory loads or stores for every lift of an integer value. However, `dotproduct` also loads or stores from three memory locations in every iteration of the loop, and thus increased cache pressure may also

play a role in the slowdown.

The fact that `interpreter`, which is also a loop-unrolling task, but in which the loop body performs character-sized writes exhibits a small speeedup suggests that lift size plays a role in residual-program performance. The residual version of `filter` also exhibits a modest speedup. However, `filter` is a much shorter-running program, manipulating a 3×3 image, suggesting that the elimination of startup overhead dominates the other effects on residual-program speed.

The two string matchers `str_match` and `kmp` exhibit interesting behaviors. `Str_match` exhibits a 20% speedup. This program seems to be an ideal case for loop-unrolling: all the operations are byte-granularity, and the unrolled inner loop is small enough to exhibit good cache locality, which yields the upsides of eliminating a static loop-condition check, while avoiding the detrimental effects of loop unrolling and `GenXGen`'s lifting strategy.

Interestingly `kmp`, whose residual program is, in principle, asymptotically faster than a naive string matcher, is 30% slower than the original. The code generated is a set of jumps and procedure calls that encode the finite automaton produced by the Knuth-Morris-Pratt string-matching algorithm[Knuth et al., 1977; Consel and Danvy, 1989]. In particular, `kmp` has a `sub_match` procedure, which in the residual program can return to one of a variety of blocks corresponding to various static states, depending the contents of the dynamic string in which the search is carried out. Thus, the residual program calls a series of disjoint procedures, and after returning from each one, the program executes a jump to a target in an exit-splitting table. Consequently, the code is large and non-local, likely creating large amount of instruction-cache pressure not present in the original program. In particular, it appears that for short strings, the overhead inherent in the specialized code outweighs the asymptotic advantage.

Overall, the BusyBox programs largely exhibit a small slowdown of

several percent, likely due to the aforementioned lift-granularity issue. The two that do exhibit a speedup, `dos2unix -u` and `tr -d a`, operate on the contents of their respective input streams at byte granularity, further indicating the importance of lift size.

Scaling Dynamic Inputs. Figs. 6.13, 6.14, and 6.15 show the execution times for the original and residual programs (in nanoseconds) as the size of the dynamic input is increased to 10,000 bytes.

(Note: Figs. 6.13, 6.14, and 6.15 use smaller dynamic inputs than were used in Fig. 6.10.)

Somewhat surprisingly, the two microbenchmarks for which I ran input-scaling experiments, the string-matching programs `str_match` and `kmp`, exhibit different scaling behaviors. The residual version of `kmp` exhibits a clear trend towards the residual program being much slower, due to the previously discussed factors, while the base string matcher performs better as input scales.

Overall, the programs that represent feature-removal tasks (Fig. 6.13) exhibit scaling indicative of near-parity in performance between the original and residual program. This result is reasonable, because, structurally, the static input selects one of several loops; the selected loop itself is dynamically controlled, and generally the loop body cannot be specialized in any non-trivial way. For `base64`, the residual version specialized with respect to `-u` performs similarly to the original, while for the residual version specialized with respect to `-d`, the original version is faster.

Both `cat` and `cat -n` trend towards performance parity between the original and the residual program. Interestingly, for `cat` with no arguments, we see a pronounced step-wise increase in times, which appears to be due to the fact that in this case `cat` reads input in larger buffers than the other programs. Similarly, `dos2unix` appears to scale similarly, with the residual program for the `-d` case being slightly slower.

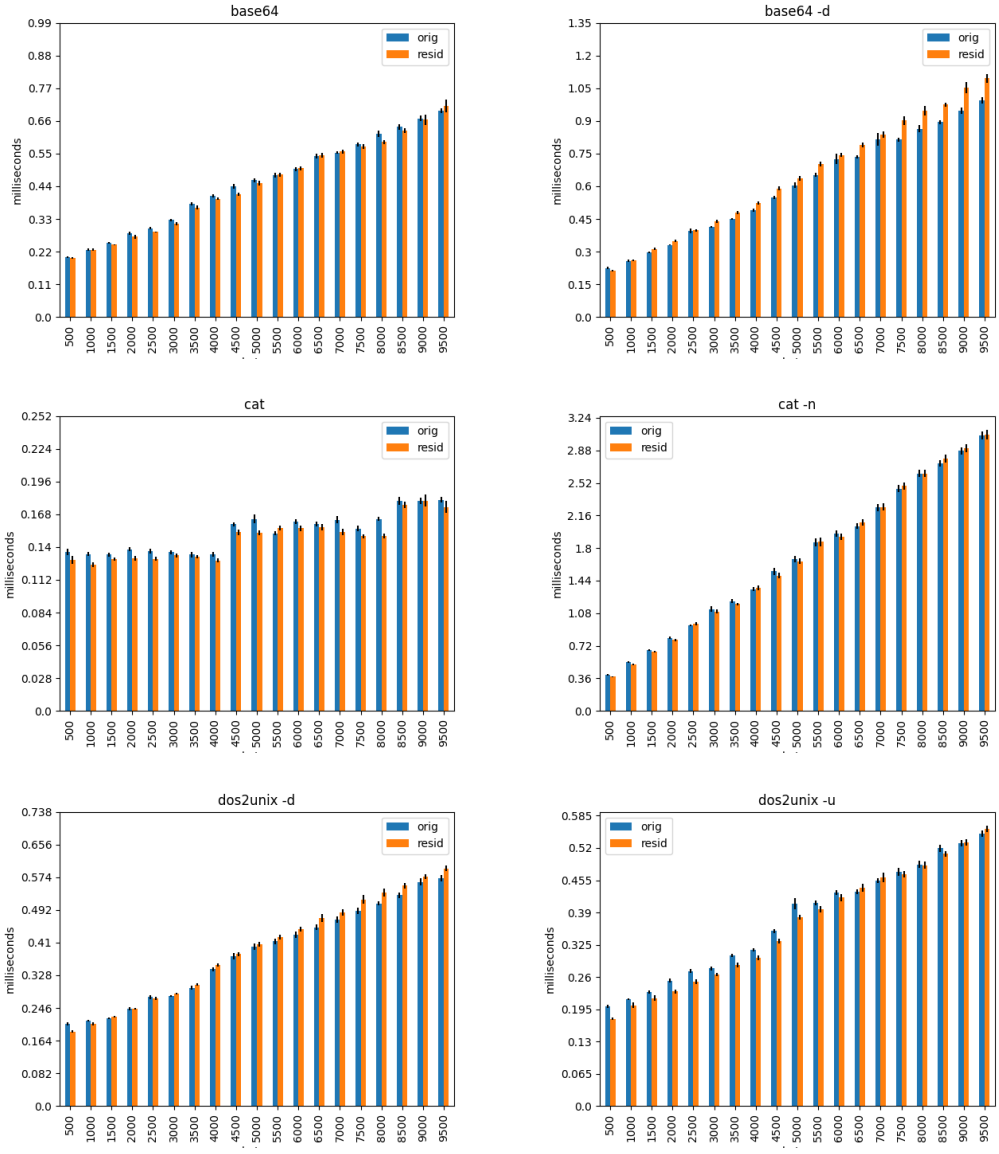


Figure 6.13: Execution times of original and residual programs in nanoseconds as input size in bytes is increased. Black lines denote 95% confidence intervals.

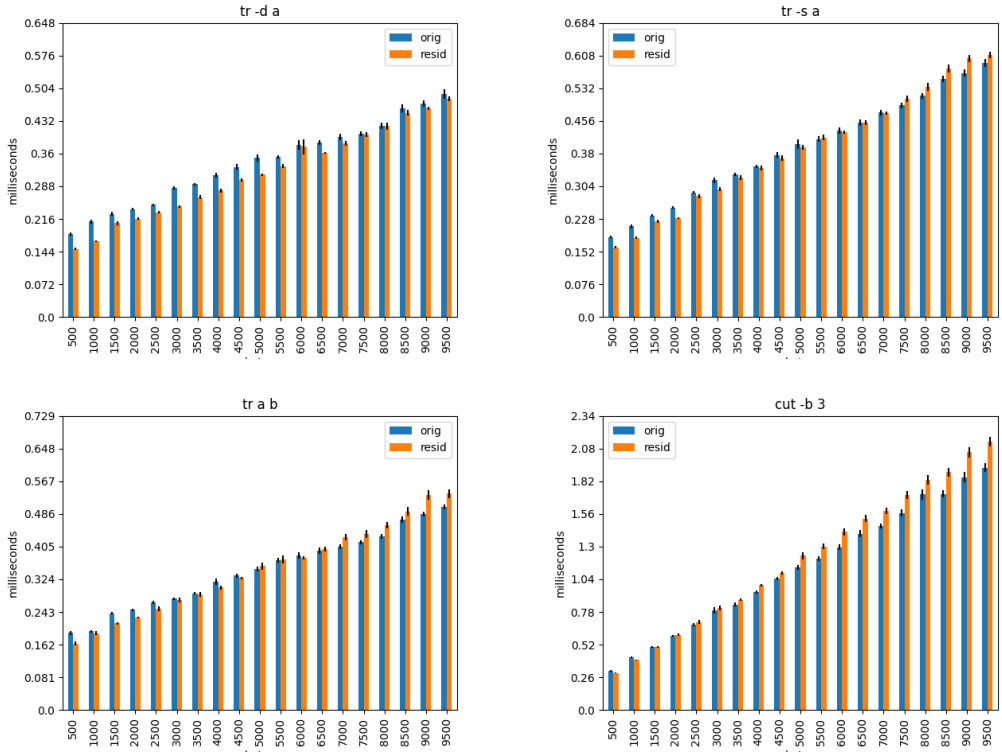


Figure 6.14: Execution times of original and residual programs in nanoseconds as input size in bytes is increased. Black lines denote 95% confidence intervals.

The three specializations of interpreter-like programs, shown in Fig. 6.14, except for `tr -d a`, exhibit scaling behavior that favors the original program. In the case of `tr -d a`, the residual program appears to remain faster, although the gap appears to close as the input size increases.

As input size increases, all four specializations of `od` become slower than the original. In `od`, dynamic dependences are carried into the control-flow predicates within the loop body, and moreover, as will be discussed in §6.3.5, many duplicate residual copies of blocks and procedures are produced, which likely affects instruction-cache locality.

With the exception of `od` and `kmp`, every figure clearly shows the effects

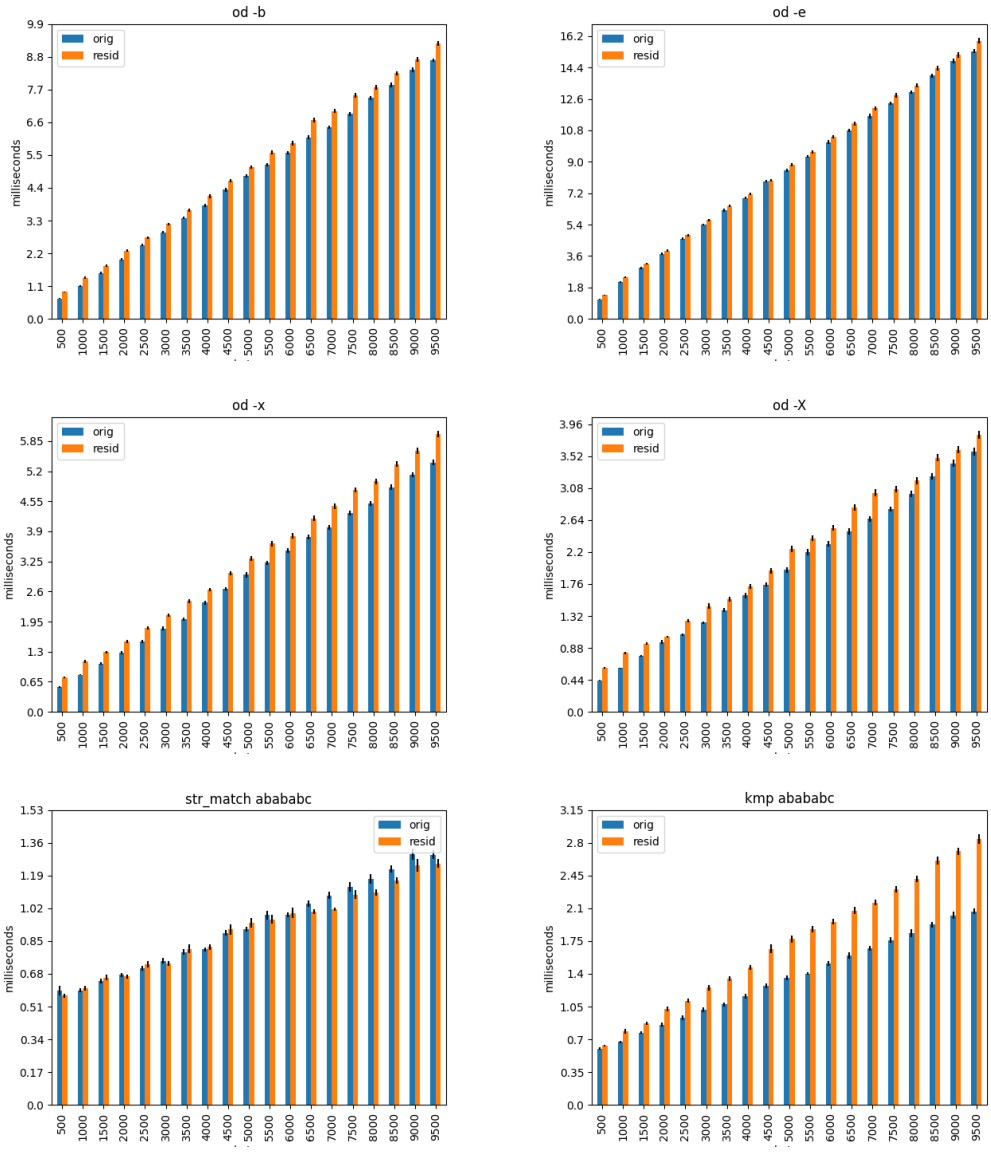


Figure 6.15: Execution times of original and residual programs in nanoseconds as input size in bytes is increased. Black lines denote 95% confidence intervals.

of eliminating initial setup overhead due to, e.g., eliminating argument parsing, although in the majority of cases that advantage is small compared to other factors as input size scales.

Findings

On this set of programs, the speedup results are mixed. In particular, the naive loop unrolling performed as part of specialization is often detrimental, due to poor instruction-cache locality. Moreover, the byte-granularity lifting may be detrimental to performance in the residual program, as suggested by the microbenchmarks.

Specialization does not have a significant impact on execution time for "feature-removal" tasks.

6.3.5 Answer to RQ5: How does specialization affect the size of the residual program?

Figs. 6.16 and 6.17 show the sizes of each program, and the numbers of different kinds of program elements.

For all six microbenchmarks, shown in Fig. 6.17, specialization produces a significant increase in program size, which is reasonable, because in all six cases, the specialization task contains one or more loops that are unrolled based on the static input. In the case of `power`, `dotproduct`, and `interpreter`, the program is completely unrolled, eliminating all conditional branches. Specializing `filter` selects one of five algorithms, and unrolls the main loop, emitting code only for the selected algorithm, with the unrolling outweighing the effect of removing the other algorithms ("feature removal"). In `str_match`, the inner loop is unrolled.

For the three feature-removal tasks from BusyBox, `dos2unix`, `base64`, and `cat`, the relative paucity of loop-unrolling combined with feature removal yields a significant reduction in code size, as well as lower numbers

Name	Static Args	Nodes	Blocks	Edges	Procedures	Callsites	Branches	Stmts
yes orig		77	20	34	11	11	14	41
yes	y	80	39	47	9	13	8	50
yes	hello	114	140	152	10	30	12	62
base64 orig		599	99	268	42	89	169	299
base64	<no argument>	284	148	212	24	49	64	147
base64	-d	310	170	232	22	70	62	156
dos2unix orig		604	136	293	50	128	157	269
dos2unix	-u	120	64	79	12	17	15	76
dos2unix	-d	152	117	133	13	44	16	79
cat orig		684	117	304	51	112	187	334
cat	<no argument>	389	121	253	32	60	132	165
cat	-n	383	191	245	32	91	54	206
env orig		444	78	203	33	74	125	212
env	a=a b=b env	342	107	250	23	44	143	132
shuf orig		651	139	311	56	130	172	293
shuf	-i 1-1000	416	146	173	23	60	27	306
od orig		876	227	492	47	158	245	426
od	-x	37745	26514	39092	219	4421	11930	21175
od	-X	43324	37572	51216	239	8904	11268	22913
od	-e	47341	39450	53606	380	9696	11773	25492
od	-b	38941	39253	51929	242	8029	10363	20307
tr orig		731	86	293	41	95	207	388
tr	-d a	2025	666	1076	31	317	410	1267
tr	-s a	2035	760	1084	33	360	326	1316
tr	a b	2021	747	1064	32	358	319	1312
cut orig		619	92	274	39	100	182	298
cut	-b 3	659	108	297	44	111	189	315
xxd orig		775	198	426	43	138	208	386
xxd resid[6]		n/a	n/a	n/a	n/a	n/a	n/a	n/a

Figure 6.16: Sizes of original and residual BusyBox applets. The column labeled "size," which gives the sum of the values in the columns to the right, is a measure of overall program size

[6]Due to an undiagnosed divergence issue with the specialization of `xxd`, I was not able to collect size data for the residual program.

Name	Static Args	Nodes	Blocks	Edges	Procedures	Callsites	Branches	Stmts	Size
power orig		70	20	30	10	11	10	39	151
power resid	10000	1057	54	59	9	23	5	1020	1207
dotproduct orig		83	20	34	10	11	14	48	172
dotproduct resid	15000 2	11077	57	63	10	24	6	11037	11237
interpreter orig		106	25	46	10	12	21	63	220
interpreter resid	<program>	222	171	177	10	25	6	181	611
filter orig		127	26	46	10	11	20	86	240
filter resid	2	1305	87	110	10	25	23	1247	1560
str_match orig		95	27	43	11	16	16	52	208
str_match resid	abababc	176	114	146	12	31	32	101	511
kmp orig		123	32	54	14	17	22	70	262
kmp resid	abababc	1009	684	899	26	76	129	778	2823

Figure 6.17: Sizes of original and residual microbenchmark programs. The column labeled "size," which gives the sum of the values in the columns to the right, is a measure of overall program size.

for all the different kinds of program elements, except for the number of basic blocks.

In the case of `yes` with an argument, although still smaller than the original version, we see an increase in code size over the no-argument residual program, due to the lifting of the contents of the argument into an unrolled loop.

In the case of `shuf -i`, the shuffle loop is not unrolled—the dynamic dependence is propagated to the loop control. Overall, specialization reduces the code size because argument parsing (and corresponding error handling) is eliminated, and all code relating to shuffling a specified input file is eliminated.

The four interpreter-like programs, `od`, `tr`, `cut`, and `xxd` exhibit some degree of code-size increase, especially `od`. All four of these programs have a dynamically controlled loop; they read `stdin` until reading end-of-file. However, the loop body has multiple static components that can be specialized inside the loop, namely lists or tables that contain instructions on how to process the input—these static lists and tables are constructed from the static input. However, the dynamic data that flows through the loop forces many of the static control constructs to become dynamic.

This effect can be quite large. In the case of `tr`, and especially `od`, there are multiple if statements in series inside the loop body. Each branch of one of these if statements sets one or more flags to a different value than the other branch. Moreover, some of the flags set by each of these statements differ from all other if statements in the series. Thus, the post-states of every branch of the if statement in the series is distinct from all others. This situation means that at the end of the loop body, the number of post-states is exponential in the number of serial if statements in the loop body. Because the flags are Booleans or enums, the number of possible states is finitely bounded, and the specialization loop converges, but produces a large number of functionally redundant blocks.

Findings

For the programs examined, we see significant reductions in code size in some programs, due to the elimination of unreachable code. In particular, specialization is useful for "feature-removal" tasks.

Conversely, as expected, unrolling tasks induce a significant increase in size.

Due to dynamic data dependencies being carried into a loop under dynamic control, a significant amount of redundant code can be produced for the "interpreter-like" programs.

6.3.6 Answer to RQ6: How much code does specialization remove from a program?

To evaluate the degree of "debloating" performed by the generating extension, we recorded the percentage of procedures from the original program that are not present in the residual program. We report the results in Fig. 6.18. Specialization of `dos2unix`, `cat`, `base64`, and `shuf` performed a substantial amount of debloating: in all three programs, more than 40%

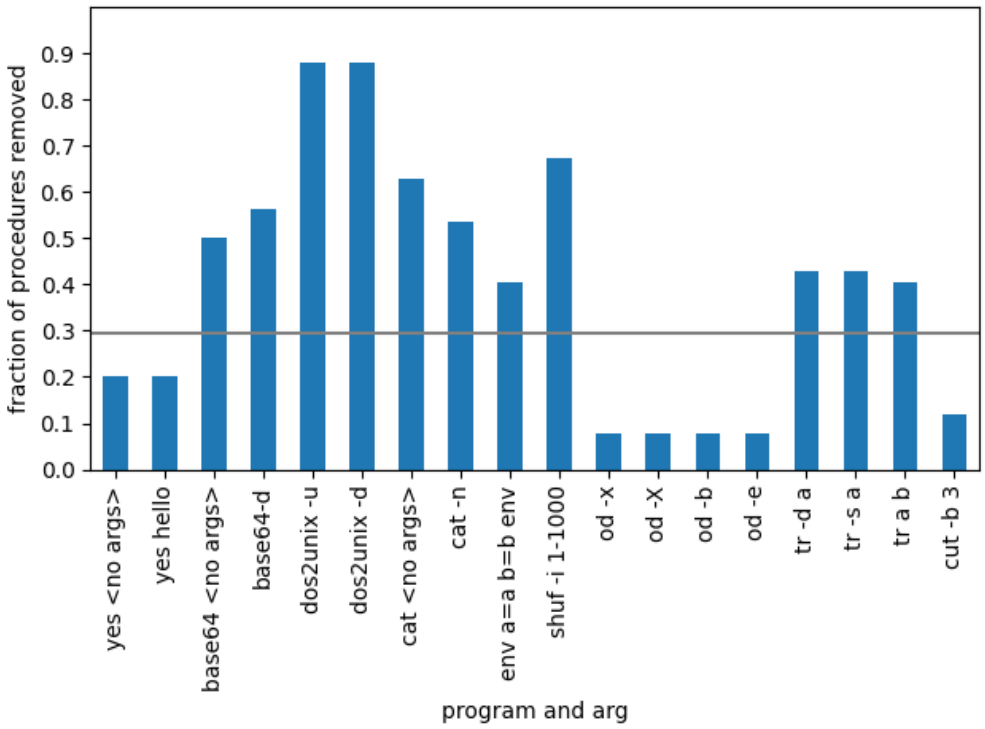


Figure 6.18: Procedures not present in the residual program, as a fraction of procedures in the original program. (Larger numbers are better. The horizontal line shows the geometric mean.)

of the original procedures were eliminated by specialization.

All four of these programs typify standard "feature-removal" tasks. Both `dos2unix` and `base64` are bidirectional encoders, where the direction of encoding is selected by a command-line flag. Thus, in both cases, selecting one flag excludes code associated only with the complementary case. Like the other two, `cat` is also a feature-removal task. `cat` can do a lot more than just echo its input, such as numbering the output lines and showing non-printing characters, and much of the printing code for a given case is disjoint from the other cases.

Similarly, `shuf` produces a random permutation of a variety of items:

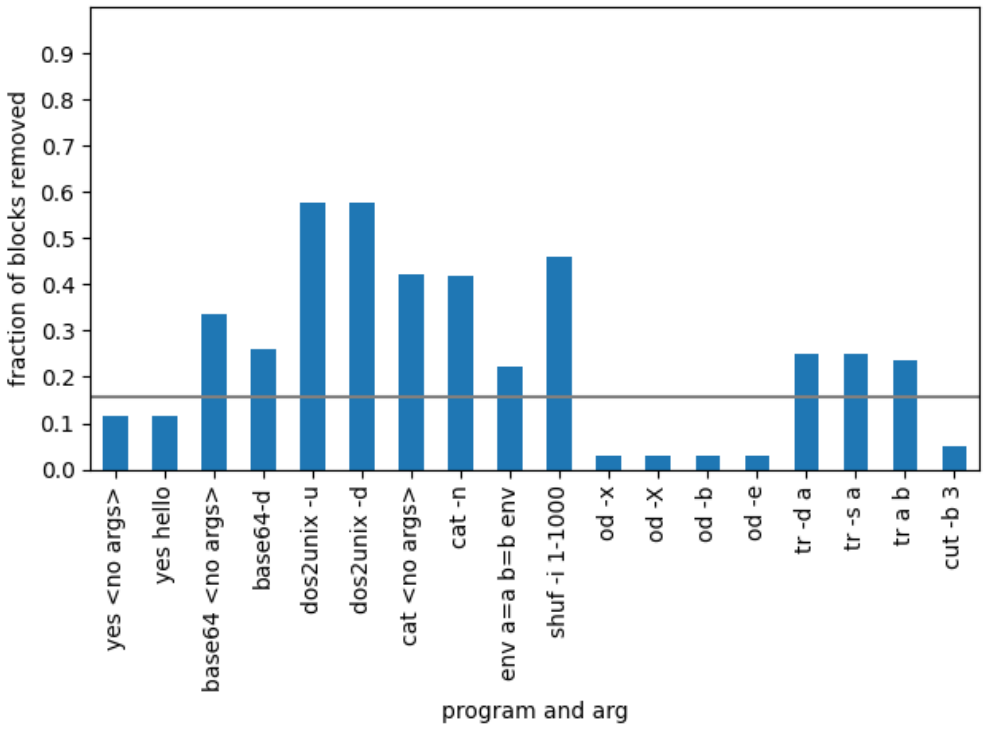


Figure 6.19: Blocks not present in the residual program, as a fraction of blocks in the original program. (Larger numbers are better. The horizontal line shows the geometric mean.)

input files, discrete tokens in the arguments, or numbers in a specified range passed as arguments. When specialized with respect to the range $1 - 1000$, specified by the flags `-i 1-1000`, all file input procedures are eliminated.

Now consider the kinds of actions that can be specialized away in `dos2unix`. To follow symbolic links, a procedure that resolves filenames is called, which must, among other things allocate heap space for a string large enough to contain the final filename. Moreover, to create a temporary file, yet more space must be allocated to serve as a template for the temporary filename, and then the file is created, opened, has flags and attributes

set, and so on. At the end if `dos2unix`, everything must be closed, the temporary file must be renamed to the target filename, and all allocated strings must be freed. Many of these operations can fail, and thus ancillary error-handling procedures can be called. In all, these activities comprise over half of the procedures in `dos2unix`, and none of it is retained in the residual program.

In addition, `dos2unix`, `cat`, and `base64` were specialized with respect to arguments that cause them to be used as filters that read standard input and print to standard output. In contrast, if a filename is passed as an argument (to the original programs), a significant amount of file-IO boilerplate is invoked, such as code to check for the existence of the input and output files. The generating extension never visits this code—the partial state is sufficient to establish that the code could never be executed by the residual program—and thus it is not present in the residual program.

Out of these, the removal of 80% of `dos2unix`'s procedures is remarkable enough to warrant further consideration, particularly because the clusters of procedures removed typify the feature-removal results seen in all the BusyBox experiments. As with all other BusyBox programs tested, arguments are parsed with the standard Uinix-style `getopt` procedure, which contains a variety of procedures for parsing, e.g., integers from strings. Because the only argument is `-d` or `-u`, the majority of this code is never reached, and thus specialized away, leaving only a small number of residual procedures that contain lifts of the computed argument-parsing results.

In addition, as specialized, `dos2unix` reads from `stdin` and write to `stdout`. However, `dos2unix` can also take a single filename as an argument. When invoked in this way, `dos2unix` converts the line endings in the file. Because the input and output file are the same, it is necessary to create a temporary file. Although conceptually simple, this task requires a whole

host of procedures to function, particularly in the presence of symbolic links (because the temporary file is placed in the same directory as the final output file). In general, unused flags often correspond to large clusters of utility code and error-handling code that is often not shared with other portions of the program. This effect is amplified in `dos2unix`, because the core loop of the program is straightforward, mapping Unix line endings to or from DOS conventions, which it does character-by- character, using only `getc/putc`, and no memory allocation or complex error handling is needed.

GenXGen[C] removes less code in `yes`, which is reasonable because `yes` does not expose any switch-configurable features. Moreover, `yes` is smaller than the other BusyBox applets programs, so set-up code that is shared across all BusyBox applets (and retained in the residual programs) takes up a larger proportion of the code for `yes`.

For `od` and `cut`, two of the three interpreter-like programs, GenXGen[C] removes a much smaller proportion of features. For the arguments on which they were specialized, both programs have a dynamically-controlled loop, and a loop that performs static actions from a list provided as a static parameter. In contrast, `tr`—the third interpreter-like program—has a significant number of dynamic data dependencies carried through the loop body, which reduces the amount of code that can actually be removed. This observation suggests that partial evaluation under dynamic control causes more difficulties for generating extensions produced from specialization-slicing-based BTA than other partial-evaluation tasks.

The ratios shown in Fig. 6.19—of blocks not present in the residual program, as a fraction of blocks in the original program—are an undercount of the actual ratios. The ratios shown in Fig. 6.19 are computed as the total number of blocks in the removed procedures as a fraction of the overall number of blocks in the residual program. That is, they do not count any of the blocks that may have been removed in the procedures

that remains in the residual program. The reason we chose this metric is due to limitations of CodeSurfer's program representation, discussed in§8.2, which make it time-consuming to easily obtain the relationship between statements in a materialized specialization-slice result and the original procedure.

However, the ratios as computed in Fig. 6.19 are still a useful metric, and a reasonable proxy for overall blocks removed. For example, it could be the case that although a large number of procedures are removed, they are all quite small, and the overall reduction in block count is quite small. Thus, given two programs A and B of roughly identical size, program A could have a large number of procedures removed, but program B has one procedure removed which contains more blocks than all of the removed procedures combined.

This phenomenon does not show up in the BusyBox programs. Overall, the percentage of blocks removed seems proportionate to the percentage of procedures removed. The feature-removal tasks still have the greatest reductions, and, e.g., `dos2unix` still has the largest proportion of blocks removed. Overall, the eliminated procedures tend to be smaller utility procedures,

Findings

GenXGen[C] is able to remove a substantial fraction of the subject program's code when used to specialize a command-line utility with respect to command-line flags that control which options of the command-line utility are invoked.

For six of the nine programs, GenXGen[C] removes over thirty percent of the available procedures. These removed clusters of procedures that correspond to functionality irrelevant to the passed command-line flags, which indicates that GenXGen[C] is suitable for feature-removal debloating.

For the same six of the ten BusyBox applets, GenXGen[C] removes over twenty percent of the original program's blocks. This number is a smaller, but still significant, reduction in size.

Results are mixed for the interpreter-like programs. Over thirty percent of procedures are removed from `tr`, but fewer are removed for `od` and `cut`. In particular, this finding suggests that specialization of static code under dynamic control is difficult, even with specialization slicing.

6.4 Experimental Evaluation of GenXGen[mc]

Although GenXGen[mc] does not implement specialization slicing-based BTA, and struggles to scale to programs comparable in scale to those specialized by GenXGen[C], the evaluation of GenXGen[mc] is nonetheless interesting and elucidates important details about my overall design and implementation choices for GenXGen. In §6.4.1, I describe the smaller set of experimental questions addressed in my evaluation of GenXGen[mc], and the metrics used to evaluate the questions. In §6.4.2, I describe the GenXGen[mc]-specific aspects of the setup, and identify which parts are shared with GenXGen[C]. In §6.4.2.1, I explain the modified state management strategies that I use to assess the impact of hashing and CoW on generating extension performance. In §6.4.3, I discuss the experimental results.

6.4.1 Experimental Questions.

Research Questions

- **RQ1-MC.** What are the individual improvements contributed by CoW and fingerprinting to memory usage by a generating extension?
- **RQ2-MC.** What are the individual improvements contributed by CoW and fingerprinting to the time taken by a generating extension to emit a residual program?
- **RQ3-MC.** Compared to the original subject program, how much does specialization speed up execution?
- **RQ4-MC.** How does specialization affect the size of a program in terms of number of instructions? Moreover, for the subset of programs that correspond to feature-removal tasks, what is the degree of debloating performed?

RQ1-MC. *What are the individual improvements contributed by CoW and fingerprinting to memory usage by a generating extension?*

Rationale. Because the generating-extension runtime uses the operating system's underlying CoW mechanism, unchanged pages are shared between parent and child whenever a process is forked to create a snapshot. Thus, a GenXGen generating extension may use less memory per state snapshot than a strategy that doesn't use CoW. In addition, because a GenXGen generating extension can determine whether a state has been visited previously by retaining state hashes, it is free to garbage-collect state processes that do not have a corresponding state/block pair in the worklist. Consequently, overall memory usage may differ significantly from naive state-management strategies.

Metrics. For each program, I record the memory usage of the generating

extension on a single representative static input. I also record the number of live non-code pages used by all state processes across the execution of a GenXGen[mc] generating extension. In addition, I also added alternative state-management modes that did not use copy on-write, and did not garbage-collect unneeded states. I compare memory usage between the four possible strategies: The standard GenXGen behavior with both active, GenXGen with hashing disabled (hence disabling garbage collection), GenXGen without CoW, and GenXGen with both disabled.

RQ2-MC. *What are the individual improvements contributed by CoW and fingerprinting to the time taken by a generating extension to emit a residual program?*

Rationale. Like RQ3 for GenXGen[C], GenXGen[mc] uses the same language-agnostic runtime to implement its state-management mechanisms. Thus, it is prudent to investigate the time taken to execute a generating extension, as with GenXGen[C]. In particular, GenXGen uses a proabilisitic hashing scheme to provide $O(1)$ state-repetition checking, which is asymptotically faster than non-hashing schemes. Thus I also choose to compare against the three naive strategies described in RQ1-MC.

Metrics. As with GenXGen[C], for each program I record the execution time of a generating extension on a representative static input. I also chose to compare the performance of GenXGen[mc]'s generating extensions against generating extensions that use naive strategies; i.e., the three versions that disable at least one of (a) hashing, and (b) CoW.

RQ3-MC. *Compared to the original subject program, how much does specialization speed up execution?*

Rationale. In addition to the speedups described in RQ4 for GenXGen[C] (see §6.1), GenXGen[mc], as described in Chapter 5, performs procedure inlining, and fine-grained lifting at the register level, potentially eliminating a significant number of memory reads and writes. However, optimizations such as loop-unrolling may be detrimental to cache locality. Because of the potentially complex interplay between these factors, I

compare the performance of the original and residual programs.

Metrics. For each program, I record the execution time of the original and residual program for a single representative static and dynamic input pair.

RQ4-MC. *How does specialization affect the size of a program in terms of number of instructions? Moreover, for the subset of programs that correspond to feature-removal tasks, what is the degree of debloating performed?*

Rationale. As described in RQ4 for GenXGen[C] (see §6.1), there is a tension between specialization as a tool for unrolling and inlining—which can increase program size—and as a tool for feature removal—i.e., for eliminating calls to code unrelated to the features invoked by the static input. Thus I measure the effect of program specialization on program size.

Metrics. For every program specialized, I measure the instruction count of the original and residual program. For the two feature-removal tasks, `gnu-wc` and `lzfx`, I also measure the change in procedure count and number of call-sites between the original and residual programs.

6.4.2 Experimental Setup.

I evaluated GenXGen[mc] using the binaries of six microbenchmarks, and four non-microbenchmark binaries, corresponding to simple command-line utilities. These programs are described below.

Microbenchmarks. The microbenchmarks include five of the GenXGen[C] microbenchmarks from §6.2: every one but `KMP`, i.e., `power`, `dotproduct`, `interpreter`, `filter`, and `str_match`. The new sixth microbenchmark is

- `sha` is a simple implementation of the `sha1` algorithm, which computes a digest of a 1024-bit string. The static input is the first 512 bits.

Command-Line Utilities. I also evaluated four other binaries: two GNU coretuils programs, a compression algorithm, and one program that uses a simple statically-linked version of `printf`:

- `gnu-wc` counts lines, chars, or words in `stdin`. The static input specifies which quantities are counted; the dynamic input is `stdin`.
- `lzfx` is an implementation of the LZFX compression algorithm, and contains both compression and decompression routines [Collette, 2013; Ziv and Lempel, 1977, 1978]. It was also used to test feature extraction in WiPER. The static input determines whether the input file is compressed or decompressed; the dynamic input is the input file.
- `printf` is a program that calls into a simple printf library. The static input is a format string; the dynamic input is the remaining arguments.
- `gnu-env` runs a program with a specified assignment to environment variables. The static input is the assignment to environment variables; the dynamic input is the name of the program to invoke.

Command-Line Utility Specialization. These programs present a cross section of real-world specialization tasks. Both `gnu-wc` and `lzfx` represent a *feature-removal task*, in which a single mode of operation is chosen out of a set of potential modes.

The program `printf` is an instance of the class of *loop-unrolling and inlining tasks*, in which a specialized library call is in-lined into a program. The fourth program, `gnu-env`, features aspects of both tasks, because the core environment-update loop is unrolled, and features corresponding

to unused command-line flags are excised.[7] Fortunately, in many circumstances, the subject program can be adapted to overcome the limitations, e.g., by manually unrolling a loop. However, the effort required to identify appropriate rewritings to overcome current limitations of the static analyses in CodeSurfer/x86, as well as to model calls to library functions, limited the number of real-world programs that we were able to use for our study.

For `gnu-wc`, specializing with respect to the static input selects one of three main application loops, each of which is optimized for a different counting task. The generating extension eliminates the other two loops. Similarly, for `lzfx`, specializing on the static input selects either the compression or decompression routine, removing the routine not chosen.

In the case of `printf`, the specialization unrolls the format string, eliminating run-time parsing and logic for unused format specifiers. Similarly, in `gnu-env`, the argument-parsing loop is unrolled, emitting a program that runs a program in a pre-defined environment.

6.4.2.1 Disabling CoW and Hashing

To evaluate RQ1-MC and RA2-MC we implemented GenXGen[mc] so that CoW and state fingerprinting can be independently disabled in generating extensions, yielding four possible execution modes (see Fig. 6.20).

[7]The reason we used only four command-line utilities in our study was because of limitations (described in more detail in Chapter 8) of CodeSurfer/x86 [Balakrishnan and Reps, 2010; Balakrishnan et al., 2005a], which GenXGen[mc] uses to implement BTA. Because CodeSurfer/x86's analysis results are based on an overapproximation of the set of states that can actually arise during execution, the dependence graph constructed by CodeSurfer/x86 often contains spurious dependence edges—i.e., the graph contains an overapproximation of the actual dependences that exist among program elements. When BTA is performed on such a graph, some variables and instructions are identified as being static that one would like to be identified as dynamic. The previous sentence has it backwards: with extra dependence edges, a forward slice from the dynamic input(s) would identify elements as dynamic that you would have liked to have been static. The consequence is that the resulting generating extension cannot do much to specialize the subject program.

To simulate disabling of CoW, we added a mechanism to force the copy of an entire process address space. When CoW is "disabled," we dirty each page without altering the state by (i) writing a single byte to each page in the address space, and then (ii) reverting the page back to its original state. These actions force every page to incur a CoW fault, causing the OS to create a copy of every page in the address space. This approach provides an upper bound on the time required because, by forcing the CoW mechanism to make the copy, a page fault must be handled by the kernel for every page, adding some overhead. We chose to estimate the cost in this way because our generating extensions are inherently multi-process: each process holds a single state. Implementing a true CoW-free approach would have required modifying the OS to eliminate CoW, which seemed unwarranted, given that the technique is not likely to be competitive.

To disable fingerprinting, we implemented an alternative version of the generating extension's state-comparison and worklist-management algorithm. Without fingerprinting, the only way to compare the states of two processes is to do a direct comparison of process memory. Moreover, we no longer have a convenient means of indexing into a table of previously seen states. Consequently, the state manager must retain a process for every state previously seen, and must compare every newly created process state with every retained state, comparing full address spaces. In contrast, in the fingerprint-based approach, we only need to store the 128-bit fingerprint; any process that does not have outstanding worklist entries can be garbage-collected.

To measure memory usage, the generating extension tracks the number of pages in use across all processes in the generating extension. Because all processes must be retained when fingerprinting is disabled, determining the memory usage across all processes is straightforward: it is the sum of all live pages across all processes. When fingerprinting is used, memory usage is the maximum number of live pages at any given point

in the program's execution. To evaluate the execution time of a generating extension, we time its end-to-end execution, from the beginning of the first basic block to the end of the last basic block.

We allowed the generating extensions to run end-to-end for the "real-world" examples. However, for the microbenchmarks, we added a time-out after 90 minutes of specialization

Experiment Timing As with GenXGen[C], timings are collected using RDTSCP, and I collected the 10% trimmed mean over 100 trials, as described in §6.2.2.

Evaluation Platform. The host hardware configuration is identical to the one described for GenXGen[C] in §6.2.2: a 2.5 GHz, 4-core, Intel Core i7 CPU (model 4870HQ), and the guest VM was allocated the same proportion of hardware resources. However, the software configuration differed. The guest OS ran as a VMWare Fusion 11 VM running on Mac OS 10.13. The guest OS was Slackware Linux 14.2, running a modified version of the 4.4.12 Linux kernel. The kernel is modified, because GenXGen[mc] was implemented prior to the decision to use eBPF to interpose on the pagefault handler. Instead, a small amount of code was added to the kernel to (i) implement a data structure for storing the CoW page-fault history for processes, and (ii) implement a system call to extract all virtual addresses of CoW faults for a process ID since `fork`.

6.4.3 Evaluation

This section discusses the evaluation of each of the experimental questions for GenXGen[mc], and presents our findings.

		Generating-extension performance [CoW,Fingerprint]				Execution time		Speedup
		[no,no]	[yes,no]	[no,yes]	[yes,yes]	orig. prog.	resid. prog.	(Slowdown)
gnu-wc[8]	time	45m 36s	50m 11s	2.755s	1.190s	283 ± 7.8 μs	106 ± 5.1 μs	2.67x
	pages	146901	46	2129	9	—	—	
lzfx[8]	time	2m 52s	2m 38s	1.065s	.167s	.8 ± 0 μs	.3 ± 0 μs	2.67x
	pages	76664	36	8516	1	—	—	
printf	time	68m 23s	66m 11s	6.138s	.744s	90.6 ± 5.1 μs	77.7 ± 4.8 μs	1.16x
	pages	240577	48	12774	6	—	—	
gnu-env	time	13h 12m	12h 26m	15.692s	3.332s	36.5 ± .1 μs	31.0 ± .1 μs	1.18x
	pages	958050	129	2129	2	—	—	
power	time	74m 39s	64m 36s	2.241s	.679s	3.1 ± .1 μs	2.9 ± .1 μs	1.06x
	pages	221416	102	2129	1	—	—	
dotprod.	time	>90m	>90m	11.366s	2.364s	3.3 ± .1 μs	2.7 ± .1 μs	1.22x
	pages	—	—	2129	1	—	—	
interp.	time	>90m	>90m	13.638	6.186s	35.9 ± .2 μs	36.2 ± .1 μs	(1.01x)
	pages	—	—	2129	1	—	—	
filter	time	>90m	>90m	16.391	6.370	3.8 ± .1 μs	2.6 ± .1 μs	1.46x
	pages	—	—	4258	2	—	—	
str_match[9]	time	28m 30s	26m 13s	1.652	.839	51.1 ± .1 μs	38.8 ± .1 μs	1.32x
	pages	185223	22	31935	3	—	—	
sha1	time	>90 m	>90 m	24.223	11.783s	4.6 ± 0 μs	3.3 ± 0 μs	1.39x
	pages	—	—	2129	2	—	—	

Figure 6.20: Run times and space usage for each generating extension, with and without CoW/fingerprinting. Run times for original and residual programs are also included, with 95% confidence intervals ("—" means "not measured.")

[8]I performed these experiments at a small scale, comparable to the original WiPER paper, and did not perform the scaling experiments that I performed later with WiPER. Consequently, the speedups reported here over-weight the elimination of the boilerplate setup code at the start of the program, which would take on diminishing importance as the size the dynamic input increases.

[9]I revisited this result to obtain a more detailed comparison of performance between the original and residual program. Due to changes in the GenXGen runtime over the past year, I was not able to re-run the machine-code generating extension directly. However, I do have the BTA results, and can manually reconstruct the program that would be produced. I ran the hand-constructed program to produce the results in the two rightmost columns. I also re-ran the original program to ensure the comparison and speedup was fair across VM states.

6.4.3.1 Answer to RQ1-MC: What are the individual improvements contributed by CoW and fingerprinting to memory usage by a generating extension?

The experimental results for RQ1-MC are presented in the *pages* rows of Fig. 6.20. Both fingerprinting and CoW play a significant role in reducing memory usage. Using CoW, however, yields the most significant reduction for every application This improvement is due to the fact that for all ten applications, the instructions that are evaluated during generating-extension execution perform the majority of their writes within a single stack page. Even when fingerprinting is not used, CoW ensures that the number of pages needed to retain all previously visited states is small, roughly the number of basic blocks that executed at least one memory write. (See the column labeled "[yes,no]".).

Findings.

For all programs, CoW yields a pronounced improvement, because the programs used to test GenXGen[mc] write a relatively small number of pages in their overall live address space, and thus there is a high degree of sharing across all processes.

Hashing also produces a smaller, but still significant decrease in page usage, by allow garbage collection, but the effects of garbage collection are lessened by the high degree of sharing between state processes.

6.4.3.2 Answer to RQ2-MC: What are the individual improvements contributed by CoW and fingerprinting to the time taken by a generating extension to emit a residual program?

Fingerprinting plays the most ignificant role in reducing generating-extension execution time. This result is unsurprising, because, without

fingerprinting, the amount of time needed to identify whether a state has been previously visited scales linearly with the number of states previously visited. Thus, the execution time scales quadratically with the number of states. In every case, when hashing was was used, specialization was completed in under 20 seconds, with all but one program, `sha1`, completing in under 10 seconds. In seven of the ten cases, hashing improved specialization times by over an hour, and in the case of `gnu-env`, a specialization task that took over twelve hours without hashing took only 3.33 seconds with hashing.

Using CoW also improves the execution times of generating extensions; the improvement is most pronounced in the case where fingerprinting is also used. When fingerprinting is used, but CoW is not used, the overhead of copying an entire process begins to dominate the execution time of the generating extension.

For the `gnu-wc` generating extensions without fingerprinting, the execution time with CoW enabled was greater than when CoW was disabled. We do not have a full explanation why, but we believe that the extra cost is due to the cost of collecting memory-usage data. When we measure memory usage with CoW enabled, we track every process currently using a given page. For certain workloads, especially when fingerprinting is not used—and thus page mappings are retained for every state visited—the cost of maintaining this data structure may become relatively large.

Findings.

Hashing yields a significant speedup over the non-hashed state-management strategies. In particular, because non-hashed strategies require $O(N^2)$ comparisons of live memory pages, hashing allows specialization tasks that would otherwise take hours to complete in seconds.

By eliminating spurious page copies, CoW also yields modest specialization-time improvements—on the order of seconds—though not the pronounced asymptotic improvement seen with hashing.

6.4.3.3 Answer to RQ3-MC: Compared to the original subject program, how much does specialization speed up execution?

For RQ3-MC, the results are presented in the three rightmost columns of Fig. 6.20. Specialization produced a speedup in all but one program. `Dotproduct` and `power`, however, exhibit larger speedups relative to GenXGen[C], with `dotproduct` exhibting a 1.22× speedup, which contrasts with the .28× speedup obtained with GenXGen[C] (cf. Fig. 6.9). When specialized with GenXGen[mc], `str_match` exhibits a 1.35× speedup (versus the 1.2× speedup obtained with GenXGen[C]). These speedups are attributable to the difference between lifting strategies in GenXGen[C] and GenXGen[mc], and the finer-grained specialization available when performing machine-code specialization.

The difference in lifting typifies the opportunities for better optimization available to GenXGen[mc]. Consider the specialization of the following line of code in `str_match`, where `pat` is the static pointer into the static pattern string, and where `pat` references `'h'`:

```
if(*s == *pat){
```

The generating extension created using GenXGen[C] does not produce `if(*s == 'h')`, but instead emits code to lift the reaching definition of `pat`, as well as the reaching definition of the item located at `pat`. To avoid conflating the effects of compiler optimization with the performance effects of specialization, recall that the residual C programs are compiled at the `O0` optimization level—that is, no optimization is performed. Thus, the emitted machine code is

```
i1: mov eax, [<address of pat>]  ;eax := pat
i2: mov dl, [eax] ;dl := *pat
i3: mov eax [<address of s>] ;eax := s
i4: mov al, [eax]; al := *s
i5: cmp dl,al
```

However, when the generating extension created using GenXGen[mc] specializes the binary for `str_match`, the instructions `i1` and `i2` are static, and the contents of `dl` are lifted, instead yielding the following residual code:

```
i': li dl, 0x68 ;dl := 'h'
i3: mov eax [<address of s>] ;eax := s
i4: mov al, [eax]; al := *s
i5: cmp dl,al
```

This code has two fewer instructions that load from memory, and hence avoids two trips to the CPU cache that the C version performs, yielding a smaller and faster inner loop than GenXGen[C]. This inner loop is repeated once for every offset in the subject string until a match is found, and hence the speedup of the overall program is proportional to the speedup of the inner loop.

The dramatically different results for `dotproduct` — a 1.22× speedup with GenXGen[mc] versus a .28× speedup with GenXGen[C] — is due to the fact that the vector length is static, and thus GenXGen[mc] can lift vector-indexing computations at the *register* level, eliminating all loads and stores of index variables. Due to reaching-definition lifting in GenXGen[C], the residual program's use of the vector index variable still entails a memory access. In addition, GenXGen[mc] can lift the loads from the static operand vector to a register, eliminating a memory read for every multiplication. Moreover, this effect is amplified relative to GenXGen[C], because GenXGen[c]'s lifting occurs at the byte-granularity.

The specialization of `filter` significantly optimizes the inner loop of the image-filtering procedure, eliminating the `if` statement that selects which image filter is applied to each pixel, as well as inlining loads from lookup tables that encode properties of the selected filter algorithm.

Both `gnu-wc` and `lzfx` have a speedup of 2.7x. In the case of `gnu-wc`, specialization eliminates the argument-parsing loop, as well as setup code that (i) sets locale information and (ii) obtains system-dependent configuration information. Similarly, the specialization of `lzfx` eliminates initialization code, in addition to inlining several functions that cannot be completely eliminated.

`printf` and `gnu-env` enjoy more modest speedups, roughly 16-18%. Most of these speedups is due to the unrolling of the core loop in each program: the format-string-parsing loop in `printf`, and the argument-parsing and environment-setup loops in `gnu-env`.

`sha1` obtains a 1.4x speedup from the elision of loads inside the main loop, along with the elision of the initial code that initializes the static data.

However, `interpreter` experiences a slight slowdown (1.01x), possibly due to the effects of aggressive unrolling on cache performance.

Findings.

GenXGen[mc] yields larger speedups than GenXGen[C]. This outcome is because GenXGen[mc] can lift registers, and thus can eliminate memory operations that GenXGen[C] cannot eliminate. In cases where static memory operations are a large proportion of a subject program's execution time, such as in tight inner loops, GenXGen[mc] can yield significant improvements in execution time. In addition, the speedups seen in `lzfx` and `gnu-env` suggest that the procedure inlining available to GenXGen[mc] can also play a role in improving performance.

The slight slowdown for `interpreter` suggests that unrolling's effects on instruction-cache locality may still offset other performance benefits.

6.4.3.4 Answer to RQ4-MC: How does specialization affect the size of a program in instructions? Moreover, for the subset of programs that correspond to feature-removal tasks, what is the degree of debloating performed?

To evaluate RQ4-MC, we measured the number of instructions in the original and residual versions of each program. The changes in the number of instructions are reported in Fig. 6.21. Specifically, we measured the number of instructions reachable from the body of `main`, including the text of `main`. The microbenchmarks `power`, `dotprod`, `interpreter`, `filter`, and `sha1`, are all "loop-unrolling" tasks, and thus experience a significant growth in program size when specialized. Similarly, in the specializations of `str_match` and `printf`, the respective core loops are unrolled according to the values of the sought substring and the format string, respectively, enlarging both programs. The specializations of `lzfx` and `gnu-wc` extract a single feature from a collection of "subprograms," and thus reduce

program	original	residual
gnu-wc	2969	1326
lzfx	1986	1094
printf	754	1038
gnu-env	1820	1123
power	30	323
dotprod.	307	1123
interp.	146	558
filter	287	1207
sha1	332	2823
str_match	34	410

Figure 6.21: Comparison of the number of instructions in the original and residual programs.

program	no. procedures in orig.	no. procedures in resid.	no. call-sites in orig.	no. call-sites in resid.
gnu-wc	16	4	125	24
lzfx	15	9	63	24

Figure 6.22: Comparison of the number of procedures and call-sites in the original and residual programs for the feature-removal examples.

the size of both programs. Although `gnu-env` is a loop-unrolling task, a large amount of error-handling code—which checks the static input for correctness—can be eliminated, yielding a net size reduction for the inputs that we supplied.

To evaluate RQ4-MC with respect to feature-extraction abilities, we examined a subset of the programs used to evaluate feature extraction in WiPER. In particular, we worked with `gnu-wc` and `lzfx`—specializing `gnu-wc` to only count words, and `lzfx` to only perform compression.

One effect of feature extraction on `gnu-wc` and `lzfx` is already shown in Fig. 6.21, which gives the decrease in the number of instructions in the specializations of both programs. In addition, Fig. 6.22 shows the decrease in the number of procedures and call-sites.

When `gnu-wc` is specialized to produce a residual program that only counts words, the number of instructions is reduced by 55%, the number of procedures is reduced by 75%, and the number of call-sites is reduced by 81%. `Gnu-wc` is organized so that the procedure in which the bulk of the work is performed has three disjoint loops, depending on whether words, characters, or lines are to be counted. The size reduction for `gnu-wc`—in all size metrics—is due to specialization removing the two unneeded counting loops, as well as initialization code not needed for counting words. The residual code consists of the remaining initialization code, and auxiliary arithmetic-utility procedures shared among all three loops.

When `lzfx` is specialized to produce a residual program that only performs compression, the number of instructions is reduced by 74%, the number of procedures is reduced by 40%, and the number of call-sites is reduced by 62%. Because `lzfx` has several utility procedures that are shared between the compression and decompression routines, the reduction in the number of instructions is proportionally larger than the reduction in the number of procedures. For example, if the compression loop is removed, the shared procedures still remain.

In both cases, specialization is able to significantly reduce the complexity of the subject program, by eliminating all unused procedures, as well as all call-sites not used in the residual program.

Findings.

The loop-unrolling tasks, yield a significant increase in instruction counts.

For the two feature-removal programs, `gnu-wc` and `lzfx`, 55.3% and 44.91% of instructions are removed, respectively. Both programs also exhibit a significant reduction in the number of procedures and the number of call-sites as well, with 75% of `gnu-wc`'s procedures removed and 40% `lzfx`'s procedures removed. In both cases, over half of all call-sites are removed.

Chapter 7

Related Work

This chapter discusses related work not covered in previous sections, and provides additional detail about several works discussed earlier. The related work can be divided into five categories:

1. Specialization of C and LLVM bytecode (§7.1), which includes techniques for specializing C programs either via a source-to-source transformation, or as a transformation of LLVM IR.

2. Specialization of machine code (§7.2), including techniques for specializing both x86 machine code and JVM bytecode.

3. Manipulations of memory snapshots (§7.3), including prior memory-hashing techniques, and mechanisms for canonicalizing memory states to detect duplicates.

4. Recording states (§7.4), which discusses record-and-replay debugging techniques, which provide mechanisms for saving and restoring states.

5. Symbolic and concolic execution (§7.1), two program-analysis techniques, which like classical specialization, are explorations of a rep-

resentation of a subject program's state-space, and hence must save and restore state representations like classical specializers.

7.1 Specialization of C and LLVM Bytecode

There have been a number of systems that apply partial-evaluation techniques to C, including C-Mix [Andersen, 1994b; Makholm, 1999], 'C [Engler et al., 1996], DyC [Grant et al., 2000], and Tempo [Consel et al., 2004]. Other systems, such as LLPE [Smowton, 2014], TRIMMER [Sharif et al., 2018b; Ahmad et al., 2021], and LMCAS [Alhanahnah et al., 2022] can be applied to C programs indirectly, by first compiling C source code to LLVM bit-code.

The influences of C-Mix on GenXGen, and the differences in the techniques used in the two systems, have been covered thoroughly in the body of the dissertation.

'C, DyC, and Tempo support dynamic code generation, which makes them not classical partial evaluators *per se*. Because the specialization of (a portion of) a program is performed at runtime, they avoid a costly compilation step by emitting residual code in the form of machine code.

With 'C, the user creates code generators via a DSL with primitives to specify dynamically generated code that can be statically type-checked. The burden is on the user to avoid redundant states and ensure that code generation terminates.

In DyC, the algorithms for identifying specialization opportunities use many of the concepts employed in classical two-stage (off-line) partial evaluators—i.e., partitioning program elements into static/dynamic sets, and specializing as needed along all control-flow paths. User annotations specify the variables for which portions of a program should be specialized, and a static pass then equips the program with generating extensions that are invoked, at run time, to create the specialized code (as machine code).

Tempo supports run-time specialization in a manner similar to DyC, using generating extensions, but also supports classical compile-time specialization, data specialization [Chirokoff et al., 1999], and their combination. There are differences in the approaches that Tempo and GenXGen take to lifting. For instance, the Tempo paper says [Consel et al., 2004, §2.1.2], "When a static expression occurs in a dynamic context, the value of this expression must be residualized. Nevertheless, some values are non-liftable, i.e. they cannot be meaningfully represented in the specialized code. A pointer to a local variable is always non-liftable. When specialization is carried out at compile time, pointers to global variables are also non-liftable. Floating-point numbers may be considered non-liftable, because of the difference between the precision of the textual and internal representations."

Pointers to local variables are often used in C to simulate call-by-reference, and GenXGen does support lifting pointers to local variables, as described in §5.1.5. With respect to floating-point numbers, GenXGen lifts all values (except pointers) byte-by-byte, so there is no loss of precision. (Admittedly, this approach does not support changing architectures or compilers between the generating extension and the residual program.)

TRIMMER and LMCAS are partial evaluators for LLVM bit-code. Both rely on LLVM optimization passes, such as loop-unrolling and constant propagation, rather than performing classical partial evaluation. By doing so, they avoid the need for a general-purpose state-management strategy, but value propagation is limited to local and global variables.

LLPE performs online partial evaluation of LLVM bit-code. It functions more like an interpreter over partial programs. LLPE also merges partial states—i.e., after "if(...) { $x = 1; y = 5;$ } else { $x = 2; y = 6;$ }" one would have a merged state in which x is either 1 or 2 and y is either 5 or 6—which has an effect on the ability to specialize the program downstream. State merging would not be compatible with our approach of exploiting OS-level

primitives and operating on native hardware states. (In practice, LLPE requires a substantial amount of hand tweaking by the user [Smowton, 2020].)

Not all of these systems share the same goals as GenXGen. However, in principle, the three systems that employ generating extensions—C-Mix, DyC, and Tempo—could benefit from the two contributions of our work, namely, (i) a new technique for state management in a generating extension that runs natively (§3), and (ii) a new software architecture for generating extensions (§4.4). For one thing, the papers on DyC and Tempo make no mention of handling heap-allocated storage, and Andersen's thesis on C-Mix states [Andersen, 1994b, §3.10.7], "Since heap-allocated data structures tend to be rather large, copying and comparing of heap-allocated data structures should be kept to a minimum. We have currently no automatic strategy for when side-effects on heap allocated objects should be allowed." The approach presented in this dissertation supports programs that use and destructively update linked data structures, with no need to perform a mark-and-sweep traversal to capture program state.

7.2 Specialization of Machine Code

Run-time code generation is a generating-extension-like approach to program specialization that produces machine code on-the-fly during program execution. Unlike our approach to machine-code specialization, which operates on stripped *binaries* without source code or symbol-table information, run-time code generation systems take user-annotated *source code* as input and perform BTA and generating-extension construction as part of compilation. In the Fox [Leone and Lee, 1996; Lee and Leone, 1996] and Lancet [Rompf et al., 2014] systems, type-level information in the source code is exploited to produce run-time machine-code generators. These systems avoid the state-management issues from §3 by exploiting

the availability of high-level semantic information from the source language. In contrast, Klimov [2009] describes a run-time code generator for Java bytecode that does not rely on information from source code. However, Klimov can only determine state equality for programs that do not use the heap; the approach identifies semantically identical states based on structural properties of Java Virtual Machine heap configurations. JIT compilation [Aycock, 2003] is an example of run-time code generation in widespread use. However, because it is performed at run-time, the emphasis is on recouping the cost of translation, which limits the kinds of optimization techniques that can be performed.

Turning to interpretation-based approaches, WiPER is a partial evaluator for x86 binaries. WiPER uses CodeSurfer/x86's semantic models of the 32-bit x86 instruction set to evaluate instructions. WiPER represents states using an applicative-map-based data structure that does not use hash-consing. Thus, state equality is determined by directly comparing the contents of the data structure.

Although GenXGen[mc] is quite different from Fox [Leone and Lee, 1996; Lee and Leone, 1996] in most respects, their use of *pseudo-instruction macros* inspired the approach to constructing machine-code generating extensions used in GenXGen[mc]. GenXGen[mc] uses similar macros to produce residual assembly code, but extends the approach to include various other state-management actions.

7.3 Manipulations of Memory Snapshots

A huge number of PL techniques rely on recording and manipulating snapshots of memory states; thus, we can only highlight a few items of related work.

Patent US7469362B2 [Hudson et al., 2008] uses hashes of the stacks of a program's threads to support associating failures to known root causes.

This technique has a number of differences with the hashing method used in GenXGen. First, it only hashes stack memory, ignoring code, globals, and heap-allocated storage. Second, it deliberately leaves out some information "... to minimize the affect of patches and minor changes to the code." Third, the hash is performed as a batch computation; in contrast, GenXGen uses incremental hashing.

Model checking is a method to check properties of programs statically by exploring the state space of a transition system. To achieve acceptable performance, model-checking algorithms must avoid exploring redundant states, and Rabin's fingerprinting technique has been used to implement incremental hashing of program states in SPIN [Nguyen and Ruys, 2008] and StEAM [Leven et al., 2004; Mehler and Edelkamp, 2006]. However, those systems are *interpreters* (for Promela and assembly language, respectively). In contrast, incremental hashing in GenXGen uses OS-level primitives to perform incremental hashing to hardware states, with code that *executes natively*.

Musuvathi and Dill [Musuvathi and Dill, 2005] developed an incremental heap-canonicalization algorithm, which allows states to be detected as duplicates when they differ only in the addresses of heap objects. There is no analog of heap canonicalization in GenXGen; moreover, heap canonicalization would appear to be incompatible with our approach of exploiting OS-level primitives and operating on native hardware states.

The runtime environment for our generating extensions was originally designed for GenXGen[mc]. Unfortunately, as described in §8.2, the technology for very precise static analysis of machine code does not scale well, and limited the applicability of GenXGen[mc] to small binaries. In contrast, because GenXGen[C] works on source code—and employs less fragile static-analysis techniques—it is capable of specializing larger pieces of code than our earlier system.

7.4 Recording States

Program execution record-and-replay debugging has been the subject of an extensive body of research, dating back to EXDAMS [Balzer, 1969], and subsequent systems such as PTRAN [Choi and Stone, 1991] and FDR [Xu et al., 2003], and instantiated in contemporary debuggers such as rr [O'Callahan et al., 2016, 2017], Microsoft Visual Studio's debugger [Barr and Marron, 2014], and gdb [gdb, 2023, Process Record And Replay] In record-and-replay debugging it is necessary to record states for possible replay in reverse. The needs for record-and-replay debugging are somewhat different than what is needed in our context. First, for record-and-replay debugging it is necessary to log all of the changes. In our work, the changes do not need to be logged because they are needed only to update the hash-value, and the actual changed state is the state of a process controlled by the OS. Second, record-and-replay debugging works with a strictly linear history of states, whereas during specialization, the states in the worklist are the frontier of a tree of states; the states at the interior nodes are discarded, although hash values for all such states are retained.

7.5 Symbolic/Concolic Execution

Partial evaluation has some resemblance to symbolic execution [King, 1976]. Moreover, our `fork`-based method for managing partial states was inspired by the state-management mechanism in the EXE symbolic-execution system [Cadar et al., 2006]. As originally formulated, partial evaluation [Futamura, 1971; Jones et al., 1993] differs from symbolic/-concolic execution [King, 1976; Godefroid et al., 2005] in several ways. Symbolic/concolic execution tools are *bug-finding* tools that attempt to identify bugs by driving a program's execution to a specific subset of the state space. These tools implement *path-exploration* algorithms that use

calls to constraint solvers to drive the program down as many new paths as possible within some time limit.

The closest connection is actually between symbolic execution and a generalization of partial evaluation, called *generalized partial computation* (GPC) [Futamura, 1988; Futamura et al., 1991] in the partial-evaluation community. GPC extends classic partial evaluation to track constraints on the dynamic portion of a partial state. A method similar to GPC was developed by Coen-Porisini et al. [Coen-Porisini et al., 1991] who applied symbolic execution to the problem of program specialization. With respect to our work, the connections are weaker: while GPC is a more powerful program-transformation method than partial evaluation, the required manipulations of the symbolic state during GPC appear to require an *interpreter-based* approach, and thus are not compatible with our approach of exploiting OS-level primitives, and operating on native hardware states.

A different, but weaker, connection has to do with *partitioning* in states. In partial evaluation, states are partitioned into static and dynamic variables. Partial evaluators do not track any information about the dynamic (i.e., non-concrete) parts of partial states; the congruence property of the BTA algorithm ensures that the dynamic state is never needed to update static variables. In concolic execution [Godefroid et al., 2005, 2008], states are partitioned into concrete and symbolic variables; as in symbolic execution, the symbolic part of the state is represented with symbolic values, which are logical formulas in some theory. A concolic-execution engine attempts to construct a symbolic approximation of the values in the symbolic (i.e., non-concrete) portion of the state at every program point. In other words, it attempts to leverage, to the extent that it can, precisely the portion of the state that a classic partial evaluator ignores. Again, such techniques appear to require an *interpreter-based* approach, in contrast with our approach of operating on native hardware states.

Bubel et al. [Bubel et al., 2009] have exploited the complementary

capabilities of symbolic execution and partial evaluation to improve the performance of a program-verification tool.

Chapter 8

Conclusion

In this thesis, I described GenXGen, a tool for specializing programs written in low-level[1] languages such as C and x86 assembly. GenXGen consists of two main components: (i) a generating-extension generator, which produces program-specific specializers, and (ii) a language-agnostic generating-extension runtime that efficiently implements the state-space traversal necessary for classical program specialization. More concretely, the generating-extension runtime supports the implementation of generating extensions for any low-level language that implements the System V AMD64 ABI [Michael Matz, 2012], or other similar conventional C/Unix-style ABI. The techniques used in (i) are also language-independent, although they require a means for computing slices of programs written in the language that the specializer is to support.

[1] I use "low-level" here—for lack of a more accurate term that is equally concise—as a shorthand for a general class of languages, such as C and x86 assembly, which do not have a sophisticated runtime environment, and do not hide information that is traditionally considered "hardware-level," such as pointers. The term "low-level" is, at best, contentious [Chisnall, 2018] and of dubious accuracy in the context of modern CPUs. Even the interface exposed by the x86 ISA, which persists because of the unavoidable necessity of backwards compatibility, abstracts away microarchitectural details that are, especially in a post-Spectre/Meltdown world, of significant importance.

8.1 Contributions

In implementing GenXGen, I made several contributions. To implement (i), the generating-extension generator, I made the following contributions:

- **I implemented a polyvariant binding-time-analysis algorithm** using a technique from the slicing literature, specialization slicing [Aung et al., 2014]. I implemented a *slice-materialization algorithm*, which converts a program with polyvariant slice results (i.e., a program with multiple slice results for a given procedure) to a program with monovariant slice results (i.e., a program with a new copy of a procedure `P` for every slice result for `P` in the polyvariant result), allowing the remaining passes of GenXGen to use a conventional ge-gen algorithm and generating-extension runtime that expects monovariant BTA results.

- **I implemented a build-tracing tool that allows GenXGen to produce generating extensions for real-world software projects** built using tools such as `make`. Prior systems based on generating extensions, such as C-MixII[Makholm, 1999], did not handle non-trivial software projects in a satisfactory manner, and required hand-made makefiles to produce generating extensions. My build-tracing tool traces builds with hundreds of source files, allowing GenXGen to successfully produce generating extensions for real-world programs, such as Busybox applets.

To implement (ii), the language-agnostic generating-extension runtime, I made the following contributions:

- Inspired by program-analysis tools such as KLEE [Cadar et al., 2008], **I implemented a process-based representation for hardware-level program states**. A Linux process is a perfectly usable representation of the state of a low-level program; one can obtain efficient saving and

restoration of program state through a standard context switch. This approach obviates the need to impose any interpreter-like operations on the static-program portion of a generating extension created using GenXGen: the statically executing portion of the subject program executes natively.

- **Eschewing the traditional PL insistence on soundness, I implement an incrementally updatable hashing-based mechanism for identifying redundant program states in $O(1)$ time.** Though nominally unsound, I show that the probability of a generating extension producing an erroneous residual program is negligible: at most 2^{-157} for a reasonable choice of parameter settings. Moreover, because I chose to use processes to represent program states, I can exploit the underlying CoW mechanism provided by the OS to identify state-changes at hardware-page granularity; consequently, GenXGen's generating extensions need only incorporate changed pages into the hash.

- **I evaluated GenXGen's performance on microbenchmarks and real-world programs**, which showed that GenXGen can perform real-world specialization tasks on non-trivial Linux programs. In practice, to create a version of a command-line program with unused features removed (for example, creating a `base64` program that contains only the base 64 encoder), specialization completes in under a second. I also show that GenXGen is capable of performing significant "debloating" in these feature-removal tasks.

8.2 Limitations and Challenges

My goal with GenXGen was to produce a "turnkey" generating-extension framework for real-world programs. Initially, I began with the goal of pro-

ducing generating extensions for stripped executable binaries, i.e., binaries with all debugging symbols removed. To produce the SDG representations needed to perform slicing-based BTA and the CFG structures needed produce the generating extension, I used CodeSurfer/x86 [Balakrishnan et al., 2005b], a version of CodeSurfer [codesurfer, 2018; Anderson et al., 2003] with binary analysis capabilities. However, the technical challenges in producing slices suitable for BTA of real-world command-line programs, such as the ones in GNU Coreutils, exposed several limitations in CodeSurfer/x86's binary-analysis features.

8.2.1 The Limitations of Library Modeling

In the typical case, to avoid producing binary programs with a large amount of duplicated library code, x86 binaries on Linux server and desktop systems are *dynamically linked*. That is, many of the general-purpose systems-level libraries, such as the C standard library, are not part of the program binary. Instead, when a dynamically linked program is run, a program known as a *loader* finds for each dynamically linked library a *shared-library* file containing the system's library implementation. Moreover, many library procedures invoke system calls, and thus such a library procedure's behavior cannot be analyzed from the contents of the binary alone, even if it were statically linked. Because of this, library code may not be present in the binary, and thus unavailable for analysis.

To address this issue, CodeSurfer/x86 uses library models, which consist of C code that model relationships between input and output parameters, while being simpler to analyze than the full source code of library procedures. However, these model procedures sometimes lack the accuracy required for performing slicing-based BTA. For example, the CodeSurfer representation of abort procedures such `exit` do not correctly model the effect of program termination on global variables. For example, `errno` is a parameter to many abort procedures, and if `errno` is in the

slice before a call to one of these procedures, `errno` is in the slice after the (*impossible*) return from the abort procedure. Thus, dependencies are propagated through procedures that do not actually return.

Moreover, in practice there are simply too many software libraries, and one cannot reasonably expect CodeSurfer's developers to provide models beyond, e.g., `libc`. Thus, for many real-world programs, the default CodeSurfer representation contains calls to procedures that lack a representation (by default). Because of this deficiency, and the inaccuracies in the existing C library models, it was often necessary to supply additional library models for each program added to the suite of programs that I tried to use for evaluation. Dealing with this situation required significant debugging and development effort to identify deficiencies in existing library models, and to develop reasonably accurate replacement model procedures or to develop new model procedures for unimplemented models.

8.2.2 The Limitations of Binary Slicing

CodeSurfer/x86 [codesurfer, 2018; Balakrishnan et al., 2005b; Anderson et al., 2003] does an admirable job of slicing stripped binaries. That CodeSurfer/x86 is as capable of providing such high-quality slices as it is is a testament to the multiple Ph.D. theses worth of research underpinning its implementation. However, in practice, the general task of constructing generating extensions for stripped binaries is one that runs up against the limitations of CodeSurfer/x86's capabilities. In the standard framing, (forward) program slicing answers the following question: given a program P, and a set S of program point/variable pairs, (p, v) what program points in P depend on S? For a binary program P that has had all debugging information stripped, simply answering the question "what are the program points of P?" is already undecidable in the general case. Identifying "variables" to slice with respect to is even harder; answering

that question requires performing not just disassembly, but one of the hardest aspects of *decompilation*: recovering program variables.

CodeSurfer identifies *abstract locations*, which include variables, heap objects, structure fields, arrays and their constituent elements: roughly speaking, anything that is a valid lvalue in C. To identify abstract locations, CodeSurfer performs *value set analysis* [Balakrishnan, 2007], or VSA, a type of abstract interpretation. Informally, for every variable/program-point pair (v, p) in a program, abstract interpretation computes a conservative overapproximation of all values v can take on at p. The approximation is computed in an *abstract domain* that (generally) encodes a concise summary of all possible values in the over-approximation. That is, if $L_{(p,v)}$ is the set of concrete values that v could take on at p, abstract interpretation[2] computes an abstract summary $S = \alpha(L_{(p,v)})$ such that given $\gamma(S)$, where γ maps abstract summaries to the set of concrete values they represent, $L_{(p,v)} \subseteq \gamma(S)$. The hope is that, for a given program-analysis task, with a good choice of abstract domain, even if $L_{p,v}$ cannot be computed exactly, a useful overapproximation in the abstract domain may still be effectively computable.

The abstract domain of VSA is constructed so that it represents the collections of integer values that typically characterize addresses used for memory accesses in real-world programs. For example, consider the following code:

```
struct s {
  int first;
  int second;
  int third;
```

[2]This description is a slightly simplified one, and not representative of the full breadth of analyses possible with abstract interpretation: in general, abstract interpretation need not be concerned with the valuation of specific program variables: in general α is *some* kind of abstraction of program behaviors, and α can, for example, map program point p to an approximation of the set of program points that are reaching definitions for p.

```
};
struct s arr[33];
void init(){
  for(int idx = 0; idx < 33; idx++){
    arr[idx].second = 10;
  }
}
```

When compiled to machine code, the instruction that writes `10` to the `second` field for each element in `arr` will be something like this:

```
mov [rax], 0xa
```

That is, the address of the `second` field of the struct at offset `idx` is stored in register `rax`, and the instruction writes `10` to the location at that address. Consider the set of addresses `rax` could take on at this instruction. Assuming four-byte integers, the first two addresses will be $\texttt{arr} + 4$ and $\texttt{arr} + 16$, because `struct s` is a 12-byte structure. In general, at this program point, `rax` takes on the values $\{\texttt{arr} + 12\mathrm{i} + 4 | 0 \leqslant \mathrm{i} < 33\}$.

The key observation is that memory access patterns in real-world programs tend to take on this "base plus stride within some bounds" pattern. Thus, VSA represents the values that variables[3] can hold as a triple of $(\text{lower bound, upper bound , stride})$ values. In practice, VSA can recover abstract locations for individual variables, heap objects, struct fields, and can also infer the layout of arrays.

The difficulty with VSA as it relates to slicing is due to the difficulty of identifying the bounds of the value-sets. In the presence of loops, identifying the bounds on a range of memory accesses is undecidable, and thus,

[3]The careful reader may note an apparent circularity here. The impetus for this discussion of VSA was *discovering* abstract locations. In practice, the analysis can be thought of as being "bootstrapped" with CPU registers as the only initial abstract locations, and, by interacting with several other analyses (described in Balakrishnan and Reps [Balakrishnan and Reps, 2010]), the overall analysis gradually discovers more abstract locations.

with infinite abstract domains, abstract interpretation may not converge without assistance. Even in finite domains, abstract interpretation may be slow to converge in the presence of loops. Thus, VSA, like other forms of abstract interpretation use *widening operators* to aid convergence in loops. These operators overapproximate the values that variables in a loop may take on. However, the overapproximation can be extremely coarse, with, e.g., the final VSA result for an array index in a loop taking on a lower bound of b, and an upper bound of $b + 2^{31}$.

Such results are problematic when using using slicing as a BTA. VSA can successfully recover, e.g., global variables and array bounds in practice. However, pointer arithmetic inside loops often induces spurious dependencies. Consider a program that, inside a loop, repeatedly writes through a pointer `p`, and that `p` always points inside of an array `a`. Widening, nonetheless, may yield results suggesting that `p` overruns the array bounds and overwrites, e.g. global variables, such as global flags that enable or disable features, or other utility values such as `errno`. Thus, spurious data dependencies between writes through pointers and global variables are common in practice, which erroneously taint large portions of a program as dynamic.

Obtaining slicing results for real-world binaries, even the smallest GNU Coreutils programs, is thus quite difficult. In practice, it requires enough understanding of the program's internals to identify the origin of spurious dependencies, and enough understanding of VSA to tune, e.g., widening parameters to avoid the worst-case behavior. Thus, in practice, simply getting slice results suitable for specialization from a single non-trivial real-world program often took several days or more of effort.

8.2.3 Limitations on CodeSurfer's C Rewriting

Because of the difficulty in producing generating extensions for larger and more complex x86 binaries, I decided to implement GenXGen[C]. The

availability of rich and accurate type information meant that producing accurate slices for large programs was much easier. However, my use of CodeSurfer/C to produce C generating extensions was somewhat of a mismatch with CodeSurfer/C's expected use cases. In particular, CodeSurfer's program representation is oriented towards program-analysis tasks, and less so for program rewriting. While CodeSurfer provided prototype rewriting APIs to manipulate program ASTs and CFGs, they are largely suited for rewriting the internal representations, and it is difficult to use CodeSurfer to rewrite a procedure `p` in a source file `f`, and produce a syntactically valid source file `f'` that is unchanged from `f`, except for the rewritten `p'`.

For example, the "null transformation" of simply taking CodeSurfer's representation of a procedure `p` and converting it back to syntactically valid C code fails in several corner cases, because the AST-printing API occasionally produces incorrect type specifiers for certain compound types. For program-understanding tasks, such as reverse engineering or manual refactoring, the output is perfectly comprehensible to a developer or analyst,[4] but is unsuitable for general-purpose program rewriting.

Moreover, because of several other AST-related issues, it was not feasible to do C-Mix-style lifting of this form:

```
printf("dynamic = dynamic + %d\n", static);
```

Moreover, it was also difficult to ensure that I could correctly produce, e.g., valid C99 structure literals. Thus, to expedite implementation, I chose to use the byte-granularity reaching-definition lifting for GenXGen[C].

In practice, due to the difficulty of obtaining other information relating to global variables, the source-to-source transformation in the ge-gen phase used information from CodeSurfer/C to produce individual procedures,

[4]The syntactically invalid C produced is also arguably a better textual representation than C's notably peculiar type-specifier syntax (see [Kernighan and Ritchie, 1988, §5.2]), and is easily produced from a linear walk of the type signature AST.

but the actual transformation of the original program's source files into generating-extension source at the project level was performed via a set of rewriting scripts implemented in a combination of Bash and Python.

8.3 Future Directions

8.3.1 Analysis and Improvement of Performance in Specialized Programs

In §6.3.4, I noted that specialized C programs often exhibited worse performance than the original program. Generally the degradation was only a few percent, though several programs exhibit significantly higher degradation in performance. For example, the residual version of `dotproduct` takes three times as long to execute as the original. This degradation is particularly striking when compared with the performance improvements seen in the residual programs produced by machine-code specialization described in §6.4.3.3. Additionally, `kmp`, which is a naive string matcher that specialization should transform into a program that is asymptotically faster than the naive string matcher, instead exhibits worse performance than the original program.

The structural properties of the residual versions of `dotproduct` and `kmp` lead me to hypothesize that there are three main issues causing the performance degradation seen in residual programs produced by GenX-Gen[C]:

1. **Reaching-definition lifting**, described in §4.4.3 and §5.1.6, addresses limitations in CodeSurfer's AST-rewriting and pretty-printing capabilities, described in §8.2.3, by introducing additional memory references into the residual program.

2. **Excessive loop unrolling** may harm instruction-cache locality, degrading program performance.

3. In the ordinary course of execution, a generating extension performs **control-flow destructuring**. As described in §2.1.3 and §4.4, a program is specialized basic-block-by-basic-block, and residual blocks are "stitched together" via goto statements. Moreover, because residual procedures may have returns corresponding to distinct static states, additional control flow is introduced to the residual program in the form of exit splitting (§5.3). Both of these aspects may degrade the effectiveness of the CPU instruction cache and branch predictor.

Performance Degradation in Dotproduct. In §6.4.3.3, I hypothesize that the performance degradation seen in `dotproduct` is due to issues (1) and (2). As described in §4.4.3 and §5.1.6, given the following code,

```
s = 5 + 5;
d = d + s;
```

where the boxed code is dynamic and the unboxed code is static, GenXGen[C] cannot produce the desired simplification of the dynamic line: `d = d + 10;`. Instead, the value of `s` after the assignment in the unboxed statement is lifted. Moreover, as described in §5.1.6, lifting is done at byte-level granularity, and the assignment is done through a reference to the base address of `s`. Thus, a GenXGen[C] generating extension would emit the following code:

```
*(s_base + 0) = 10;
*(s_base + 1) = 0;
*(s_base + 2) = 0;
*(s_base + 3) = 0;
d = d + s;
```

In the experimental evaluation, both the original and residual programs are compiled at `-O0`, and thus, the residual program's assignment to `s` is implemented as four operations, each of which must:

(i) Compute the address of `s` from `s_base`

(ii) Store one byte of `s` in memory.

In the compiled residual program, each of (i) and (ii) requires one instruction, each of which entails a memory access, so the residual assignment of the lifted value to `s` requires eight instructions that access memory.

Modern processor architectures are sufficiently complex that instruction count is, at best, a poor proxy for performance metrics. However, the increase in instruction count, where the introduced instructions are memory accesses, coupled with the sharp contrast with the results for the machine-code version of `dotproduct`, which exhibits performance *improvement*, suggests that excessive memory accesses are problematic from a performance standpoint. As described in §6.4.3.3, GenXGen[mc] is capable of performing much finer-grained lifting than GenXGen[C]; in particular, GenXGen[mc] can lift registers, potentially eliminating additional memory operations. Because one of the integer operands in the element-wise vector multiplication can be eliminated, an improvement in performance is reasonable to expect in the machine-code version. That result suggests that resolving issue (1) via a better implementation of lifting in GenXGen[C] is worth pursuing.

In addition, it may be the case that issue (2)—loop-unrolling—is detrimental to residual-program performance, and that GenXGen[mc]'s elimination of stores outweighs this effect in many cases, such as `dotproduct`, and thus the issue is more readily seen in the residual programs produced by GenXGen[C]. In particular, a large amount of loop unrolling may harm instruction-cache locality on the CPU and degrade performance, by forcing more reads from lower cache levels or system memory.

Performance Degradation in KMP. In contrast with `dotproduct`, the two string matchers, `str_match` and `kmp` are not subject to the byte-level lifting described in issue (1). The predominant lifted values in the residual matchers are type `char`, and thus one byte wide in C. Indeed, the naive string matcher, `str_match`, exhibits a performance *speedup* when specialized, while `kmp`, whose specialized version should be asymptotically faster, exhibits a significant slowdown. I hypothesize that the distinguishing factor is issue (3), the degree of control-flow destructuring performed in the specialization of `kmp`.

In particular, `kmp` matches the target-pattern string against the subject string at each offset using a carefully constructed `test` procedure and some additional state that, upon return from the `test` procedure, encodes the state of the finite-state automaton produced in the KMP algorithm. When specialized on a given target-pattern string, the residual basic blocks and procedure calls encode the transition relation of the finite automaton (see [Consel and Danvy, 1989] for a detailed discussion).

Critically, the residual procedure calls have control-flow paths that reach distinct static states, each of which corresponds to a finite-state-machine state. Thus, exit-splitting must be performed for `kmp`: each call to a residual version of the `test` procedure is followed by a switch statement, where each branch of the switch statement itself contains a goto targeting a basic block. Consequently, in the residual program, for each character in the dynamic input (i.e., the subject string), a procedure call is performed, as well as a return, a computed jump (i.e., the switch statement), and an unconditional goto. The top-level loop, which repeatedly calls the variants of this `test` procedure, thus consists of "spaghetti code": that is, it consists of a group of basic blocks that encode finite-state-machine states, and the residual program must repeatedly jump to and from blocks that are not adjacent in the text section of the compiled program.

This behavior is potentially problematic for performance. The utility of,

e.g., the instruction cache, is contingent on an assumption of locality, and the code destructuring performed by GenXGen[C] may yield a program that has poor cache behavior. Similarly, because specialization produces multiple variants of basic blocks and residual procedures, and also performs exit splitting, which introduces additional control-flow structures not present in the original program, the residual program may have significantly more control-flow locations, which may harm the performance of both the branch predictor and the branch-target predictor.

Evaluating the Hypotheses and Rectifying Performance Issues. If there was more time to devote to this research, I would carry out the following plan to evaluate these hypotheses. I would first rectify the byte-level-lifting issue described in issue (1). As described in §5.1.6, the primary reason for byte-level assignments was the difficulty in obtaining properly formatted structure literals. However, for standard numeric types—i.e., numeric types of standard hardware-supported sizes— one could implement reaching-definition lifting using a standard assignment of the given type.[5] In addition, the "lvalue lifting" through a reference to a variable base is only necessary for e.g., lifted array assignments and lifted assignments to structure fields. For cases where the original program's assignment to a variable simply assigns a full-sized value to the variable at offset 0, the use of the base pointer can be eliminated. With these fixes, the lifting of `s = 5 + 5`, for example, can be emitted as `s = 10`, thereby eliminating the introduction of memory accesses not present in the original program.

To evaluate whether hypotheses (1), (2), and (3) explain the performance degradation in specialized programs, I would also perform a wider variety of experiments and collect more detailed and robust performance metrics for the residual programs.

[5] In fact, for any type of, e.g., 4- or 8-byte width, lifting can be implemented by type-casting an assignment of an unsigned `int` or `long int`, respectively.

For issue (1), I would evaluate the performance of residual programs after the improved byte-level lifting is implemented. In addition, to gain an estimate of the best-case performance for lifting in GenXGen[C]—i.e., the lifting that could be done with access to full-fledged AST rewriting—I would manually construct simple generating-extension programs for the microbenchmarks, akin to the one pictured in Fig. 1.2, which perform the best-case lifts.

My hypotheses about issues (2) and (3) are predicated on the micro-architectural performance characteristics of the compiled versions of residual programs, and thus evaluation would require collection of fine-grained performance metrics about, e.g., cache hit rates, and the accuracy of branch and branch-target prediction. To collect these metrics, I would use the hardware performance counters made available in AMD and x86 CPUs. These counters are special-purpose registers that store a variety of fine-grained, per-core performance information, such as data- and instruction-cache misses at the various levels of caching, TLB misses, and branch mispredicts. Tools such as perf, Intel's vTune, and AMD's μProf provide access to these counters, along with other sophisticated microbenchmarking capabilities. To ensure accurate results, and to improve experimental timings, I would run these experiments in a non-virtualized context, and I would use the `chrt` command to force the kernel to run the profiled program with, e.g., a FIFO scheduling policy, which causes the CPU to run the program to completion without preempting it, thereby giving it exclusive access to the CPU. That approach should reduce variance in timing, and ensure better accuracy for the performance-counter measurements in the analyzed program.

Moreover, for the real-world programs analyzed, I would perform further benchmarking using perf to identify the most heavily executed portions of, e.g., the busybox applets. In doing this, I would hope to manually extract portions of the programs, such as time-consuming inner

loops, which could be used to construct further microbenchmarks for diagnosing performance issues caused by specialization.

If the hypotheses about issues (2) and (3) were confirmed, I would then modify GenXGen[C] to constrain the degree of *data-polyvariance*[6] in the residual program. Both the loop-unrolling and "spaghetti-code" properties arise from a single block being specialized with respect to many static states. One possible way of reducing the amount of reducing extreme data polyvariance would be via a re-rolling approach similar to that taken in systems such as Trimmer [Sharif et al., 2018a]. In this approach, for each original-program basic block b in a residual procedure P_v, the number of copies of b produced for P_v at specialization time is recorded. If the number of copies of any block exceeds a threshold, P_v is "re-rolled". Instead of containing multiple variants of each block b, a version of P_v is produced containing a single copy of each b reached. However, the underlying state-space exploration will still continue to be performed, to identify the set of blocks reachable in P_v. Because static state can no longer be incorporated into the blocks in P_v, all code is nominally dynamic for the purposes of code generation, and the original version of each block b is emitted verbatim (and linked to the others appropriately). Thus all static parameters to the re-rolled P_v must be lifted. This compromise solution constrains unrolling and the degree of available specialization, while still allowing for feature removal, because even though the residual blocks cannot be specialized further via lifting, only blocks reached in the generation of P_v are emitted.

Because a given data-polyvariant block b in P_v may call some procedure Q, and hence for each state v on which block b was specialized, multiple procedures Q_v will be emitted. Thus, if P_v is rerolled, the rerolling must

[6]Recall that in §4.2, there are two uses of the term "polyvariance" in the literature on specialization. One, which we are *not* concerned with here, is binding-time polyvariance, which is when there are multiple binding-time-analysis results for a procedure. Data polyvariance, on the other hand, is a property of the residual program—namely, there may be multiple versions of a residual bock or procedure, each of which is associated with the static state on which it was specialized.

be performed recursively "downwards" for each Q_v produced, and every Q_v called at a single block b must be unified into a single procedure.

8.3.2 Generalized Partial Computation

Chapter 6 exhibits GenXGen's ability to analyze and produce generating extensions for non-trivial Linux programs, and shows that the OS-assisted state-management techniques permit sub-second specialization times for complex specialization tasks. Moreover, as seen in §6.3.6, GenXGen is capable of performing feature removal/debloating on non-trivial real-world programs, such as `dos2unix` and `base64`. However, the lack of debloating performed for `od`, and the blow-up in program size for `od` seen in Fig. 6.16 in §6.3.5, demonstrates the limitations of classical program specialization in the presence of complex control that is dependent on dynamic input.

Examination of the difficulties that GenXGen has with interpreter-like programs, such as `od`, suggests that techniques from *generalized partial computation* [Futamura et al., 1991] may be useful for improving specialization. Generalized partial computation essentially extends classical partial evaluation by augmenting the static-state representation with logical predicates, and uses a theorem prover to further simplify code, and to further prune unreachable paths by using the additional state information to prove the unreachability of some dynamic branches.

For example, the blow-up in size of the residual code in `od` is due to multiple dynamic branches in series. In a simpler form, many of these series of dynamically-controlled `if` statements have structure like this:

```
if(dyn_cond1){
  v1 = 1;
}else{
  v1 = 2;
```

```
}

if(dyn_cond2 && dyn_cond1){
  v2 = 3;
}else{
  v2 = 4;
}
```

Standard partial evaluation yields four post-states, because both paths must be taken at each if statement, due to the dynamic predicate:

$$\{\texttt{v1} \mapsto 1, \texttt{v2} \mapsto 3\}$$
$$\{\texttt{v1} \mapsto 1, \texttt{v2} \mapsto 4\}$$
$$\{\texttt{v1} \mapsto 2, \texttt{v2} \mapsto 3\}$$
$$\{\texttt{v1} \mapsto 3, \texttt{v2} \mapsto 4\}$$

However, the static state correlates with the dynamic state, and two of the post-states are impossible: $\{\texttt{v1} \mapsto 1, \texttt{v2} \mapsto 4\}$, and $\{\texttt{v1} \mapsto 2, \texttt{v2} \mapsto 3\}$. In particular, even though `dyn_cond1` and `dyn_cond2` are not statically known, their value in a given static state can be inferred from the path taken through the if statements. Thus, if the static state is augmented with logical propositions about the dynamic states, the two post-states for the first if statement are:

$$\{\texttt{v1} \mapsto 1\} \wedge \texttt{dyn\_cond1}$$
$$\{\texttt{v1} \mapsto 2\} \wedge \neg\texttt{dyn\_cond1}$$

Thus, a straightforward application of a theorem prover to the post-state facts and the predicate `dyn_cond1 && dyn_cond2` would allow the specializer to avoid taking the two spurious paths at the second if statement. This example suggests that for complex programs, the application of generalized partial computation may yield significant improvements in the size and speed of specialization results.

Historically, generalized partial computation has not found widespread

use, due to the computational cost of theorem-proving. However, the advent of powerful and comparatively fast SMT solvers such as Z3 [de Moura and Bjørner, 2008] suggests that techniques from generalized partial computation may now be more readily applicable. By augmenting the state representation with facts gleaned from dynamic predicates on the path to a given state, it may be possible to produce better specialization results for programs such as `od`.

8.3.3 Layer Collapsing Across the Kernel Boundary

One of the most interesting advances in the development of the Linux kernel over the past several years has been the advent of eBPF [Borkmann and Starovoitov, 2014]. While originally, the Berkeley Packet Filter was a kernel-memory-space virtual machine for implementing fast firewalls, it has been extended to become a general-purpose kernel-side VM that runs trusted and verified code in kernel space to implement, e.g., zero-copy I/O [Begunkov, 2021b,a; Begunkov and Wei, 2023]. The code produced by classical program specialization has structural properties that make it a candidate for use with eBPF. First, partial evaluation can "collapse layers" in programs, inlining procedures, and intermingling caller and callee code. Second, partial evaluation unrolls static loops; the eBPF verifier requires all code to be loop-free. Moreover, by virtue of simplifying code, partial evaluation may be able to produce code that is easier for the eBPF verifier to analyze.

For programs that already run in a trusted context, these factors suggest opportunities for significant performance improvement. For example, consider a case where a generating extension can operate on a representation of the kernel-side portion of an IO call, where the actual writes or reads are treated as dynamic. A generating extension can intermingle the caller and callee code, producing loop-free code, which can be compiled to eBPF bytecode. In effect, one can think of the opportunity as one to

create a tool that performs an automated version of the sort of program "re-architecting" suggested by research into exokernels [Engler et al., 1995]. This prospect poses several interesting questions and challenges: clearly, moving an entire program into kernel space is infeasible, even if it is a program of trusted provenance. Thus, some means of identifying clear boundaries at which one could expose a reasonable interface would be important. On the other hand, the specialization tasks may be simpler; instead of specializing a full program, one might instead specialize a subset of a library, starting at a relevant procedure call.

8.4 Concluding Notes

GenXGen demonstrates the advantages of "cross-pollination" in programming-language research. By incorporating features available at the OS and hardware level, and by eschewing the traditional insistence on an absolute guarantee of soundness in favor of a probabilistic one, GenXGen is able to perform non-trivial specialization tasks in a reasonable amount of time. Moreover, though this work is not by any means AI research, it has the "flavor" of a corollary of "The Bitter Lesson" [Sutton, 2019]. Classical program specialization is a "meta-technique" for program rewriting—essentially a search over a tacit graph induced by the program points and static states of a program—which has intermittently fallen in and out of vogue, generally due to the difficulty of the underlying program-analysis tasks, as well as the computational infeasibility of performing specialization at scale. As the resurgence of theorem proving and logic programming shows, when given improved computing resources, revisiting these out-of-fashion techniques can bear significant fruit. The successes seen in this dissertation suggest that, coupled with decades of advances in computational power, further efforts to improve the quality of BTA results and improving the general efficiency of state-space exploration

are tasks that may well yield even more interesting results.

Bibliography

2019. *Using the GNU Compiler Collection (GCC)*. https://gcc.gnu.org/onlinedocs/gcc-9.2.0/gcc/

2023. *Debugging With GDB* (10 ed.). https://sourceware.org/gdb/current/onlinedocs/gdb.html/Process-Record-and-Replay.html#Process-Record-and-Replay

Aatira A. Ahmad, Abdul R. Noor, Hashim Sharif, Usama Hameed, Shoaib Asif, Mubashir Anwar, Ashish Gehani, Junaid H. Siddiqui, and Fareed Zaffar. 2021. TRIMMER: An Automated System for Configuration-Based Software Debloating. *IEEE Trans. on Softw. Eng.* (2021). Early access.

Mohannad Alhanahnah, Rithik Jain, Vaibhav Rastogi, Somesh Jha, and Thomas W. Reps. 2022. Lightweight, Multi-Stage, Compiler-Assisted Application Specialization. In *7th IEEE European Symposium on Security and Privacy, EuroS&P 2022, Genoa, Italy, June 6-10, 2022*. 251–269. https://doi.org/10.1109/EuroSP53844.2022.00024

L. O. Andersen. 1994a. *Program Analysis and Specialization for the C Programming Language*. Ph.D. Dissertation. DIKU, Univ. of Copenhagen.

Lars Ole Andersen. 1994b. *Program analysis and specialization for the C programming language*. Ph.D. Dissertation. University of Cophenhagen.

Paul Anderson, Thomas Reps, and Tim Teitelbaum. 2003. Design and Implementation of a Fine-Grained Software Inspection Tool. *Trans. on Softw. Eng.* 29, 8 (2003), 721–733.

Min Aung, Susan Horwitz, Rich Joiner, and Thomas Reps. 2014. Specialization slicing. *ACM Transactions on Programming Languages and Systems* (*TOPLAS*) 36, 2 (2014), 1–67.

John Aycock. 2003. A Brief History of Just-in-time. *ACM Comput. Surv.* 35, 2 (June 2003), 97–113. https://doi.org/10.1145/857076.857077

G. Balakrishnan. 2007. *WYSINWYX: What You See Is Not What You eXecute*. Ph.D. Dissertation. Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI. Tech. Rep. 1603.

G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. 2005a. CodeSurfer/x86 – A Platform for Analyzing x86 Executables, (Tool Demonstration Paper). In *Comp. Construct.*

Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005b. CodeSurfer/X86—A Platform for Analyzing X86 Executables. In *Proceedings of the 14th International Conference on Compiler Construction* (Edinburgh, UK) (*CC'05*). Springer-Verlag, Berlin, Heidelberg, 250–254. https://doi.org/10.1007/978-3-540-31985-6_19

G. Balakrishnan and T. Reps. 2010. WYSINWYX: What You See Is Not What You eXecute. *Trans. on Prog. Lang. and Syst.* 32, 6 (2010).

Robert M. Balzer. 1969. EXDAMS: extendable debugging and monitoring system. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1969 Spring Joint Computer Conference, Boston, MA, USA, May 14-16, 1969* (*AFIPS Conference Proceedings*), Harrison W. Fuller (Ed.), Vol. 34. AFIPS Press, 567–580. https://doi.org/10.1145/1476793.1476881

Earl T Barr and Mark Marron. 2014. Tardis: Affordable time-travel debugging in managed runtimes. *ACM SIGPLAN Notices* 49, 10 (2014), 67–82.

Lennart Beckman, Anders Haraldson, Östen Oskarsson, and Erik Sandewall. 1976. A partial evaluator, and its use as a programming tool. *Artificial Intelligence* 7, 4 (1976), 319–357.

Pavel Begunkov. 2021a. add BPF-driven requests. `https://lore.kernel.org/io-uring/cover.1613563964.git.asml.silence@gmail.com/` Request for Comment on proof-of-concept Linux kernel patch.

Pavel Begunkov. 2021b. io_uring zero-copy-send. `https://lore.kernel.org/io-uring/cover.1638282789.git.asml.silence@gmail.com/` Request for Comment on proof-of-concept Linux kernel patch.

Pavel Begunkov and David Wei. 2023. Fast ZC Rx Data Plane using io uring. (2023).

D. Binkley. 1993. Precise Executable Interprocedural Slices. *Let. on Prog. Lang. and Syst.* 2 (1993), 31–45.

Ras Bodík, Rajiv Gupta, and Mary Lou Soffa. 1997. Interprocedural Conditional Branch Elimination. In *Prog. Lang. Design and Impl.* 146–158.

Hans-Juergen Boehm. 1993. Space efficient conservative garbage collection. *ACM SIGPLAN Notices* 28, 6 (1993), 197–206.

Daniel Borkmann and Alexi Starovoitov. 2014. BPF Updates. `https://lore.kernel.org/netdev/1396029506-16776-1-git-send-email-dborkman@redhat.com/` Linux kernel patch adding the eBPF virtual machine.

Andrei Z. Broder. 1993. Some applications of Rabin's fingerprinting method. In *Sequences II*. Springer, 143–152.

Richard Bubel, Reiner Hähnle, and Ran Ji. 2009. Interleaving Symbolic Execution and Partial Evaluation. In *Formal Methods for Components and Objects*.

Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. 2006. EXE: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the ACM Conference on Computer and Communications Security*. Citeseer.

Cyril Cassagnes, Lucian Trestioreanu, Clement Joly, and Radu State. 2020. The Rise of eBPF for Non-intrusive Performance Monitoring. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*. IEEE, 1–7.

Sandrine Chirokoff, Charles Consel, and Renaud Marlet. 1999. Combining Program and Data Specialization. *Higher-Order and Symbolic Computation* 12, 4 (1999), 309–335.

David Chisnall. 2018. C Is Not a Low-Level Language: Your Computer is Not a Fast PDP-11. *Queue* 16, 2 (apr 2018), 18–30. `https://doi.org/10.1145/3212477.3212479`

Jong-Deok Choi and Janice M. Stone. 1991. Balancing Runtime and Replay Costs in a Trace-and-Replay System. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, California, USA, May 20-21, 1991*, Barton P. Miller and Charles E. McDowell (Eds.). ACM, 26–35. `https://doi.org/10.1145/122759.122761`

codesurfer 2018. CodeSurfer System. https://web.archive.org/web/20180829033330/https://www.grammatech.com/products/c

Alberto Coen-Porisini, Flavio De Paoli, Carlo Ghezzi, and Dino Mandrioli. 1991. Software Specialization Via Symbolic Execution. *Trans. on Softw. Eng.* 17, 9 (1991), 884–899.

Andrew Collette. 2013. LZFX Data Compression Library.

Charles Consel. 1990. Binding Time Analysis for High Order Untyped Functional Languages. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) (*LFP '90*). Association for Computing Machinery, New York, NY, USA, 264–272. `https://doi.org/10.1145/91556.91668`

Charles Consel and Olivier Danvy. 1989. Partial evaluation of pattern matching in strings. *Inform. Process. Lett.* 30, 2 (1989), 79–86.

Charles Consel, Julia L. Lawall, and Anne-Françoise Le Meur. 2004. A Tour of Tempo: A Program Specializer for the C Language. *Sci. of Comp. Prog.* 52 (2004), 341–370.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

Dawson R Engler, Wilson C Hsieh, and M Frans Kaashoek. 1996. C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 131–144.

D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) (*SOSP '95*). Association for Computing Machinery, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

Andrey P. Ershov. 1977. On the Partial Computation Principle. *Inf. Proc. Let.* 6, 2 (April 1977), 38–41.

Yoshihiko Futamura. 1971. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 2, 5 (1971), 45–50. (Updated and revised version published as [Futamura, 1999].).

Yoshihiko Futamura. 1988. Program Evaluation and Generalized Partial Computation. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems* (*FGCS*). OHMSHA Ltd. Tokyo and Springer-Verlag, 685–692.

Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391.

Yoshihiko Futamura, Kenroku Nogi, and Akihiko Takano. 1991. Essence of Generalized Partial Computation. *Theor. Comp. Sci.* 90, 1 (1991), 61–79.

Jeremy Gibbons and Nicolas Wu. 2014. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. 339–347.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Prog. Lang. Design and Impl.*

Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *Network and Dist. Syst. Security*.

E. Goto. 1974. *Monocopy and Associative Algorithms in an Extended LISP*. Tech. Rep. TR 74-03. Information Science Laboratory, Univ. of Tokyo, Tokyo, Japan.

Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J.Eggers. 2000. DyC: An Expressive Annotation-Directed Dynamic Compiler for C. *Theor. Comp. Sci.* 248, 1–2 (2000), 147–199.

Allan Heydon, Timothy Mann, Roy Levin, and Yuan Yu. 2006. *Software Configuration Management Using Vesta*. Springer-Verlag.

S. Horwitz, J. Prins, and T. Reps. 1988. On the Adequacy of Program Dependence Graphs for Representing Programs. In *Princ. of Prog. Lang.* 146–157.

Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural Slicing Using Dependence Graphs. *Trans. on Prog. Lang. and Syst.* 12, 1 (Jan. 1990), 26–60.

William H. Hudson, Vamshidhar R. Kommineni, Yi Meng, Kenneth Kai-Baun Ma, and Gerald F. Maffeo. 2008. U.S. Patent Number 7.469,362 B2, Using a Call Stack Hash to Record the State of a Process.

Neil Jones, Carsten Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc.

Neil D Jones. 1988. Automatic program specialization: A re-examination from basic principles. In *Partial evaluation and mixed computation*. 225–282.

Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language* (2nd ed.). Prentice Hall Professional Technical Reference.

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

Andrei V Klimov. 2009. A Java Supercompiler and its Application to Verification of Cache-Coherence Protocols. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 185–192.

Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6, 2 (1977), 323–350. https://doi.org/10.1137/0206024 arXiv:https://doi.org/10.1137/0206024

P. Lee and M. Leone. 1996. Optimizing ML with Run-Time Code Generation. In *Prog. Lang. Design and Impl.*

Mark Leone and Peter Lee. 1996. A declarative approach to run-time code generation. In *Workshop on Compiler Support for System Software (WCSSS)*, Vol. 73. Citeseer.

Peter Leven, Tilman Mehler, and Stefan Edelkamp. 2004. Directed Error Detection in C++ with the Assembly-Level Model Checker StEAM. In *Spin Workshop*.

J. Lim. 2011. *Transformer Specification Language: A system for generating analyzers and its applications*. Ph.D. Dissertation. Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI. Tech. Rep. 1689.

V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Sec. Symp.*

Henning Makholm. 1999. Specializing c-an introduction to the principles behind c-mix. (1999). This work discusses C-MixII, a full rewrite of C-Mix.

Steve Marx. 2018. Understanding Ethereum Smart Contract Storage. https://programtheblockchain.com/posts/2018/03/09/understanding-ethereum-smart-contract-storage/. Accessed: 2022-04-13.

Tilman Mehler and Stefan Edelkamp. 2006. Dynamic incremental hashing in program model checking. *Electronic Notes in Theoretical Computer Science* 149, 2 (2006), 51–69.

Andreas Jaeger Mark Mitchell Michael Matz, Jan Hubička. 2012. *System V Application Binary Interface, AMD64 Architecture Processor Supplement*. https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf

Madanlal Musuvathi and David L. Dill. 2005. An Incremental Heap Canonicalization Algorithm. In *Spin Workshop*.

E.W. Myers. 1984. Efficient Applicative Data Types. In *Princ. of Prog. Lang.*

Viet Yen Nguyen and Theo C. Ruys. 2008. Incremental hashing for Spin. In *International SPIN Workshop on Model Checking of Software*. Springer, 232–249.

Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2016. Lightweight User-Space Record And Replay. *arXiv preprint arXiv:1610.02144* (2016).

Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 377–389.

Chris Okasaki. 1999. *Purely functional data structures*. Cambridge University Press.

Gabrielle Paolini. 2010. *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*. Technical Report. Intel Corporation.

Michael O Rabin. 1981. Fingerprinting by random polynomials. *Technical report* (1981).

T. Reps. 1984. *Generating Language-Based Environments*. The M.I.T. Press.

T. Reps. 1998. Program Analysis via Graph Reachability. *Inf. and Softw. Tech.* 40, 11–12 (1998), 701–726.

T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. 1994. Speeding Up Slicing. In *Found. of Softw. Eng.* 11–20.

T. Reps, T. Teitelbaum, and A. Demers. 1983. Incremental Context-Dependent Analysis for Language-Based Editors. *TOPLAS* 5, 3 (July 1983), 449–477.

Tiark Rompf, Arvind K Sujeeth, Kevin J Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. 2014. Surgical precision JIT compilers. In *Acm Sigplan Notices*, Vol. 49. ACM, 41–52.

J. H. Saltzer, D. P. Reed, and D. D. Clark. 1984. End-to-End Arguments in System Design. *ACM Trans. Comput. Syst.* 2, 4 (nov 1984), 277–288. `https://doi.org/10.1145/357401.357402`

Peter Sestoft and Harald S: Gfndergaard. 1988. A bibliography on partial evaluation. *ACM Sigplan Notices* 23, 2 (1988), 19–26.

Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018a. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 329–339.

Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018b. TRIMMER: Application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 329–339.

Christopher S.F. Smowton. 2014. *I/O Optimisation and Elimination via Partial Evaluation*. Ph.D. Dissertation. Computer Laboratory, Univ. of Cambridge, Cambridge, UK. Tech. Rep. UCAM-CL-TR-865.

Christopher S.F. Smowton. 2020. Personal communication.

Venkatesh Srinivasan and Thomas Reps. 2015. Partial Evaluation of Machine Code. *SIGPLAN Not.* 50, 10 (Oct. 2015), 860–879. https://doi.org/10.1145/2858965.2814321

Minyoung Sung, Soyoung Kim, Sangsoo Park, Naehyuck Chang, and Heonshik Shin. 2002. Comparative Performance Evaluation of Java Threads for Embedded Applications: Linux Thread vs. Green Thread. *Inf. Process. Lett.* 84, 4 (nov 2002), 221–225. https://doi.org/10.1016/S0020-0190(02)00286-7

Richard Sutton. 2019. The bitter lesson. *Incomplete Ideas* (*blog*) 13, 1 (2019).

Valentin F Turchin. 1986. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems* (*TOPLAS*) 8, 3 (1986), 292–325.

M. Weiser. 1981. Program Slicing. In *Int. Conf. on Softw. Eng.* IEEE Comp. Soc., Wash., DC, 439–449.

M. Weiser. 1984. Program Slicing. *Trans. on Softw. Eng.* SE-10, 4 (July 1984), 352–357.

Min Xu, Rastislav Bodik, and Mark D Hill. 2003. A" flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*. 122–135.

Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* 23, 3 (1977), 337–343.

Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory* 24, 5 (1978), 530–536.