Lazy Evaluation for the Lazy: Automatically Transforming Call-by-value Into Call-by-need

Breno Campos Ferreira Guimarães
brenosfg@dcc.ufmg.br
UFMG
Belo Horizonte, Minas Gerais, Brazil

Fernando Magno Quintão Pereira
fernando@dcc.ufmg.br
UFMG
Belo Horizonte, Minas Gerais, Brazil

Abstract

This paper introduces *lazification*, a code transformation technique that replaces strict with lazy evaluation of function parameters whenever such modification is deemed profitable. The transformation is designed for an imperative, low-level program representation. It involves a static analysis to identify function calls that are candidates for lazification, plus the generation of closures to be lazily activated. Code extraction uses an adaptation of the classic program slicing technique adjusted for the static single assignment representation. If lazification is guided by profiling information, then it can deliver speedups even on traditional benchmarks that are heavily optimized. We have implemented lazification on LLVM 14.0, and have applied it on the C/C++ programs from the LLVM test-suite and from SPEC CPU2017. We could observe statistically significant speedups over `clang -O3` on some large programs, including a speedup of 11.1% on Prolang's `Bison` without profiling support and a speedup of 4.6% on SPEC CPU2017's `perlbench` (one of the largest programs in the SPEC collection) with profiling support.

CCS Concepts: • **Software and its engineering → Compilers**.

Keywords: lazy evaluation, code transformation

ACM Reference Format:
Breno Campos Ferreira Guimarães and Fernando Magno Quintão Pereira. 2023. Lazy Evaluation for the Lazy: Automatically Transforming Call-by-value Into Call-by-need. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC '23), February 25–26, 2023, Montreal, QC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3578360.3580270

ACM ISBN 979-8-4007-0088-0/23/02…$15.00
https://doi.org/10.1145/3578360.3580270

1 Introduction

A *strict* programming language evaluates actual parameters before passing them to functions. Many strict languages provide users with syntax to instruct the compiler to implement delayed evaluation for specific values. Examples include C#'s **Lazy<T>** class [20], Swift's **lazy** property [23], D's **lazy** storage class [2] and Idris's **Lazy** type qualifier [6, 7]. In all these cases, the onus of reasoning about the safety and profitability of lazy evaluation falls on the programmer.

This paper shows that compilers can lift such responsibility from users. To this end, it proposes *lazification of function arguments*, a compiler optimization that implements selective call-by-need. It identifies function calls whose actual parameters may benefit from being lazily evaluated, and transforms the program to carry out such strategy. The transformation analyzes the program's control flow, marking formal parameters[1] which are candidates for lazification. Then, call sites are analyzed to determine the safety and profitability of lazily evaluating actual instances of such parameters. For safe calls deemed profitable, we leverage Program Slicing [5, 40, 45, 46] to create a delegate function that encapsulates the value's computation. We then clone the callee function to create a version of it whose formal parameter is a *thunk* lazily evaluated at runtime. Our transformation is guided by profiling information, following the work of Chang and Felleisen [13]. The aforementioned approach is the materialization of the following contributions:

Lazification: the description of an algorithm that can be applied onto a low-level, imperative, intermediate representation of a compiler to replace call-by-value with call-by-need whenever deemed profitable.

Value Slicing: an adaptation of the classic program slicing algorithm to run on programs in the static single assignment (SSA) format. This adaptation extracts, from a program, a closure that computes an SSA value alive at a particular point of that program.

Summary of Results. We implemented lazification in LLVM [28], and evaluated it on 772 C/C++ programs, including SPEC CPU2017. Without profiling support, lazification

[1]Following typical jargon, we define formal parameters as the name of parameters as declared in the definition of functions; and we define actual parameters as the expressions passed to functions at their invocation sites.

causes an average slowdown of almost 10% compared to `clang` -O3. With profiling support, we observe a speedup of 1.10% with a confidence interval of 99%. In two of the benchmarks of the LLVM test suite, Prolang's `Bison` and `city`, speedups were above 10%. In SPEC CPU2017's `perlbench`, the observed speedup was 4.6%. Without profiling support, lazification increases code by 18.9% on average; with it, code expands by 9.2%.

2 Overview

Section 2.1 motivates the use of lazy evaluation in general purpose programming languages, and Section 2.2 explains how programs can be transformed to support it.

2.1 The Evaluation Zoo

The *evaluation strategy* of a programming language determines when expressions passed as arguments to procedures are evaluated. Three particular strategies concern this paper:

Definition 2.1 (Evaluation strategies)**.** Given a caller function $f : S \mapsto T$ and a callee function $g : R \mapsto S$, the evaluation strategy determines when $g(r), r \in R$ is resolved once the invocation $f(g(r))$ happens in an environment E that binds variable names to values. This paper concerns the three strategies below. We assume that $g(r) = s, s \in S$.

- **Call-by-value**: the evaluation of $f(g(r))$ is performed by first evaluating $g(r)$ in E to yield s, then computing $f(s)$ in environment E. Call-by-value is also called *strict evaluation.*
- **Call-by-name**: the evaluation of $f(g(r))$ is performed by passing to f a *closure* $(\lambda x.g(x), E)$, containing a reference to g and the environment E. The evaluation of $g(r)$ is performed at every program point in f where the value of $g(r)$, e.g., s, is used. Call-by-name is a form of *non-strict evaluation.*
- **Call-by-need**: the evaluation of $f(g(r))$ proceeds by passing to f a *closure* $(\lambda x.g(x), E, M)$, where $M : R \mapsto S$ is a table that maps arguments $r' \in R$ to values $s' \in S$. The evaluation of $g(r)$ is performed at every program point in f where $g(r)$ is used. The first time $g(r)$ is evaluated, the value s of $g(r)$ is computed and stored in M. In subsequent evaluations of $g(r)$, the precomputed $M(r)$ is returned. Call-by-need is a form of *non-strict evaluation.*

Programming languages that employ call-by-value semantics are referred to as *strict.* Most programming languages are strict. Notable examples of non-strict languages are Haskell [22], R [10] and Miranda [42]. Several reasons explain why strict evaluation is more popular. One is the cost of implementing non-strict evaluation. The closures mentioned in Definition 2.1 are commonly known as *thunks.* The creation and initialization of *thunks* incur a cost in terms of both time and memory. In fact, early compilers for non-strict languages would apply strictness analysis to implement strict evaluation where possible [16, 30]. Haskell still does it. A second reason is the unintuitive nature of non-strict evaluation in the presence of side effects, as discussed by Stadler et al. [38] in the context of the R programming language. ALGOL60, for instance, was the first language to introduce call-by-name semantics [4]. However, call-by-name led to confusing behavior, as famously evidenced by Jensen's device [29]. Call-by-name would later be discarded in ALGOL68 in favor of call-by-value [43]. Finally, strict evaluation leads to more predictable program behavior. In this regard, the Idris tutorial contains an illustrative discussion, which we quote: "[Given a type `thing: Int`], *what is the representation of* `thing` *at run-time? Is it a bit pattern representing an integer, or is it a pointer to some code which will compute an integer*?"[2] As a consequence, several features that are common in strict languages lack equivalents in the non-strict world.

Strict evaluation is not inherently superior to its non-strict counterpart. In particular scenarios, the deferred nature of non-strict evaluation can provide significant performance benefits, by avoiding unnecessary computations. Some strict programming languages such as C#, D, Idris, Kotlin, Scala, Scheme, Swift and OCaml, provide programmers with means to implement selective non-strict evaluation.

2.2 Lazification by Examples

The optimization described in this paper (i) automatically identifies opportunities for lazy evaluation; and (ii) transforms the code to capitalize on such opportunities. To demonstrate its workings, we shall use Example 2.2, which shows a situation where the optimization proposed in this paper delivers a large benefit.

Example 2.2. The logical conjunction (&&) at Line 06 in Figure 1 implements *short-circuit* semantics: if it is possible to resolve the logical expression by evaluating only the first term (`key != 0`), then the second term is not evaluated. Nevertheless, the symbols used in the conjunction, namely `key` and `value`, are fully evaluated before function `callee` is invoked at Line 22 of Figure 1. This fact is unfortunate, because the computation of the second variable, **value**, involves a potentially heavy load of computation, comprising the code from Line 15 to Line 21 of Figure 1.

The computation of **value** in Figure 1, as discussed in Example 2.2, is a promising candidate for *lazification.* Figure 2 shows the code that results from this optimization, as proposed in this paper[3]. Our implementation of lazification is a form of *function outlining* [49]: part of the program code is extracted into a separate function–a closure—which can be invoked as needed. This *thunk* appears in Lines 01-08 of

[2]See https://docs.idris-lang.org/en/v0.9.18.1/faq/faq.html

[3]For the sake of presentation only, we shall illustrate the effects of lazification using high-level C code. However, the prototype that we shall evaluate in Section 4 has been implemented onto LLVM, and affects exclusively the intermediate representation of this compiler.

```
01  void callee(
02    int key,
03    int value,
04    int N
05  ) {
06    if (key != 0 && value < N) {
07      printf("User has access\n");
08    }
09  }
```

```
10 void caller(
11   char *s0,
12   int *keys,
13   int N
14 ) {
15   int key = atoi(s0);
16   int value = -1;
17   for (int i = 0; i < N; i++) {
18     if (keys[i] == key) {
19       value = i;
20     }
21   }
22   callee(key, value, N);
23 }
```

Figure 1. Program that benefits from the lazy evaluation of parameter `value` when function `callee` is invoked at L-15.

```
01 struct thunk {
02   int (*f)(struct thunk *);
03   char *s0;
04   int *keys;
05   int N;
06   int val;
07   bool memo;
08 };
09
10 void callee(
11   int key,
12   struct thunk *tk,
13   int N
14 ) {
15   if (key != 0 && tk->f(tk) < N) {
16     printf("User has access\n");
17   }
18 }
```

Thunk initialization

```
19 void caller(
20   char *s0,
21   int *keys,
22   int N
23 ) {
24   int key = atoi(s0);
25   struct thunk value_thunk;
26   value_thunk.f =
27     &slice_callee_value;
28   value_thunk.memo = false;
29   value_thunk.s0 = s0;
30   value_thunk.keys = keys;
31   value_thunk.N = N;
32   callee(key, &value_thunk, N);
33 }
```

Check if value has been memoized.

Memoize results of lazy evaluation.

```
34 int slice_callee_value(struct thunk *tk) {
35   if (tk->memo) { return tk->val; }
36   int key = atoi(tk->s0);
37   int value = -1;
38   for (int i = 0; i < tk->N; i++) {
39     if (tk->keys[i] == key) {
40       value = i;
41     }
42   }
43   tk->val = value;  tk->memo = true;
44   return value;
45 }
```

Figure 2. Lazified version of the program in Figure 1. The interventions discussed in this paper are implemented at the level of the LLVM intermediate representation; however, this figure shows their effects as C code for the sake of readability.

Figure 2. The *thunk* is a triple formed by a table that binds values to free variables (Lines 03–05); a single-value cache (Lines 06 and 07) and a pointer to a function (Line 02). This function implements the computation to be performed lazily. The implementation of this function appears in lines 34 to 45 of Figure 2. An invocation to this closure in Line 15 of Figure 2 replaces the use of formal parameter `value`, which was computed eagerly in the original program. Example 2.3 illustrates the potential benefit of lazification.

Example 2.3. If the test `key != 0` is often false in Line 06 of Figure 1, then lazy evaluation becomes attractive. In this case, the speedup that lazification achieves in Figure 1's program is proportional to `N`. For instance, on a single-core x86 machine clocked at 2.2GHz, using a table with one million entries (`N` = 10^6), and ten thousand input strings, the original program runs in 4.690s, whereas its lazy version in Figure 2 runs in 0.060s. This difference increases with `N`.

Our implementation of lazification is able to transform the program in Figure 1 into the program in Figure 2 in a completely automatic way: it requires no annotations nor any other intervention from users. Nevertheless, the profitability of lazification depends on the program's dynamic behavior. If a function argument is rarely used, then it pays off to pass this argument as call-by-need. Otherwise, lazification leads to performance degradation. Regressions happen not only due to the cost of invoking the closure, but also to the fact that function outlining decreases the compiler's ability to carry out context-sensitive optimizations. This observation has motivated the use of profiling information in the compilation of functional programming languages as a means to choose which arguments must be lazily evaluated [13]. We follow such approach, as we explain in Section 3.5.

3 The Implementation of Lazification

This section describes our implementation of the techniques proposed in this paper to replace eager with lazy evaluation of function arguments. Figure 3 provides a unified view of the different steps that constitute the optimization that this paper advocates. That figure will guide our presentation.

3.1 The Underlying Language

Our presentation, in this section, uses a low-level language, formed by programs in Static Single Assignment (SSA) form [17]. These programs are represented as *control-flow graphs* (CFG).

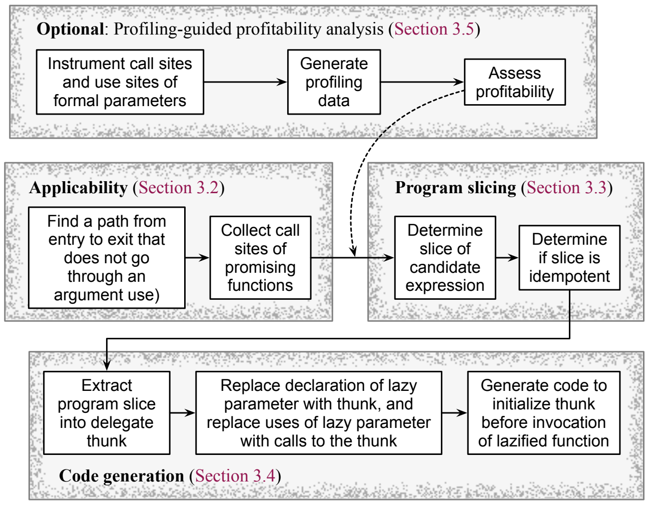
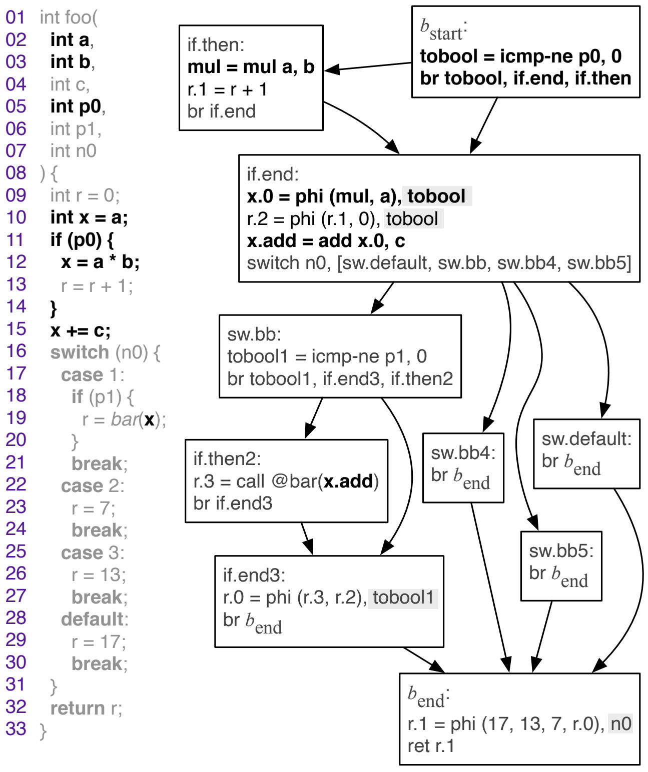

Optional: Profiling-guided profitability analysis (Section 3.5)
Instrument call sites and use sites of formal parameters
Generate profiling data
Assess profitability
Applicability (Section 3.2)
Find a path from entry to exit that does not go through an argument use)
Collect call sites of promising functions
Program slicing (Section 3.3)
Determine slice of candidate expression
Determine if slice is idempotent
Extract program slice into delegate thunk
Replace declaration of lazy parameter with thunk, and replace uses of lazy parameter with calls to the thunk
Generate code to initialize thunk before invocation of lazified function
Code generation (Section 3.4)

Figure 3. Overview of the different steps that constitute our implementation of lazification of function arguments.

Nodes in a CFG represent *basic blocks*: maximal sequences of instructions that execute in succession regardless of the program flow. CFG edges represent program flows determined by *terminators*. Terminators are instructions that end basic blocks: unconditional branches (with one successor); conditional branches (with two successors); and switches (with three or more successors). CFGs contain two special nodes: b_{start} and b_{end}. The former has zero predecessors; the latter, zero successors. If $G = (V, E)$ is a control-flow graph, then every node $b \in V, b \neq b_{start}$ can be reached from b_{start}. Similarly, every node $b \in V, b \neq b_{end}$ reaches b_{end}. Conditional branches are controlled by *boolean predicates*; switches are controlled by *integer predicates*. Given that we are assuming the SSA representation, programs might contain phi-functions at the beginning of basic blocks. An instruction such as $x = \text{phi}(x_1, x_2)$ at the beginning of a block b will copy either x_1 or x_2 into x, depending on the path through which b has been reached. Figure 4 shows a program written in C, and its low-level control-flow graph.



```
01 int foo(
02   int a,
03   int b,
04   int c,
05   int p0,
06   int p1,
07   int n0
08 ) {
09   int r = 0;
10   int x = a;
11   if (p0) {
12     x = a * b;
13     r = r + 1;
14   }
15   x += c;
16   switch (n0) {
17     case 1:
18       if (p1) {
19         r = bar(x);
20       }
21       break;
22     case 2:
23       r = 7;
24       break;
25     case 3:
26       r = 13;
27       break;
28     default:
29       r = 17;
30       break;
31   }
32   return r;
33 }
```

```
b_start:
tobool = icmp-ne p0, 0
br tobool, if.end, if.then

if.then:
mul = mul a, b
r.1 = r + 1
br if.end

if.end:
x.0 = phi (mul, a), tobool
r.2 = phi (r.1, 0), tobool
x.add = add x.0, c
switch n0, [sw.default, sw.bb, sw.bb4, sw.bb5]

sw.bb:
tobool1 = icmp-ne p1, 0
br tobool1, if.end3, if.then2

if.then2:
r.3 = call @bar(x.add)
br if.end3

if.end3:
r.0 = phi (r.3, r.2), tobool1
br b_end

sw.bb4:
br b_end

sw.bb5:
br b_end

sw.default:
br b_end

b_end:
r.1 = phi (17, 13, 7, r.0), n0
ret r.1
```

Figure 4. A program and its control-flow graph (CFG). Black-bold fonts highlight the backward slice that computes **x**, as used in Line 19. Gray boxes in the CFG show predicates used to gate phi-functions (to be explained in Section 3.3.1).

3.2 Identification of Applicability

In this paper, lazification is implemented *per call site*. In other words, given a caller function $g(\ldots)$, a callee function $f(\ldots, a, \ldots)$, and an invocation site of f, at some program point $p : f(\ldots, exp, \ldots), p \in g$, lazification will lead to four automatic interventions in the code:

1. An outlined delegate function $d_{exp}(thunk_{exp})$, corresponding to a backward slice of exp, will be synthesized and inserted into the target program.
2. A clone $f_p(\ldots, thunk_{exp}, \ldots)$ of f will be produced and incorporated into the set of function definitions of the target program.
3. The invocation of f at p will be replaced with an invocation to f_p, with $thunk_{exp}$ initialized with the free variables in exp.
4. Each use of exp in g is replaced by a call to d_{exp}.

Lazification is only profitable if some invocations of f might not use all its formal parameters. The problem of determining if such is the case statically would be undecidable, as a consequence of Rice [35]'s Theorem. Therefore, to determine the applicability of lazification, we adopt a conservative estimate of this property. This analysis identifies the so called *promising functions*, which are defined as follows:

Definition 3.1 (Promising Function)**.** Given a function f, plus a, one of its formal arguments, function f is *promising* in regard to a if its CFG contains a path from b_{start} to b_{end} that does not traverse any point where a is used.

Example 3.2. Function `callee` in Figure 1 is promising in regards to `value`. There exists a path from the function's entry point to the function's end point that does not go across the use of formal parameter `value`. Such path is created by the short-circuit semantics of the operator &&. Function `foo` in Figure 4 is promising regarding `b` and `p1`. Regarding the latter, the CFG of `foo` contains a use of `p1` in block `sw.bb`. Every path from b_{start} to b_{end} that dodges block `sw.bb` is a promising path in regard to parameter `p1`.

Whether a function is *promising* can be computed by a depth-first-search through its CFG, starting at b_{start}. Visit each successor b_i recursively, interrupting the search if b_i contains a use of parameter a. If the search eventually reaches b_{end}, then a viable path exists and the function is promising. Otherwise, if all eligible nodes have been visited and b_{end} is not reachable, then the function is discarded as an optimization candidate. Nevertheless, while Definition 3.1 aids in identifying which functions are candidates for lazification, it is innately conservative. There is no guarantee that the promising paths found by the DFS are exercised during program execution. Therefore, the DFS is effective in pinpointing occasions where lazification is *applicable*, but it is less reliable when it comes to evaluating whether such application is *profitable*. That is the goal of a profiler.

3.3 Slicing

To implement lazy evaluation of a function f's formal parameter, we replace this formal parameter, in the definition of f, with a *thunk*. The *thunk* contains a delegate function, computed as the backward slice of some actual parameter in an invocation of f. Definition 3.3 revisits the notion of backward slice, adapting to our context the formalization proposed by Weiser [45] in the early eighties.

Definition 3.3 (Backward Slice)**.** Given a program P, plus a variable v defined at a program point $p \in P$, the backward slice of v at p is a subset P_s of P's program points containing p, such that if P computes v with value n, given input I, then P' computes v with value n, given input I. We call the pair (v, p) a *Slice Criterion.*

Notice that an adaptation of Weiser's concept of program slice is in order. Weiser consider a *statement* as the slice criterion, whereas we consider a *variable*. We are interested in the contents of this variable, once it is used as the actual argument of a function call. Example 3.4 illustrates our notion of backward slice. Section 5 discusses further differences.

Example 3.4. Let the use of variable **x** at Line 19 of the C program in Figure 4 be a slice criterion. The backward slice that computes this slice criterion, e.g., (**x**, Line 19), appears in a darker font in the C program.

3.3.1 Gating Phi-Functions. The computation of a backward slice, given criterion (v, ℓ) involves finding the set of *program dependencies* that compute v at line ℓ. Following Ferrante et al. [18], we recognize two types of dependencies: *data* and *control*, concepts that Definition 3.5 revisits. Like Definition 3.3, Definition 3.5 adapts the early notion of dependency stated by Ferrante et al. [18]. However, whereas in that setting Ferrante et al. were interested in program statements, we care for dependencies between program symbols:

Definition 3.5 (Dependencies)**.** A variable u is *data* dependent on a variable v if u is defined by an instruction that uses v. A variable u is *control* dependent on a variable v if the assignment of u depends on a terminator (e.g., a conditional branch) controlled by v. A variable u *depends* on a variable v if it is either data or control dependent on v, or if it depends on a variable w that depends on v.

Definition 3.5 gives us a trivial algorithm to compute data dependencies: for each instruction "$u = \ldots, v, \ldots$", we make u data dependent on v. To compute control dependencies, we resort to an old approach adopted in the context of SSA-form programs: *gating* [41]. Gating consists in appending, as a special notation, predicates to certain phi-functions. As seen in Section 3.1, *predicate* is a variable that controls a branch or a switch. Definition 3.6 provides an algorithmic denotation of gates, which Example 3.7 illustrates. For a precise description of the gating transformation, we refer the reader to Section 5 in the work of Herklotz et al. [21].

Definition 3.6 (Gates)**.** Given a control-flow graph $G = (V, E)$, and two vertices $b_0 \in V$ and $b_1 \in G$, we say that b_1 *post-dominates* b_0 if every path from b_0 to b_{end} goes through b_1. We say that b_1 is the *immediate* post-dominator of b_0 if b_1 post-dominates b_0, and for any other node b_p that post-dominates b_0, either $b_p = b_1$, or b_p post-dominates b_1. Given a predicate p that controls a terminator at basic block b_0, we say that p *gates* every phi-function in the basic block b_1 that immediately post-dominates b_0.

Example 3.7. The boolean predicate `tobool` in the CFG of Figure 4 gates the two phi-functions at the beginning of basic block `if.end`. The integer predicate `n0` gates the phi-function at the beginning of basic block b_{end}. Gates appear in gray boxes in the CFG of Figure 4.

Gating transforms control dependencies into data dependencies, because the gates add to the right side of phi-functions—as a special notation—the predicates that control those assignments. Notice that Definition 3.6 leads to a simplified version of the so called *Gated Static Single Assignment* representation [31]. The only difference between gating, as implied by Definition 3.6 and gating as defined by Ottenstein et al. concerns what Ottenstein et al. calls η-functions: special copy instructions that split variables defined within a loop and used outside it. In our context, such form of live range splitting does not occur. For those familiar with Chen et al. [14]'s work, gating yields what they call *predicated SSA-form.*

3.3.2 Single Dynamic Assignment. Given a slice criterion (v, ℓ), this paper builds a backward slice that computes a single dynamic assignment of v. In other words, v is assigned only once during the execution of the delegate function that implements the lazy computation of v. To ensure single dynamic assignment, the slice cannot compute the value of v within a loop. This restriction is necessary because the value of v whose computation is postponed due to lazy evaluation must be unique: that is the value of v when the function that uses it as an actual parameter is invoked. Example 3.8 illustrates this observation.

Example 3.8. Figure 5 (a) shows a program, and the backward slice that refers to the criterion (**x**, Line 08). Figure 5 (b) shows the corresponding delegate function. Notice that this backward slice contains a loop; however, once its *thunk* is evaluated, this loop has been fully resolved. On the other hand, if the criterion lays inside the loop, then it will be constrained by the loop boundaries. Figure 5 (c) shows the same program with the criterion inside the loop (Line 07). Because a slice must produce a single dynamic assignment, the criterion is defined in a single iteration of the loop, although it can still depend on loop-invariant values, like **c**.

3.3.3 Purity. In this paper we consider only slices that are *pure*, that is, that do not cause side effects. To implement this restriction, in practice, we consider as invalid slices containing the following pieces of code: (i) instructions that store

(a)
```
int foo(int a, int b, int c) {
  int x = a;
  int sum = 0;
  while (x < b) {
    x += c*c;
    sum += x;
  }
  bar(x);
  return sum;
}
```

(b)
```
int slice(
  int a,
  int b,
  int c
) {
  int x = a;
  while (x < b) {
    x += c*c;
  }
  return x;
}
```

(c)
```
int foo(int a, int b, int c) {
  int x = a;
  int sum = 0;
  while (x < b) {
    x += c*c;
    sum += x;
    bar(x);
  }
  return sum;
}
```

(d)
```
int slice(
  int x,
  int c
) {
  x += c*c;
  return x;
}
```

Figure 5. (a-b) Example of program slice whose criterion lays outside a loop. (c-d) Same example, this time with slice criterion inside a loop.

into memory; (ii) functions without bodies[4]; (iii) instructions that may trigger exceptions.

Slices can still contain read-only memory accesses, i.e., load instructions[5]. Nevertheless, store operations are forbidden. This restriction, coupled with the interdiction of library functions, effectively prevent the existence of slices containing side effects[6]. We avoid side effects for two reasons. First, because lazification reorders program actions. Thus, it could modify the semantics of the program by altering the execution order of operations that change state. Second, because we implement memoization, as explained in Section 3.4.2. Therefore, in the presence of side effects, state would change only once, and not multiple times, as originally intended.

3.4 Code Generation

Once a slice of a program P is composed, we use it to generate a new program P_s that computes the same slice criterion. P_s contains a subset of the basic blocks in P. Program P_s might contain some of the control-flow edges present in P; however, it can also contain edges that did not exist in the original program. To explain how we build the control-flow graph of P_s, we define the notion of the n^{th} *dominator* of a basic block. This notion relies on the following standard definitions [3]: given a control-flow graph $G = (V, E)$, and two vertices $b_0 \in V$ and $b_1 \in V$, we say that b_0 *dominates* b_1 if every path from b_{start} to b_1 goes through b_0. We say that b_0 is the *immediate* dominator of b_1 if b_0 dominates b_1, and for any other node b_d that dominates b_1, either $b_d = b_0$, or b_d dominates b_0. Immediate dominance forms an antisymmetric relation whose shape is a directed tree: *the Dominance Tree.*

Definition 3.9 (First Dominator)**.** Given a set of nodes B, and a basic block $b \notin B$, the *first* dominator of b (within B) is $b_n \in B$ (thus $b_n \neq b$) such that b_n dominates b and b_n is the closest parent of b in the dominance tree.

3.4.1 Outlining.

The *first* dominator of a node is not necessarily its *immediate* dominator: given two nodes, b_1 and b_2 that dominate b, one of them will be the first dominator, even though none of them might be b's immediate dominator. We use Definition 3.9 to formalize the notion of a *sliced program.*

Definition 3.10 (Sliced Program)**.** Let a program P be represented by a CFG (V_p, E_p). And let $V_s \subseteq V_p$ be the set of basic blocks from P that belong into a backward slice created after some slice criterion. From V_s we derive a new program $P_s = (V_s, E_s \cup \{b_{start}, b_{end}\})$, where E_s is defined as follows:

1. If $b_0 \longrightarrow b_1 \in E_p$ for some $b_0 \in V_s$ and $b_1 \in V_s$, then $b_0 \longrightarrow b_1 \in E_s$.
2. If $b_0 \longrightarrow b_1 \in E_p$ for some $b_0 \notin V_s$ and $b_1 \in V_s$, and b_1 contains a use of a variable v that is not defined in b_1, then $b_f \longrightarrow b_1 \in E_s$, where b_f is the first dominator of b_1 in V_s.
3. If a block $b \in V_s$ contains a definition of every variable used in it, then E_s contains an edge $b_{start} \longrightarrow b$.
4. If b contains the slice criterion, then E_s contains the edge $b \longrightarrow b_{end}$.

The essential property of a SSA-form program is that every definition of a variable dominates all its uses. Any sliced program P_s satisfies this property, as long as the original program P is *strict*. Strictness means that variables cannot be used without being defined. Budimlic et al. [8] have demonstrated the SSA property for strict programs. We use Budimlic et al.'s observation to prove Theorem 3.11 (proof is available in the first author's master dissertation [19]).

Theorem 3.11. *Let $P_s = (V_s, E_s)$ be the sliced program derived from a strict SSA-form program $P = (V_p, E_p)$. For any basic block $b \in V_s$ that contains an instruction that uses a variable v, we have that either b contains a definition of v, or V_s contains a node that dominates b.*

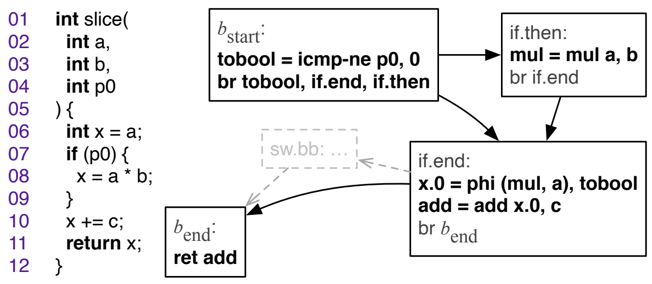
Example 3.12. Figure 6 shows the sliced program that corresponds to the slice criterion (**x**, Line 19). The dashed block does not contain instructions that pertain to the slice (it does

[4]An exception is made for some standard library functions and some LLVM intrinsic functions which are known to be free of side-effects

[5]Loading from pointers are allowed as long as these pointers are tagged as "*read-only*". We use a conservative analysis to determine read-only pointers: a pointer p is read-only if no alias of p (as determined by the combination of LLVM's `basic-aa`, `tbaa` and `globals-aa`) is the base pointer of a store operation anywhere in the program module, except immediately after the pointer is allocated. This analysis is performed for every function in a module, but it is flow insensitive.

[6]There is another, more subtle cause of side effects: infinite loops. Since our slices may contain loops, they could alter the semantics of the program by changing when an infinite loop executes. However, since we are concerned with optimizing C/C++ specifically, these languages allow loops free from side-effects with non-constant conditions to be assumed to terminate (See C11 6.8.5/6 and C++17 6.8.2.2/1)

not belong into V_s). Therefore, that block is bypassed by Rule (2) in Definition 3.10.



```
int slice(
  int a,
  int b,
  int p0
){
  int x = a;
  if (p0) {
    x = a * b;
  }
  x += c;
  return x;
}
```

Figure 6. Sliced program produced when Definition 3.10 is applied onto the slice criterion (**x**, Line 19) in Figure 4. The dashed block does not belong into the slice.

3.4.2 Memoization. The procedure described in Section 3.4.1 allows us to encapsulate the computation of a given value into a delegate function. This function, alongside free variables on which it depends, are enough to synthesize *thunks* to effectively implement call-by-name semantics. However, the implementation of call-by-need requires *memoization*. Memoization guarantees that the delegate function is executed at most once, even when the *thunk* itself is evaluated multiple times. To generate code to carry out memoization, we augment *thunks* and the generated delegate function with a cache. For *thunks*, we add two fields:

- A memoized value, whose type matches the delegate function’s return type.
- A boolean memoization flag.

Example 3.13. Fields `val` and `memo` in lines 6 and 7 of Figure 2 store memoization data.

The delegate function is also modified to take advantage of memoization. We achieve this by adding two special nodes to its CFG: b_{memo_entry} and b_{memo_return}. The first replaces b_{start} as the function’s new entry point. It contains two instructions: a check to determine if the memoization flag is set, and a conditional test predicated by this check. In case the check is false, b_{memo_entry} branches to b_{start}, the function’s original entry point. Otherwise, it branches to b_{memo_return}, which itself contains a single instruction: a return statement, whose single operand is the memoized value. Finally, in b_{end}, the function’s original exit block, we add two instructions immediately preceding its return statement. One to memoize the value computed by the function in the *thunk*, and another to set the *thunk*’s memoization flag.

Example 3.14. Line 35 of Figure 2 encodes the conditional block created to check if the *thunk* has already been called. Line 43 in Figure 2 sets the memoization flag, once the *thunk* is executed for the first time.

3.4.3 Callee cloning. The last step required to *lazify* a call site is to transform the caller and callee functions. We start by modifying the call site where the callee is invoked within the caller. The original invocation $f(\ldots, a, \ldots)$ is replaced with a call to a clone of f, with signature $f_a(\ldots, thunk_a, \ldots)$. Function f_a is equivalent to f, except that every use of a is replaced by an invocation of $thunk_a$. In the LLVM intermediate representation, the invocation of the *thunk* involves two instructions: the first loads the address of the delegate function from $thunk_a$; the second calls that function itself.

Example 3.15. Figure 2, Lines 10-18, show the lazified version of `callee`. The original `value` formal parameter was replaced by a *thunk* `thk`. The only use of `value` was replaced by the invocation `thk->fptr(thk)` in line 15 of the new function.

Changes in the Caller. The caller function is also transformed to accommodate lazification. Given the actual parameter a being lazified, we replace the original definition of a by the declaration and initialization of the *thunk*. The original definition of a is replaced by the declaration of a variable `thk` of type $thunk_a$. We add stores to initialize `thk`’s contents: the function pointer is initialized to the delegate function’s address; the value of the free variables in scope are copied into their counterparts in the *thunk*; the memoization flag is initialized to false. The call $f(\ldots, a, \ldots)$ is replaced by a call $f_a(\ldots, \texttt{thk}, \ldots)$. Example 3.16 illustrates these interventions.

Example 3.16. Figure 2 shows the *thunk* initialization code in Lines 25-31. This initialization stores in `value_thunk` all the data necessary to carry out the computation that, originally, would happen in lines 16-21 of Figure 1.

Further uses of a in the caller (other than in the modified call site) are also replaced by a call to `thk`’s delegate function pointer. Replacing the uses of a in the caller function is not necessary for correctness: the existing computation of a and its other uses could be kept as-is. However, this substitution avoids redundancy. If the original computations were kept, then the gains of lazification would be lost. Thus, reusing the *thunk* throughout the caller guarantees that the code encapsulated in the delegate function runs at most once.

3.5 Identification of Profitability

We follow Chang and Felleisen [13]’s approach to identify profitable scenarios where lazification should be applied. More details about our implementation can be found in Appendix A.1. The key differences between our approach and Chang and Felleisen’s are in terms of engineering: we are instrumenting the low-level, imperative, LLVM intermediate representation. Chang and Felleisen was instrumenting a high-level, side-effect free, representation of the Racket abstract syntax tree.

4 Evaluation

This section discusses three research questions, each related to the impact of lazification on one of the following aspects of optimized programs:

RQ1: Running time of programs.
RQ2: Compilation time of programs.
RQ3: Code size of programs.

Experimental Setup: Experiments were executed on a dedicated server featuring an Intel Xeon E5-2620 CPU at 2.00GHz, with 16GB RAM, running Linux Ubuntu 18.04, with kernel version 4.15.0-123. The baseline compiler used to test our optimization was `clang` 14. Programs run sequentially.
Benchmarks: The evaluation presented in this section uses benchmarks from the LLVM test-suite plus SPEC CPU2017. The LLVM test-suite contains 754 executable programs, including microbenchmarks. SPEC CPU2017 contains 43 programs, but only 27 are written in C or C++, and only 18 of them provide Makefiles to be plugged into the LLVM test suite. Experiments run in the LLVM test-suite, for they require two inputs, one for profiling and another for validation. We use the “train" input for profiling, and the “reference" for validation. Most of the benchmarks in the LLVM test-suite are small code snippets that offer no opportunity for lazification. In total, 111 of our benchmarks contain promising functions (see Definition 3.1).
Measurement Methodology: Running times reported in this section are the average of 10 executions. We adopt a confidence level of 99%; thus, we consider as statistically significant only differences in running time with a p-value under 0.05, as derived via Student’s t-test.

4.1 RQ1 - Running Time

We report results for the 111 benchmarks containing promising functions, including 18 programs from SPEC CPU2017. We evaluate three setups for lazification. In this case, the *usage ratio* of a parameter is the ratio between the number of times the function is called, and the number of calls where the parameter is used:

Naïve: we lazify *every* call site where a promising function is invoked, without any dynamic execution data.
PGO-0.4: we lazify only call sites whose parameter usage ratio is below 40%, as determined via profiling.
PGO-0.1: same setup as PGO-0.4, but lazifying only call sites with a parameter usage ratio below 10%.

The last two setups, which involve profiling, use the “train" input of each benchmark to collect data, and the “reference" input to run the optimized code. Figures 7, 8 and 9 report running-time variations measured as $\frac{runtime_{orig}}{runtime_{opt}}$. Thus, values above 1.0 represent speedups and values below 1.0 are slowdowns. $runtime_{orig}$ is the running time of binaries built with `clang` 14, at the maximum optimization level (-O3). The figure highlights programs from SPEC CPU2017, plus the programs with largest speedups and regressions.

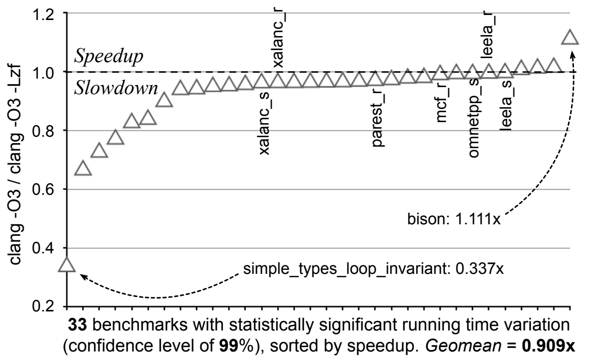


Figure 7. Running-time variation due to the naïve lazification, i.e., without the support of profiling data.

The Naïve Setup. Figure 7 shows running-time variations for programs optimized using the naïve setup. In total, 33 out of 111 benchmarks showed a statistically significant difference to the baseline. Figure 7 only shows results for these. The largest observed speedup was 1.11x for `bison`. No speedup could be measured in programs from SPEC CPU2017. Largest slowdowns occurred for small benchmarks, seen in the tail end of Figure 7. A maximum regression of 0.337x was observed in `simple_types_loop_invariant`, whose baseline runs for 1.5 seconds. Overall, naïve lazification incurs a geomean speedup of 0.909x, meaning that it slows programs down. This result is expected, due to the lack of profiling information.

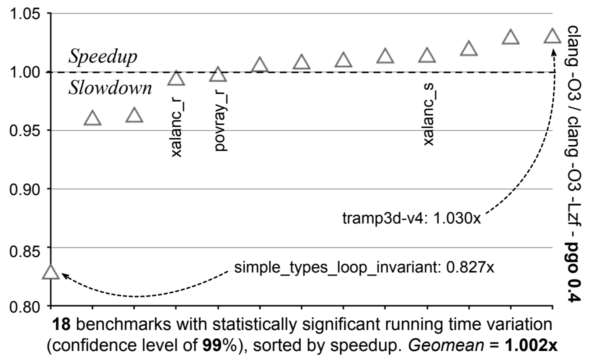


Figure 8. Running-time variation due to profiling-guided lazification, with a usage ratio of 0.4.

The PGO 0.4 Setup. Figure 8 shows the result of applying lazification with the PGO-0.4 setup. In total, 54 programs were optimized—less than half of the 111 optimized by the naïve setup. Thus, profile data increases the optimization’s parsimony significantly. From these 54 programs, only 18 showed mean runtimes which differed statistically from the baseline. Eight programs show speedups, while five had slowdowns. SPEC CPU2017's `Xalan-C`, which performed worse than baseline in the previous experiment, now improves over baseline with a speedup of 1.012x. Similarly, the `simple_types_loop_invariant` benchmark, while still showing the largest slowdown, improved from 0.337x to 0.828x. Overall, when parameterized with a usage ratio threshold of 40%, lazification incurs a geomean speedup of 1.002x.

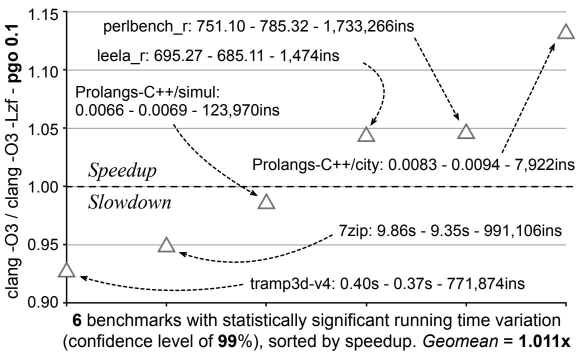


Figure 9. Running-time variation due to profiling-guided lazification, with a usage ratio of 0.1. For each program, we show running time (average of 20 executions) with `clang -O3`, with `clang -O3` plus lazification, and the number of instructions of the program, when compiled with `clang -O3`.

The PGO 0.1 Setup. Figure 9 shows running-time variations when optimizing with the PGO-0.1 setup. For this experiment, only six programs showed a statistically significant difference from the baseline. Programs were evenly split with half being under baseline and half above baseline. Overall, lazification with a 10% usage ratio provides a geomean speedup of 1.011x. Although small on average, when restricted to individual benchmarks, more noticeable boosts in performance can be observed. Amongst the programs which show speedups, there is a notable improvement of 1.045x for SPECrate's `perlbench`, the longest running benchmark in this suite. We emphasize that such speedup was obtained against `clang` at its highest optimization level. At this level, `clang` applies 278 passes onto the target program.

4.2 RQ2 - Compilation Time

This section evaluates the impact of lazification on the compilation time. Numbers are relative to `clang -O3`. We consider lazification in its most aggressive flavor: the **Naïve** setup introduced in Section 4.1. This configuration gives us a worst case scenario. We omit the other two settings evaluated in Section 4.1, because we could not observe statistically significant variations in compilation time to the naïve setup.

Discussion: Figure 10 shows absolute compilation times, for two versions of `clang -O3`: without and with lazification. Considering only the 111 benchmarks where lazification

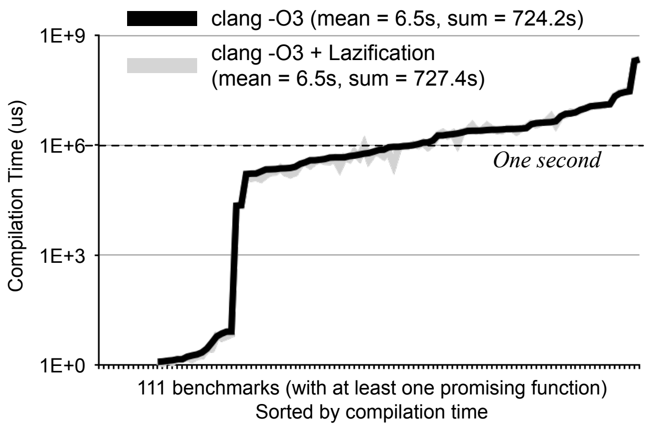


Figure 10. Variation in the compilation time in the 111 benchmarks that contained at least one promising function (a function that could be lazified).

could be applied, our optimization has increased compilation time by a small factor: $727.4s/724.2s = 0.44\%$. There were pathological cases, although not many. The compilation time of SPEC CPU2017's `leela_r`, for instance, went from 4.2s to 6.1s. Lazification still runs on the applicability analysis of Section 3.2, even in programs where it cannot be applied. Nevertheless, this static analysis takes negligible time compared to the rest of the `clang -O3` passes. In total, lazification increases the compilation time of all the 754 programs in our test suite from 776s to 780s.

4.3 RQ3 - Code Expansion

This section measures the impact of lazification on the final size of binary programs. Our baseline in this evaluation is also `clang` -O3, and we use the same three setups presented in Section 4.1. We measured the size, in bytes, of the `.text` section of the optimized binaries used in Section 4.1, and compared them against their baseline counterparts. We compute code bloat as $\frac{size_{opt}}{size_{orig}}$, where $size_{orig}$ is the size of the `.text` section of the binary generated by `clang` 14 at its highest optimization level (-O3). Figure 11 shows absolute numbers and ratios observed in this experiment.

Discussion. Our implementation of lazification leads to code expansion in the three different setups that we consider. The most aggressive setup, **naïve**, naturally experiences the largest increase: we jump from 28.19 to 30.85 million LLVM instructions (mean growth of 9.4%, median growth of 10.5%). If profiling data is used to restrict the scope of the optimization, the code expansion decreases, albeit remaining noticeable. The median expansion in the **pgo-4.0** setup is 4.1%, whereas in the **pgo-1.0** setup it is of 2.3%.

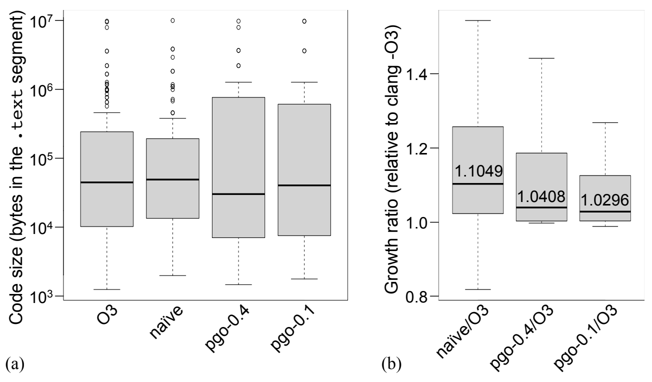


Figure 11. Binary-size overhead of lazification, considering the three setups discussed in Section 4.1. Part (a) shows absolute sizes, and part (b) shows ratios relative to -O3; Numbers in boxes represent medians.

5 Related Work

We are not aware of a code transformation technique that replaces the computation of actual parameters with closures that will be evaluated lazily in the context of a low-level, imperative, program representation. Nevertheless, the compiler-related literature abounds with examples of code motion that extrapolate the boundaries of functions. The extended version of this paper provides a more comprehensive overview of this literature (see Appendix A.2). In this section, we briefly cover the most important related techniques.

First, notice that we perform a form of *inter-procedural redundancy elimination* (IPRE), a concept that Agrawal et al. [1] has introduced. However, whereas Agrawal et al. extracts individual instructions from within the body of functions, we extract whole slices of programs, which we pack in a closure. Second, there exists work that replaces eager implementations of library functions with lazy versions [47]. Our work, in contrast, assumes no semantic knowledge of functions: our optimization is implemented at the level of assembly instructions, not on an API. Third, there exists much research on the implementation of *strictness analysis* [9, 16, 27, 44], whose goal is to replace lazy with eager evaluation of arguments. This paper proposes exactly the opposite. Finally, *program slicing*, although a well-known technique [45], is typically defined over statements, not values, as we do in Section 3.3. This difference persists in recent implementations of backward slicing in LLVM [48], and that consequently could not be used to produce closures. We now go over two lines of work that directly relate to our optimization.

Function Outlining. Zhao and Amaral [49] have proposed function outlining as a way to reduce the code of programs. The slices that we extract from programs resemble Zhao and Amaral's implementation. However, we do outlining on non-structured SSA-form programs, whereas Zhao and Amaral do it on the program's AST. Although Zhao and Amaral's language supports goto, outlining is restricted to single-entry-single-exit (SESE) regions. Furthermore, we do outlining by value, whereas Zhao and Amaral does it by region. That means that we outline the slice of the program that computes a single SSA-form value v, whereas Zhao computes a given SESE code region R. Finally, the scope of our outlining is the dynamic assignment of the value v (either the surrounding function itself, or the outermost loop that contains v). The scope of Zhao and Amaral's outlining is the maximal region R that contains the cold code.

Profiling for Laziness. We adopt Chang and Felleisen [13] profile to discover promising calling sites. Chang and Felleisen also implement a form of "lazification"; however, their implementation follows up the implementation [11, 12] of lazy evaluation in typical functional programming languages. Thus, they work on a declarative setting, whereas we work on an imperative setting. Laziness in Chang and Felleisen's case is computed by a downward traversal of the program's AST (these are the rules in Section 3.2 of Chang and Felleisen's paper). We, in turn, compute lazy expressions via a backward traversal of the program's CFG in SSA form. Program slicing and closure extraction is not a concern in Chang and Felleisen's context, because they define lazification within a language that provides two constructs: *force* and *delay*. Hence, lazification consists in marking which expressions can be evaluated lazily (via delay) and which expressions must be evaluated right away (via enforce).

6 Conclusion

This paper has introduced a compiler optimization that replaces eager evaluation of formal parameters with lazy evaluation, whenever such transformation is deemed profitable. The code transformation that this paper discusses is a form of aggressive speculation: under the right circumstances, which a profiler can identify, it can improve even highly optimized binaries. As an example, we have been able to observe a statistically significant speedup of almost 5% over `clang -O3` when running an optimized version of `perlbench`, one of the largest programs in SPEC CPU2017. Speedups above 10% on smaller programs from the LLVM test suite could also be measured.

Data Availability Statement. The implementation of lazification is available at https://github.com/lac-dcc/wyvern. An interesting future work would be to extend this implementation to apply lazification onto multiple function arguments, factoring out redundancies between closures.

Acknowledgments

The authors would like to thank CNPq (grant 406377/2018-9), FAPEMIG (grant PPM-00333-18) and CAPES (Edital PRINT). We thank the anonymous reviewers and our colleagues from UFMG's Compilers Lab for many useful suggestions.

References

[1] Gagan Agrawal, Joel Saltz, and Raja Das. 1995. Interprocedural Partial Redundancy Elimination and Its Application to Distributed Memory Compilation. In *PLDI*. ACM, New York, NY, USA, 258–269. https://doi.org/10.1145/207110.207157

[2] Andrei Alexandrescu. 2010. *The D programming language*. Addison-Wesley, Bostom, MA, US.

[3] Andrew W. Appel and Jens Palsberg. 2003. *Modern Compiler Implementation in Java* (2nd ed.). Cambridge University Press, USA.

[4] John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, et al. 1960. Report on the algorithmic language ALGOL 60. *Commun. ACM* 3, 5 (1960), 299–314.

[5] David W. Binkley and Keith Brian Gallagher. 1996. Program Slicing. In *Advances in Computers*, Marvin V. Zelkowitz (Ed.). Vol. 43. Elsevier, Amsterdam, The Netherlands, 1–50. https://doi.org/10.1016/S0065-2458(08)60641-5

[6] Edwin C. Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

[7] Edwin C. Brady. 2013. Idris: General Purpose Programming with Dependent Types. In *PLPV*. ACM, New York, NY, USA, 1–2. https://doi.org/10.1145/2428116.2428118

[8] Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. 2002. Fast Copy Coalescing and Live-Range Identification. In *PLDI*. ACM, New York, NY, USA, 25–32. https://doi.org/10.1145/512529.512534

[9] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. 1986. Strictness Analysis for Higher-Order Functions. *Sci. Comput. Program.* 7, C (jun 1986), 249–278. https://doi.org/10.1016/0167-6423(86)90010-9

[10] John M. Chambers. 2020. S, R, and Data Science. *Proc. ACM Program. Lang.* 4, HOPL, Article 84 (jun 2020), 17 pages. https://doi.org/10.1145/3386334

[11] Stephen Chang. 2013. Laziness by Need. In *ESOP*. Springer-Verlag, Berlin, Heidelberg, 81–100. https://doi.org/10.1007/978-3-642-37036-6_5

[12] Stephen Chang and Matthias Felleisen. 2012. The Call-by-Need Lambda Calculus, Revisited. In *ESOP*. Springer-Verlag, Berlin, Heidelberg, 128–147. https://doi.org/10.1007/978-3-642-28869-2_7

[13] Stephen Chang and Matthias Felleisen. 2014. Profiling for Laziness. *SIGPLAN Not.* 49, 1 (jan 2014), 349–360. https://doi.org/10.1145/2578855.2535887

[14] Yishen Chen, Charith Mendis, and Saman Amarasinghe. 2022. All You Need is Superword-Level Parallelism: Systematic Control-Flow Vectorization with SLP. In *PLDI*. ACM, New York, NY, USA, 301–315. https://doi.org/10.1145/3519939.3523701

[15] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. 1997. A New Algorithm for Partial Redundancy Elimination Based on SSA Form. In *1997*. ACM, New York, NY, USA, 273–286. https://doi.org/10.1145/258915.258940

[16] Chris Clack and Simon L. Peyton Jones. 1985. Strictness analysis — a practical approach. In *Functional Programming Languages and Computer Architecture*, Jean-Pierre Jouannaud (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 35–49.

[17] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *POPL*. ACM, New York, NY, USA, 25–35. https://doi.org/10.1145/75277.75280

[18] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (1987), 319–349. https://doi.org/10.1145/24039.24041

[19] Breno Campos Ferreira Guimaraes. 2022. *Automatically Transforming Call-by-Value into Call-by-Need.* Master's thesis. Universidade Federal de Minas Gerais.

[20] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. 2008. *The C# programming language.* Pearson Education, Hoboken, NJ, USA.

[21] Yann Herklotz, Delphine Demange, and Sandrine Blazy. 2023. Mechanised Semantics for Gated Static Single Assignment. In *CPP*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/3573105.3575681

[22] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A History of Haskell: Being Lazy with Class. In *HOPL*. ACM, New York, NY, USA, 12–1–12–55. https://doi.org/10.1145/1238844.1238856

[23] Apple Inc. 2021. *The Swift Programming Language.* Apple Inc., Cupertino, CA, USA.

[24] Choonki Jang, Jaejin Lee, Bernhard Egger, and Soojung Ryu. 2012. Automatic Code Overlay Generation and Partially Redundant Code Fetch Elimination. *ACM Trans. Archit. Code Optim.* 9, 2, Article 10 (jun 2012), 32 pages. https://doi.org/10.1145/2207222.2207226

[25] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. 1999. Partial Redundancy Elimination in SSA Form. *ACM Trans. Program. Lang. Syst.* 21, 3 (may 1999), 627–676. https://doi.org/10.1145/319301.319348

[26] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1992. Lazy Code Motion. In *PLDI*. ACM, New York, NY, USA, 224–234. https://doi.org/10.1145/143095.143136

[27] T.-M. Kuo and P. Mishra. 1987. On Strictness and Its Analysis. In *POPL*. ACM, New York, NY, USA, 144–155. https://doi.org/10.1145/41625.41638

[28] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*. IEEE Computer Society, USA, 75.

[29] Bruce J MacLennan. 1986. *Principles of programming languages: design, evaluation, and implementation.* Holt, Rinehart & Winston, New York, NY, US.

[30] Alan Mycroft. 1981. *Abstract interpretation and optimising transformations for applicative programs.* Ph.D. Dissertation. University of Edimburgh.

[31] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. 1990. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages. In *PLDI*. ACM, New York, NY, USA, 257–271. https://doi.org/10.1145/93542.93578

[32] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.* 11, 9 (may 2018), 1002–1015. https://doi.org/10.14778/3213880.3213890

[33] Rodric M. Rabbah and Krishna V. Palem. 2003. Data Remapping for Design Space Optimization of Embedded Memory Systems. *ACM Trans. Embed. Comput. Syst.* 2, 2 (may 2003), 186–218. https://doi.org/10.1145/643470.643474

[34] Bin Ren and Gagan Agrawal. 2011. Compiling Dynamic Data Structures in Python to Enable the Use of Multi-Core and Many-Core Libraries. In *PACT*. IEEE Computer Society, USA, 68–77. https://doi.org/10.1109/PACT.2011.13

[35] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. https://doi.org/10.2307/1990888

[36] Josep Silva. 2012. A Vocabulary of Program Slicing-Based Techniques. *ACM Comput. Surv.* 44, 3, Article 12 (jun 2012), 41 pages. https://doi.org/10.1145/2187671.2187674

[37] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin Slicing. In *PLDI*. ACM, New York, NY, USA, 112–122. https://doi.org/10.1145/1250734.1250748

[38] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. 2016. Optimizing R Language Execution via Aggressive Speculation. In *DLS*. ACM, New York, NY, USA, 84–95. https://doi.org/10.1145/2989225.2989236

[39] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.* 13, 4s, Article 134 (apr 2014), 25 pages. https://doi.org/10.1145/2584665

[40] Frank Tip. 1994. *A Survey of Program Slicing Techniques*. Technical Report. IBM T. J. Watson Research Center, NLD.

[41] Peng Tu and David Padua. 1995. Efficient Building and Placing of Gating Functions. In *PLDI*. ACM, New York, NY, USA, 47–55. https://doi.org/10.1145/207110.207115

[42] David Turner. 1986. An overview of Miranda. *ACM Sigplan Notices* 21, 12 (1986), 158–166.

[43] Adriaan Van Wijngaarden, Barry J Mailloux, John EL Peck, Cornelius HA Koster, M Sintzoff, CH Lindsey, LGLT Meertens, and RG Fisker. 1969. Report on the algorithmic language ALGOL 68. *Numer. Math.* 14, 2 (1969), 79–218.

[44] P. Wadler. 1988. Strictness Analysis Aids Time Analysis. In *POPL*. ACM, New York, NY, USA, 119–132. https://doi.org/10.1145/73560.73571

[45] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

[46] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A Brief Survey of Program Slicing. *SIGSOFT Softw. Eng. Notes* 30, 2 (mar 2005), 1–36. https://doi.org/10.1145/1050849.1050865

[47] Guoqiang Zhang and Xipeng Shen. 2021. Best-Effort Lazy Evaluation for Python Software Built on APIs. In *ECOOP (LIPIcs)*, Vol. 194. Schloss Dagstuhl, Leibniz-Zentrum für Informatik, 15:1–15:24. https://doi.org/10.4230/LIPIcs.ECOOP.2021.15

[48] Yingzhou Zhang. 2021. SymPas: Symbolic Program Slicing. *J. Comput. Sci. Technol.* 36, 2 (2021), 397–418. https://doi.org/10.1007/s11390-020-9754-4

[49] Peng Zhao and Jose Nelson Amaral. 2005. Function Outlining and Partial Inlining. In *SBAC-PAD*. IEEE Computer Society, USA, 101–108. https://doi.org/10.1109/CAHPC.2005.26

A Appendix

The appendix describes some aspects of our approach that are not original—ideas that have already been presented in previous work. In particular, we provide more details on how we implement profiling. Our implementation of profiling follows much of Chang and Felleisen [13]'s ideas. The key differences are in terms of engineering: we are instrumenting the low-level, imperative, LLVM intermediate representation. Chang and Felleisen was instrumenting a high-level, side-effect free, representation of the Racket abstract syntax tree.

A.1 More on the Identification of Profitability

As mentioned in Section 3.2, our static analysis provides a conservative estimate on the profitability of lazifying a given function call. If the static promising paths in the callee are seldom exercised during execution, lazification will simply add extra work while rarely avoiding redundancies. In such cases, lazification is likely to lead to performance degradation, as Example A.1 demonstrates.

Example A.1. In the lazified program shown in Figure 2, the promising path occurs when the check `key != 0` in line 15 is false. Whether this is the case depends on the contents of the input string `s0`. For an input of ten thousand strings, 99.9% of which cause `key != 0` to be true, and $\texttt{N} = 10^6$, this program runs in 21.4 seconds on a single-core x86 machine clocked at 2.2GHz. Its non-optimized counterpart, shown in Figure 1, runs in 11.7 seconds, nearly twice as fast. This difference increases with the size of `N`.

To avoid the scenario described in Example A.1, we resort to profile-guided optimization. The instrumentation-based profiler augments the input program to record data on the dynamic uses of formal parameters for each function call. Following the usual practice, the target program is profiled with a set of training inputs. The program is then compiled a second time, and the available profiling data is used to decide when to apply call-by-need for a given call site. Notice that only promising functions (see Def. 3.1) are instrumented. Thus, given a promising function f, for each call $f_i(a_1, a_2, \ldots, a_j)$, we keep track of two statistics:

- num_calls_i : the number of times the function call is executed.
- $eval\_count[j]$: for each real parameter a_j, the number of times it was evaluated at least once for each call f_i.

Instrumentation adds code to track these data on two fronts. (i) For every call site in the program, we add a call to an auxiliary function immediately preceding it, which tags it as the active call site. This tagging is also used to uniquely identify call sites, so we can map them back to the LLVM IR. (ii) For every promising function, we add:

- To its b_{start} entry node, an initialization of a bitmap $bits[j]$ which tracks whether each of its j formal parameters were used at least once during that execution
- For each b_i node containing a use of a formal parameter a_i, an instruction to set the $i-th$ bit of $bits[]$
- To its b_{end} exit node, an auxiliary function to increment num_calls_i, as well as every $eval\_count_j$ whose $bits[j]$ is set

Given a profiled program, our instrumentation allows us to compute, for each call site, the *usage ratio* of each of its real parameters. The usage ratio of a parameter a_j in a call $f_i(\ldots, a_j, \ldots)$ is simply $\frac{eval\_count[j]}{num\_calls_i}$. The profile-guided version of the optimization is parameterized with a *threshold* value. When evaluating whether to lazify a call site, only parameters whose usage ratio is below the given threshold are lazified.

A.2 This Work in Perspective

Section 5 covers two lines of work that are directly related to this paper: function outlining and the implementation of lazy evaluation in functional programming languages. Nevertheless, there exists a large literature of programming techniques that are related to this work. This section goes over this literature. Notice that this section does not discuss further the implementation of lazy programming languages. These techniques differ from the present work insofar as languages such as Haskell have been implemented with lazy evaluation in mind. They do not undergo a "lazification" process that modifies eager evaluation. The construction of thunks happens at the level of the abstract syntax tree. The dependencies necessary to run the thunk are computed via a back traversal of this AST. This section also does not cover the implementation of lazy evaluation as a design-pattern. The manual implementation of lazy evaluation is well-known, as described in Jonathan Müller's blog (https://www.foonathan.net/2017/06/lazy-evaluation/). Yet, such approaches cannot easily be compared with the techniques discussed in this paper, for they are not automatic.

Partial Redundancy Elimination. We call *partially redundant code* computation that might execute two or more times, depending on the program flow. Many classic compiler-related papers discuss the implementation of partial redundancy elimination [15, 25, 26]. Lazification bears a few similarities with these approaches. In particular, like Knoop et al. [26], we strive to meet *performance safety* within the unexercised path. In other words, we do not introduce useless computation in the program. However, we assume the existence of a limited set of program points where computation can happen. In our case, code can only execute at the points where formal parameters are accessed in the optimized function call. The traditional approach to partial code motion, in turn, uses two data-flow analyses—available expressions and busy expressions—to determine where to move redundant code. These analyses run intra-procedurally: they do not require a program slicing algorithm, as they do not create closures containing redundant computations. We emphasize that contrary to Knoop et al. and other works on intra-procedural redundancy elimination, we incur a higher cost on the exercised path: the *thunk* containing the lazy expression must be loaded with arguments and invoked, whenever it becomes active.

Inter-procedural Partial Redundancy Elimination. The notion of partial redundancy elimination has been extended to an inter-procedural analysis by Agrawal et al. [1]. Further developments of Agrawal et al.'s ideas have been used in the generation of communication routines during the automatic parallelization of programs [24, 33, 34]. The ideas of Agrawal et al. are very similar to those earlier proposed by Knoop et al. [26]; however, Agrawal et al. propagate flow information outside the boundaries of functions. Additionally, they let code be moved out of the function where it was originally located; hence, effectively making lazy code motion an inter-procedural optimization. In many ways, Agrawal et al.'s approach seems to do the opposite of what we propose in this paper. Whereas we want to move computation from outside a procedure to inside it, Agrawal et al. is performing the inverse path: they extract expressions from a routine, and let them execute before this routine is called. Notice that this extraction process does not lead to the creation of new functions—they are not doing function outlining. Instead, expressions are extracted and moved *as is*, except that program symbols are renamed whenever necessary. Example A.2 illustrates intra and inter-procedural elimination of redundancies.

Example A.2. Figure 12 (a) shows an example of intra-procedural partial elimination of redundancies, as proposed by Knoop et al. [26]. In this case, the computation of **exp** is lowered to within the conditional block. Lowering avoids computing this expression if the body of the conditional is not visited by the program flow. Figure 12 (b) shows an example of inter-procedural redundancy elimination. The idea is similar to the intra-procedural case: the computation of **exp** is invariant within the loop; hence, it can be hoisted out of that loop, if it is extracted from function g.

```
a = exp();
if (b) {
  c = a;
} else {
  …
}
```

```
if (b) {
  a = exp();
  c = a;
} else {
  …
}
```

(a)

```
f() {
  while(a) {
    g(a);
    update(a);
  }
}

g() {
  exp();
  use(a);
}
```

```
f() {
  exp();
  while(a) {
    g(a);
    update(a);
  }
}

g() {
  use(a);
}
```

(b)

Figure 12. (a) Partial redundancy elimination; (b) Inter-procedural Partial redundancy elimination.

Strictness Analysis. In the late eighties, different research groups invented techniques to transform lazy evaluation of actual parameters into eager evaluation [9, 16, 27, 44]. These transformations rely on a data-flow analysis presented by Mycroft [30] in his Ph.D thesis, which became later known as *strictness analysis*. In the words of Burn et al. [9]: "*Mycroft shows that by annotating functions with strictness information, it is possible to optimise functional program execution by using eager evaluation techniques wherever this can be done without violating the lazy semantics*". This type of "strictification" is the opposite of what we propose in this paper. The goal of this paper is to lazify the computation of actual parameters that otherwise would be evaluated eagerly. Mycroft-style strictification, in turn, aims at making strict the evaluation of arguments that, otherwise, would be processed lazily.

Example A.3. Figure 13 illustrates the application of strictness analysis. The parameters x is evaluated multiple times into the body of function g, whereas y is only evaluated once. In this case, it pays off to pre-compute the value of y before invoking function g, as that avoid multiple instantiations of a closure that will only be executed once.

```
g(x, y) {
  if (x == 0) {
    return y;
  } else {
    return g(x-1, y)
  }
}

g(1000, lazy_exp())
```

```
g(x, y) {
  if (x == 0) {
    return y;
  } else {
    return g(x-1, y)
  }
}

g(1000, eager_exp())
```

Figure 13. Example of strictness analysis.

API-level Lazification. Recently, Zhang and Shen [47] have released `Cunctator`, a framework to replace some library calls, in Python, by equivalent calls that are lazily evaluated. For instance, `Cunctator` can replace calls to `NumPy` routines with equivalent structures from `WeldNumPy`, or replace `panda`'s dataframes with `spark` dataframes with the same interface. In many ways, `Cunctator` moves from the programmer the burden of choosing lazy data-structures whenever such choice is profitable. Similar ideas can be found in a few domain-specific languages, such as `Weld` [32] or `Delite` [39]. We emphasize that this line of work is very different from what we propose in this paper. Zhang and Shen's analysis is dynamic: the profitability of lazification is measured at running time; our work, in turn, is fully static. Second, Zhang and Shen's (and also Palkar et al. and Sujeeth et al.'s) approach is applied at a higher-level than what we do: they replace APIs, e.g., one function call with another of similar purpose, as Example A.4 shows. Therefore, transformations such as slicing or gating are of no service in their case.

Example A.4. Figure 14 illustrates how Zhang and Shen [47]'s style of lazification works. In that case, calls to eager library functions are replaced with calls to lazy functions that implement the same operations.

```
a = eager_exp();
b = eager_exp();
c = eager_exp(a, b);
update(a);
d = eager_exp(c);
print(d);
```

```
a = lazy_exp();
b = lazy_exp();
c = lazy_exp(a, b);
eval(c);
update(a);
d = lazy_exp();
eval(d);
print(d);
```

Figure 14. API-level lazification, as implemented by `Cunctator`[47].

Program Slicing. Program slicing is a core component of this work. We believe that the original treatment of this topic appears in the work of Weiser [45]. A number of classic surveys constitute today the canonical literature on the subject [5, 40, 46]. More recently, Silva [36] have summarized most of the new slicing algorithms published after the 2000's. The key technique to identify a program slice, given a slice criterion, is already present in Weiser's seminal work. Yet, we found it surprisingly difficult to derive an algorithm to extract closures from program slices performed on a program in Static-Single Assignment form. Our difficulties stemmed from two observations. First, much work on program slicing does not aim at producing executable codes; rather, they are used for debugging—a goal that finds in the notion of *thin slicing* [37] its most well-known representative. Second, the classic definition of program slice considers statements, not variable uses, as slice criteria. As a consequence, control dependencies surrounding these statements end up incorporated into the program slice. Example A.5 illustrates how this notion of program slice differs from the concept that we use in this paper.

Example A.5. The classic definitions of program slicing [5, 40, 46], including the more modern treatment found in the work of Silva [36], would not consider the use of variable **x** in Figure 4 (Left) as a slice criterion. Rather, they would consider as a slice criterion the instruction at Line 19, e.g., `r = bar(x)`. In this case, the slice contains the conditional at Line 18, plus the switch at Line 16 of Figure 4 (Left). None of these lines are part of the slice that we create, as Figure 6 shows.

The difference between our work and previous techniques, which Example A.5 illustrates, persists even in recent work that performs program slicing in the context of the LLVM compiler, using SSA-form programs [48]. Thus, we believe that the declarative algorithm in Definition 3.10, plus the use of gating predicates, as stated in Definition 3.6, are contributions of this paper. We emphasize that this combination of techniques works for the construction of idempotent slices.