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

BRENO CAMPOS FERREIRA GUIMARÃES, Universidade Federal de Minas Gerais, Brazil
FERNANDO MAGNO QUINTÃO PEREIRA, Universidade Federal de Minas Gerais, Brazil

In strict programming languages, parameters of functions are evaluated before these functions are invoked. In lazy programming languages, the evaluation happens after invocation, if the formal parameters are effectively used. Languages are strict or lazy by default, sometimes providing developers with constructs to modify this expected evaluation semantics. In this case, it is up to the programmer to decide when to use either approach. This paper moves this task to the compiler by introducing the notion of "*lazification*" of function arguments: a code transformation technique that replaces strict with lazy evaluation of parameters whenever such modification is deemed profitable. This transformation involves a static analysis to identify function calls that are candidates for lazification, plus a code extraction technique that generates 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 onto LLVM 14.0, and have applied it onto hundreds of C/C++ programs from the LLVM test-suite and from SPEC CPU2017. During this evaluation, we could observe statistically significant speedups over `clang -O3` on some large programs, including a speedup of 11.1% on Prolang's `Bison` and a speedup of 4.6% on SPEC CPU2017's `perlbench`, which has more than 1.7 million LLVM instructions once compiled with `clang -O3`.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: lazy evaluation, programming languages, compilers

1 INTRODUCTION

Several programming languages furnish developers with ways to enable communication with the compiler. These language constructs—type qualifiers or meta-language annotations—do not describe *what* the compiler should implement, but rather *how* a given piece of code should be implemented. For instance, GCC has an extensive list of C/C++ extensions[1], through which developers can guide the application of code optimizations. LISP provides the *declare* clause, which is used with similar purpose [Steele 1990].

Although useful, these constructs tend to become obsolete with the evolution of compilation technology. For example, the **register** specifier in C/C++, once used to suggest a variable should be stored in a CPU register, has been made largely irrelevant by automatic register allocation, to the point where it is ignored by C compilers in most scenarios[2]. This qualifier was deprecated in C++11, and is now unused since C++17 [ISO 2011, 2017]. Other examples of C/C++ keywords whose usefulness has been partially or completely superseded by compiler analyses exist, such as **inline** [Ayers et al. 1997; Chang and Hwu 1989; Chang et al. 1992] and **restrict** [Diarra 2018; Sperle Campos et al. 2016].

[1]https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html
[2]https://gcc.gnu.org/onlinedocs/gcc/Explicit-Register-Variables.html

Authors' addresses: Breno Campos Ferreira Guimarães, brenosfg@dcc.ufmg.br, Universidade Federal de Minas Gerais, 6637 President Antonio Carlos Av., Belo Horizonte, Minas Gerais, Brazil, 31270-901; Fernando Magno Quintão Pereira, fernando@dcc.ufmg.br, Universidade Federal de Minas Gerais, 6637 President Antonio Carlos Av., Belo Horizonte, Minas Gerais, Brazil, 31270-901.

Nevertheless, developers still have the upper hand on at least one aspect of code generation: the *evaluation strategy*. Most programming languages are *strict*, meaning that actual parameters are evaluated before being passed to functions. However, 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 [Hejlsberg et al. 2008], Swift's **lazy** property [Inc. 2021], D's **lazy** storage class [Alexandrescu 2010] and Idris's **Lazy** type qualifier [Brady 2013a,b]. In all these cases, the onus of reasoning about the safety and profitability of lazy evaluation falls on the programmer.

The Contributions of this Work. In this paper, we argue that compilers can lift such responsibility from users. To this end, we propose *lazification of function arguments*, a compiler optimization which identifies function calls whose actual parameters may benefit from being lazily evaluated, and transforms the program to carry out such strategy, effectively implementing selective call-by-need. Our optimization first analyzes the program's control flow, marking formal parameters[3] which are candidates for *lazification*. Then, call sites are analyzed to determine the safety and profitability of lazily evaluating corresponding actual instances of such parameters. For safe calls deemed profitable, we leverage Program Slicing [Binkley and Gallagher 1996; Tip 1994; Weiser 1984; Xu et al. 2005] 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. This optimization leverages profile information to better guide its profitability analysis. The aforementioned *modus operandi* is the materialization of the following contributions:

Lazification: the description of an algorithm that can be applied onto the low-level intermediate representation of a compiler to replace call-by-value with call-by-need whenever such transformation is 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 have implemented our optimization on top of the LLVM compilation infrastructure [Lattner and Adve 2004]. Thus, analyses and transformations are performed on LLVM's intermediate representation. We evaluated the effectiveness of our prototype onto the C/C++ programs in the LLVM test-suite and the SPEC CPU2017 collection. In total, we run lazification on 772 C/C++ programs optimized with `clang -O3`. Unguided *lazification*, i.e., without the support of profiling data, yields performance regressions onto these heavily optimized codes: a slowdown of almost 10% on average. However, once profiling data becomes available, the optimization starts paying off. Average speedups are still low, for they are distributed across many benchmarks: we could observe a speedup of 1.10% with a confidence interval of 95% when using our most conservative optimization setup. However, once individual benchmarks are analyzed, we can perceive some significant performance improvements. In two of the benchmarks of the LLVM test suite, Prolang's `Bison` and `city`, speedups were above 10%. In SPEC CPU2017's `perlbench`, one of the largest benchmarks we have, the observed speedup was 4.6%. Our optimization increases code: on its most aggressive mode, i.e., without any profiling cutoff, the average expansion was of 18.9%. In the most conservative mode, average expansion was of 9.2%.

Software. The ideas in this report are implemented in LLVM 14.0. Our implementation is publicly available under the `GPL 3.0` license at https://github.com/lac-dcc/wyvern.

[3]Following typical programming language 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.

2 OVERVIEW

This section motivates the use of lazy evaluation in general purpose programming languages (Sec. 2.1), and explains how programs can be transformed automatically to support it (Sec. 2.2).

2.1 The Evaluation Zoo

The *evaluation strategy* of a programming language determines how expressions in the language should be evaluated to yield their final values. The decision of which strategy to employ is particularly relevant in the context of procedure calls. When expressions are passed as arguments to procedures, this strategy dictates when and how they should be evaluated. While several evaluation strategies exist, this paper concerns three particular strategies:

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 pre-computed $M(r)$ 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 [Hudak et al. 2007], R [Chambers 2020] and Miranda [Turner 1986]. 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 [Clack and Peyton Jones 1985; Mycroft 1981]. Such pattern is still employed nowadays[4]. A second reason is the unintuitive nature of non-strict evaluation in the presence of side effects, as discussed by Stadler et al. [2016] in the context of the R programming language. ALGOL60, for instance, was the first language to introduce call-by-name semantics [Backus et al. 1960]. However, call-by-name led to confusing behavior, as famously evidenced by Jensen's device [MacLennan 1986]. Call-by-name would later be discarded in ALGOL68 in favor of call-by-value [Van Wijngaarden et al. 1969]. 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?*"[5] As a consequence, several features that are common in strict languages lack equivalents in the non-strict world. As an example, until very recently, semantic subtyping systems were unsound for non-strict semantics [Petrucciani et al. 2018].

[4]See Haskell's strictness analysis at https://wiki.haskell.org/Performance/Strictness.
[5]Discussion available at https://docs.idris-lang.org/en/v0.9.18.1/faq/faq.html

Strict evaluation is not, however, 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. In fact many strict programming languages provide programmers with means to implement selective non-strict evaluation, as Example 2.2 illustrates.

```
01  void dotimes(int count, (lazy) void exp)
02  {
03      for (int i = 0; i < count; i++)
04      exp();
05  }
06
07  void foo()
08  {
09      int x = 0;
10      dotimes(10, write(x++));
11  }
```

(a)

```
01  class DataImporter {
02    var filename = "data.txt"
03  }
04
05  class DataManager {
06    lazy var importer = DataImporter()
07    var data: [String] = []
08  }
09
10  let manager = DataManager()
11  manager.data.append("Some data")
12  manager.data.append("Some more data")
13  print(manager.importer.filename)
```

(b)

Fig. 1. Examples of lazy evaluation in strict languages. (a) An example of the lazy storage class in D, taken directly from the language's documentation (https://dlang.org/articles/lazy-evaluation.html). (b) An example of Swift's lazy stored properties, also taken from the language's documentation (https://docs.swift.org/swift-book/LanguageGuide/Properties.html).

Example 2.2. Figure 1 (a) shows an example used in D's documentation of its **lazy** storage class. The expression **exp** received in line 1 is declared as **lazy**. It is only evaluated in line 4, when it is needed. Thus, the expression **write(x++)** in line 10 is computed lazily, rather than being computed at the time function **dotimes** is called.

Lazy evaluation is also common in the initialization of data. In this case, it provides programmers with a built-in implementation of the *proxy* design pattern. Thus, instead of initializing large objects that might not necessarily be used later on, initialization happens on demand, that is, only when needed. Example 2.3 illustrates this pattern in the Swift programming language.

Example 2.3. Figure 1 (b) Shows a program taken from Swift's documentation of its lazy modifier for stored properties. Class **DataManager** in line 5 contains a property of class **DataImporter**, bound to member variable **importer**. However, this property is declared as **lazy**, implying the property is only initialized when needed. Therefore, the creation of a **DataManager** in line 10 does not trigger the initialization of its **importer** property. This initialization is deferred to line 13, when it is first referenced.

2.2 Lazification by Examples

Examples 2.2 and 2.3 show situations where it is up to the programmer to recognize profitable opportunities where lazy evaluation might be safely applicable. The optimization described in this paper moves this task to the compiler. In other words, we describe a compiler optimization that (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 2.4, which shows a situation where the optimization proposed in this paper delivers a large benefit.

Example 2.4. The logical conjunction (&&) at Line 02 in Figure 2 (a) 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 15 of Figure 2 (a). This fact is unfortunate, because the computation of the second variable, **value**, involves a potentially heavy load of computation, comprising the code from Line 09 to Line 14 of Figure 2 (a).

```
01  void callee(int key, int value, int N) {
02    if (key != 0 && value < N) {
03      printf("User has access\n");
04    }
05  }
06
07  void caller(char *s0, int *keys, int N) {
08    int key = atoi(s0);
09    int value = -1;
10    for (int i = 0; i < N; i++) {
11      if (keys[i] == key) {
12        value = i;
13      }
14    }
15    callee(key, value, N);
16  }
```

(a) Original Program

```
01  struct thunk {
02    int (*fptr)(struct thunk *);
03    char *s0;      // The table of free
04    int *keys;     // variables that forms
05    int N;         // the closure.
06    int val;       // Support to one-shot
07    bool memo;     // memoization.
08  };
```

(b) Optimized Program

```
09  void callee(int key, struct thunk *thk, int N) {
10    if (key != 0 && thk->fptr(thk) < N) {
11      printf("User has access\n");
12    }
13  }
14
15  void caller(char *s0, int *keys, int N) {
16    int key = atoi(s0);
17    struct thunk value_thunk;
18    value_thunk.fptr = &slice_callee_value;
19    value_thunk.memo = false;
20    value_thunk.s0 = s0;
21    value_thunk.keys = keys;
22    value_thunk.N = N;
23    callee(key, &value_thunk, N);
24  }
25
26  int slice_callee_value(struct thunk *thk) {
27    if (thk->memo) {
28      return thk->val;
29    }
30    int key = atoi(thk->s0);
31    int value = -1;
32    for (int i = 0; i < thk->N; i++) {
33      if (thk->keys[i] == key) {
34        value = i;
35      }
36    }
37    thk->val = value;
38    thk->memo = true;
39    return value;
40  }
```

Fig. 2. (a) Example of a program that benefits from the lazy evaluation of parameter `value`, once function `callee` is invoked at Line 15. (b) Lazified version of the program. Gray boxes highlight the interventions proposed in this paper. These interventions are implemented at the level of the LLVM intermediate representation; however, their effects are shown as C code for the sake of readability.

The computation of **value** in Figure 2 (a), as discussed in Example 2.4, is a promising candidate for *lazification*. Figure 2 (b) shows the code that results from this optimization, as proposed in this paper[6]. Lazification, as engineered in this paper, is implemented as a form of *function outlining*: 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 Figure 2 (b). 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. This function appears in lines 26 to 40 of Figure 2 (b). An invocation to this closure in Line 10 of Figure 2 (b) replaces the use of formal parameter `value`, which was computed eagerly in the

[6]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.

original program. There is no upper limit to the amount of speedup that can be achieved by this style of low-level function lazification. Example 2.5 illustrates this aspect of our approach.

Example 2.5. If the test `key != 0` is often false in Line 02 of Figure 2 (a), then lazy evaluation becomes attractive. In this case, the speedup that lazification delivers onto the program in Figure 2 (a) is linearly 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 (b) runs in 0.060s. This difference increases with `N`.

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. Thus, to make lazification practical, we have made it profile-guided, as we explain in Section 3.5. Nevertheless, our current implementation of lazification is able to transform the program in Figure 2 (a) into the program in Figure 2 (b) in a completely automatic way: it requires no annotations or otherwise any other intervention from users.

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.

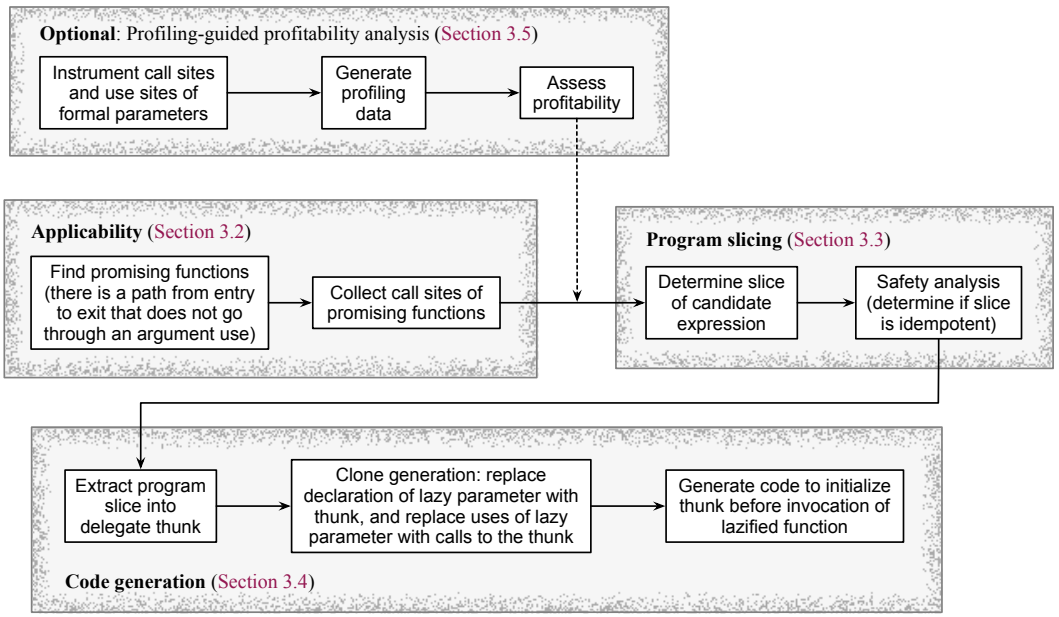

Fig. 3. Overview of the different steps that constitute our implementation of lazification of function arguments.

3.1 The Underlying Language

Our presentation, in this section, uses a low-level language, formed by programs in Static Single Assignment (SSA) form [Cytron et al. 1989]. These programs are represented as *control-flow graphs*

(CFG). 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.

Example 3.1. Figure 4 (a) shows a program, whose control-flow graph appears in Figure 4 (b). Similarly, Figure 4 (c) shows the CFG of the program seen in Figure 4 (d).

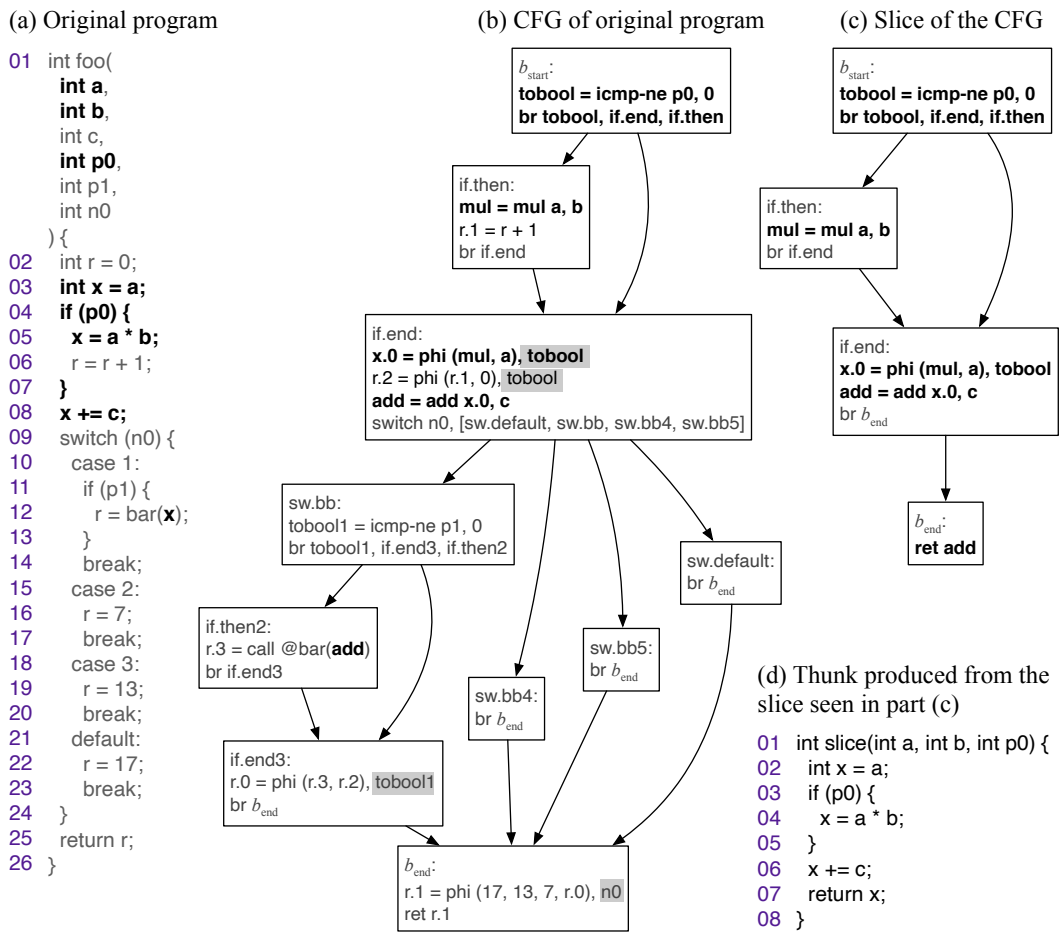

Fig. 4. Example of program slice, as extracted by the algorithm described in Section 3.3 of this paper. The gray boxes in part (b) highlight the predicates used to gate phi-functions, as explained in Section 3.3.1.

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 [1953]'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 3.2 (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 3.3. Function `callee` in Figure 2 (a) 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 (a) is not promising regarding any of its formal parameters, except `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`.

Depth-First Search. Whether a function is *promising* can be computed by a depth-first-search through its CFG. Start a traversal by visiting its initial b_{start} node. 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 3.2 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*. Section 3.5 explains how profitability can be estimated through profiling.

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 a invocation of f. Definition 3.4 revisits the notion of backward slice, adapting to our context the formalization proposed by Weiser [1984] in the early eighties.

Definition 3.4 (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, because he assumed an imperative setting, where a slice criterion would be a *program statement*. However, in our context, we need a *variable*, e.g., a program symbol, as a slice criterion, for we are interested in the contents of this variable, once it is used as the actual argument of a function call. Example 3.5 illustrates our notion of backward slice and Section 5 provides further discussion on these differences.

Example 3.5. Figure 4 (a) shows a program written in C. We let the use of variable **x** at Line 12 be a slice criterion. The backward slice that computes this slice criterion, e.g., (**x**, Line 12), appears in Figure 4 (a) in boldface. Figure 4 (d) shows the delegate function derived from this backward slice.

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 the classic definition of Ferrante et al. [1987], we recognize two types of dependencies: *data* and *control*, concepts that Definition 3.6 revisits. Like Definition 3.4, Definition 3.6 adapts the early notion of dependency stated by Ferrante et al. [1987]. However, whereas in that setting Ferrante et al. were interested in program statements, we care for dependencies between program symbols:

Definition 3.6 (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 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 3.6 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* [Tu and Padua 1995]. 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 3.7 provides an algorithmic denotation of gates, which Example 3.8 illustrates.

Definition 3.7 (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 3.8. The boolean predicate `tobool` in Figure 4 (b) 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 `sw.epilog`. Gates appear in gray boxes in Figure 4 (b).

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 3.7 leads to a simplified version of the so called *Gated Static Single Assignment* representation [Ottenstein et al. 1990]. The only difference between gating, as implied by Definition 3.7 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.

3.3.2 Identification of Slice Boundaries. The algorithm that we use in this paper to produce program slices contains two *boundary conditions* and three *nullifying conditions*. The nullifying condition prevents a slice from being created. The boundary conditions, in turn, stop the backward propagation of dependencies when computing slices. We adopt the following boundary conditions:

Loops: If the slice criterion (v, ℓ) is such that ℓ exists inside a loop, then the slice cannot contain variables that vary with the loop.

Functions: If f is the function that contains the slice criterion (v, ℓ), then the slice will contain only variables defined within f.

Loop Boundary. The first of these criteria, **Loops**, is necessary for correctness: the delegate function created out of a slice must be invariant in the context where it is called. In other words, the values used in the computation of this function cannot vary with the loop that encloses the caller that will receive the *thunk* containing it. In terms of implementation, the **Loop** boundary condition implies that arguments of phi-functions in loop headers are not considered as part of the backward slice. Example 3.9 illustrates this observation.

Example 3.9. Figure 5 (a) shows a program, and the backward slice that refers to the criterion (**x**, Line 08). The instructions that make up this slice are highlighted in Figure 5 (b). Figure 5 (c) shows the corresponding delegate function, in high-level C. 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 (d) shows the same program, this time with the criterion inside the loop (Line 07). Because a slice must be invariant, it comprises only values defined in a single iteration of the loop. Notice that it can still depend on loop-invariant values, such as **c**. However, it cannot depend on **add** or **a**, the arguments of the phi-function in block **while.cond** of Fig. 5 (e). This boundary condition exists because the value of **x.0**, the variable computed by that phi-function, is loop variant.

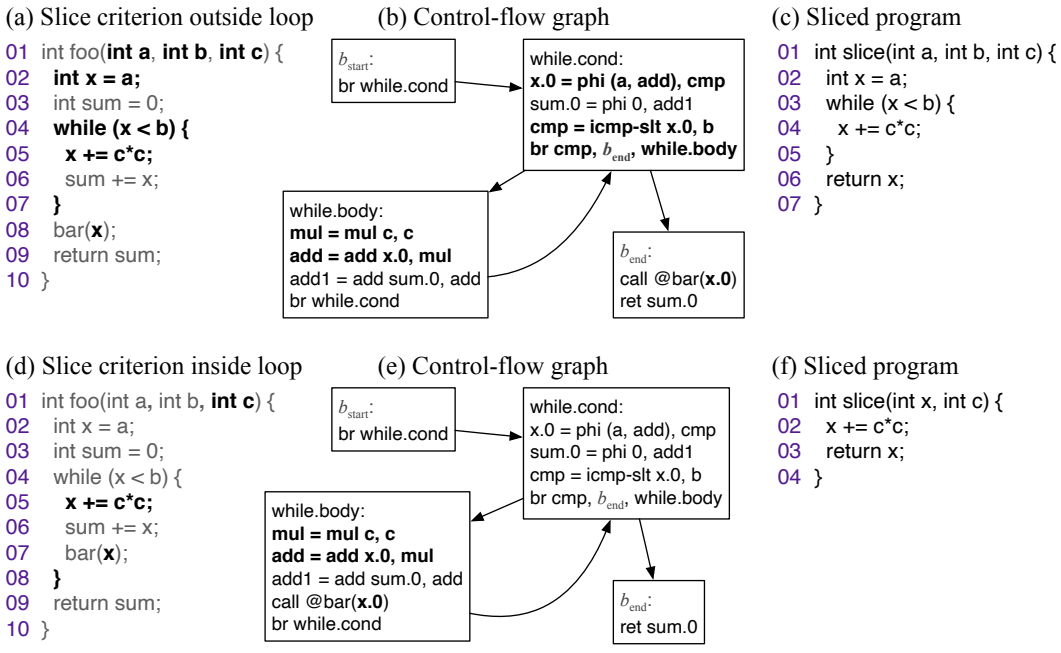

Fig. 5. (a-c) Example of program slice whose criterion lays outside a loop. (d-f) Same example, this time with slice criterion inside a loop.

Function Boundary. Although the optimization proposed in this paper is inter-procedural, our slices do not propagate backwardly from callee into caller functions. This restriction is not a fundamental limitation of our approach. As we shall see in Section 5, there exists a whole body of literature on inter-procedural code motion that would already give us the theoretical equipment to extend slices past function definitions. However, we opted to restrict slices to within the scope of functions for simplicity. Notice that slices can still contain function calls—what is forbidden is to contain dependencies that go past the formal parameters of the function that contains the slice criterion. Example 3.10 shows an actual slice that we produce, containing a function invocation.

```
01  __attribute__((pure))
02  bool is_num(char *str, int N) {
03    for (int i = 0; i < N; i++) {
04      if (str[i] < '0' || str[i] > '9') {
05        return false;
06      }
07    }
08    return true;
09  }
10
11  int caller(char *s0, char *s1, int N) {
12    bool is_num0 = is_num(s0, N);
13    bool is_num1 = is_num(s1, N);
14    return target(s0, s1, is_num0, is_num1, N);
15  }
```

(a) Original Program

```
01  bool slice_target_is_num1(struct thunk *thk) {
02    if (thk->memo)
03      return *(thk->value);
04    bool ret = is_num(thk->s1, thk->N);
05    return ret;
06  }
07
08  int caller(char *s0, char *s1, int N) {
09    bool is_num0 = is_num(s0, N);
10    struct thunk is_num1_thk;
11    is_num1_thk.fun_ptr = &slice_target_is_num1;
12    is_num1_thk.memo = false;
13    is_num1_thk.s1 = s1;
14    is_num1_thk.N = N;
15    return target(s0, s1, is_num0, &is_num1_thk, N);
16  }
```

(b) Optimized Program

Fig. 6. (a) Original program, whose slice criterion is (**is_num1**, Line 14). (b) The *thunk* formed from the backward slice, which includes the call to function **is_num()** at Line 13 of part (a).

Example 3.10. Figure 6 shows an example of a slice that includes a function invocation: the call to function **is_num()**, at Line 13 of Figure 6 (a). However, although the backward propagation of dependencies can enter function calls, it cannot move past the formal parameters of the enclosing function. Therefore, these dependencies include, as stop conditions, the arguments **s0**, **s1** and **N** of function **caller**, where the slice criterion is defined. In other words, the dependency graph that forms the slice is rooted at these nodes.

Nullifying Conditions. In this paper we consider only slices that are *idempotent*, that is, that do not cause side effects. To implement this restriction, in practice, we consider as invalid slices containing the following two pieces of code:

(1) Instructions that store into memory.
(2) Invocations of library functions (functions without bodies)[7].
(3) Instructions that may trigger exceptions, in languages that support them.

Notice that slices can still contain read-only memory accesses, i.e., load instructions[8]. Nevertheless, store operations are forbidden. This restriction, coupled with the interdiction of library

[7] An exception is made for standard library functions which are known to be free of side-effects

[8] 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.

functions, effectively prevent the existence of slices containing side effects[9]. 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

3.4.1 Extraction via Outlining. 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.

Definition 3.11 (Order of Dominance). Given a control-flow graph $G = (V, E)$, and two vertices $b_0 \in V$ and $b_1 \in G$, 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*. Node b_n is the n^{th} dominator of node b if b_n dominates b, and the path from b_n to b on the dominance tree contains n nodes. Given a set of nodes B, and a basic block $b \notin B$, the *first* dominator of b is $b_n \in B$ such that b_n is the n^{th} dominator of b, and for any other node $b_m \in B$ that is the m^{th} dominator of b, we have that $n < m$.

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 3.11 to formalize the notion of a *sliced program*.

Definition 3.12 (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 $b_0 \rightarrow b_1 \in E_p$ for some $b_0 \in V_s$ and $b_1 \in V_s$, then $b_0 \rightarrow b_1 \in E_s$.
(2) If $b_0 \rightarrow b_1 \in E_p$ 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 $b_f \rightarrow b_1 \in E_s$, where b_f is the first dominator of b_1 in $V_s \setminus b$.
(3) If a block $b \in V_s$ contains a definition of every variable used in it, then E_s contains an edge $b_{start} \rightarrow b$.
(4) If b contains the slice criterion, then E_s contains the edge $b \rightarrow b_{end}$.

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. [2002] have demonstrated the essential SSA property for strict programs in general. We use Budimlic et al.'s observation to state Theorem 3.13.

Theorem 3.13. *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.*

[9]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)

> Let $\ell : u = \text{use}(v)$ be any instruction in b that uses a variable v. The definition of v, in the original program P, is either in b, or in a block b_d that dominates b. In the latter case, $b_d \in V_s$, by the definition of backward slice.

Example 3.14. Figure 7 shows four examples of sliced programs. The white blocks in the upper part of the figure do not contain instructions that pertain to the slice (they do not belong into V_s). Therefore, they are bypassed by Rule (2) in Definition 3.12. Notice that the **Loop** condition discussed in Section 3.3.2 ensures that the slice criterion cannot belong into cycles of data dependencies. Therefore, Rule (4) in Definition 3.12 always applies.

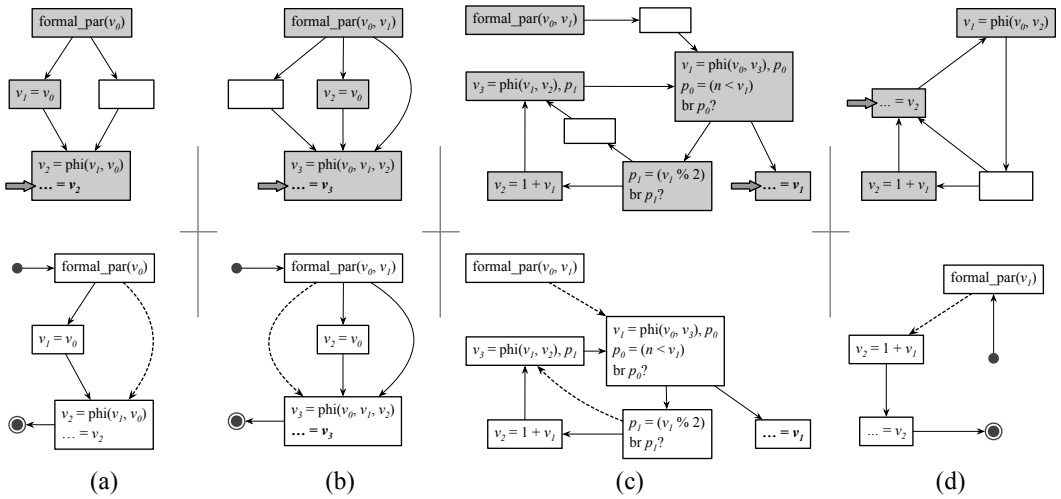

Fig. 7. Four examples of sliced programs produced after Definition 3.12. The arrow shows the slice criterion. The upper part of each figure shows the blocks of the original program touched by the slice criterion (the slice is highlighted in gray). The bottom part shows the resulting outlined function.

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 3.15. Figure 2 (b) shows the two fields that implement the memoization cache in Lines 6 and 7: `val` and `memo`.

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 3.16. Lines 27-29 of Figure 2 (b) show the conditional block created to check if the *thunk* has already been called. Lines 37 and 38 in Figure 2 (b) show the code that 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 3.17. In the optimized program shown in Figure 2 (b), the cloned `callee` function is shown in line 9, with its original `value` formal parameter now replaced by a *thunk* `thk`. The only use of `value` in the original callee from Figure 2 (a) was in the conditional `value < N` in line 3. This use is replaced by the invocation `thk->fptr(thk)` in line 10 of the new function.

Changes in the Caller. The second set of transformations is applied to the caller function. Given the actual parameter a being lazified, we replace the original definition of a by the declaration and initialization of the *thunk*. We employ the techniques described in Sections 3.4.1 and 3.4.2 to generate the $thunk_a$ type, consisting of:

- A function pointer, whose signature matches that of the delegate function generated by the algorithm described in Section 3.4.1.
- A list of member fields, whose types match those of the free variables on which the computation of a depends.
- The memoized value and memoization flag, as described in Section 3.4.2.

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 3.18 illustrates these interventions.

Example 3.18. Figure 2 (b) shows the *thunk* initialization code in Lines 17-22. This code replaces the original definition of `value` in Line 9 of Figure 2 (a).

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

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 3.19 demonstrates.

Example 3.19. In the lazified program shown in Figure 2 (b), the promising path occurs when the check `key != 0` in line 10 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 $N = 10^6$, this program runs in 21.4 seconds on a single-core x86 machine clocked at 2.2GHz. Its non-optimized counterpart, shown in Figure 2 (a), runs in 11.7 seconds, nearly twice as fast. This difference increases with the size of `N`.

To avoid the scenario described in Example 3.19, 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. 3.2) 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.

4 EVALUATION

The goal of this section is to answer the following research question:

RQ1 How prevalent are function calls where at least one actual parameter could be lazified?
RQ2 How much speedup can be obtained by our implementation of lazification?
RQ3 How does our implementation of lazification impact the size of optimized code?

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.

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 could be plugged into the LLVM test suite. The experiments in Section 4.1 use the 27 programs from SPEC CPU2017. The experiments in Section 4.2 (and consequently those from Sec. 4.3) must run in the LLVM test-suite, for they require two inputs, one for profiling and another

for validation. These experiments use 772 programs (18 from SPEC CPU2017). We use the “train" input for profiling, and the “reference" for validation. Notice that 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 3.2).

Measurement Methodology: Running time numbers reported in this section are the average of 20 executions. We adopt a confidence level of 95%; thus, we consider as statistically significant only differences in running time with a p-value under 0.05. We use Student’s T-Test to determine whether the mean running times of two populations of executions are statistically different.

4.1 RQ1 - Prevalence

Our technique applies to promising functions, as formalized in Definition 3.2. However, as mentioned in Section 3.2, the existence of paths in the function’s CFG that do not use all its formal parameters does not guarantee that lazification is profitable. Profitability would require these paths to be traversed in practice. This section measures empirically how often such paths are traversed.

Methodology: We run a dynamic analysis on the C/C++ programs in SPEC CPU2017, using an LLVM pass to instrument the usage of formal arguments. The benchmarks were invoked with their “reference" inputs. Instrumentation collects occurrences of function calls whose arguments were not used before returning. In this experiment, we mark as “optimizable" the functions that contain *at least one* formal parameter without recorded uses. We performed two separate experiments, each employing a different criteria to choose which functions to instrument:

All: In the first scenario, we instrumented *every* function in all binaries; hence, recording usage data for all function calls.

Promising: We restricted instrumentation to the promising functions formalized by Def. 3.2.

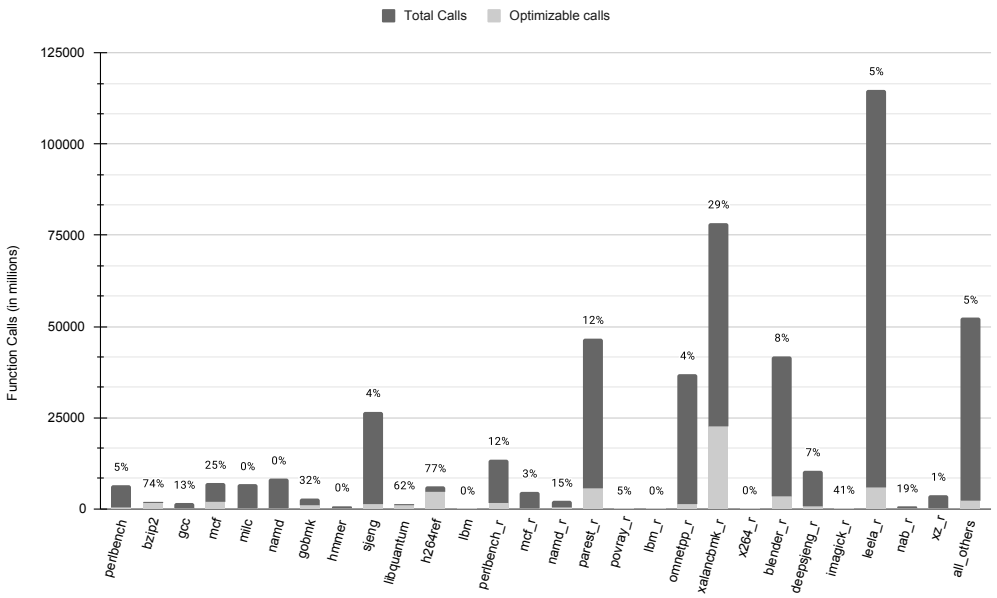

Fig. 8. Percentage of function calls with unused arguments, when tracking *all* functions.

Discussion: Figure 8 shows the result of the experiment for the scenario where all functions were instrumented. While there were a few programs with a significant percentage of optimizable calls, such as `bzip2` with 74% and `h264ref` with 77%, most programs have a low number of such calls. Additionally, most of the programs with high ratios of optimizable functions had fewer function calls in general. For instance, `leela_r` and `xalancbmk_r`, the only two programs with over 7 billion function calls, both had less than 30% of optimizable calls. Therefore, in relation to all function calls performed during execution, the prevalence of optimization opportunities is relatively low.

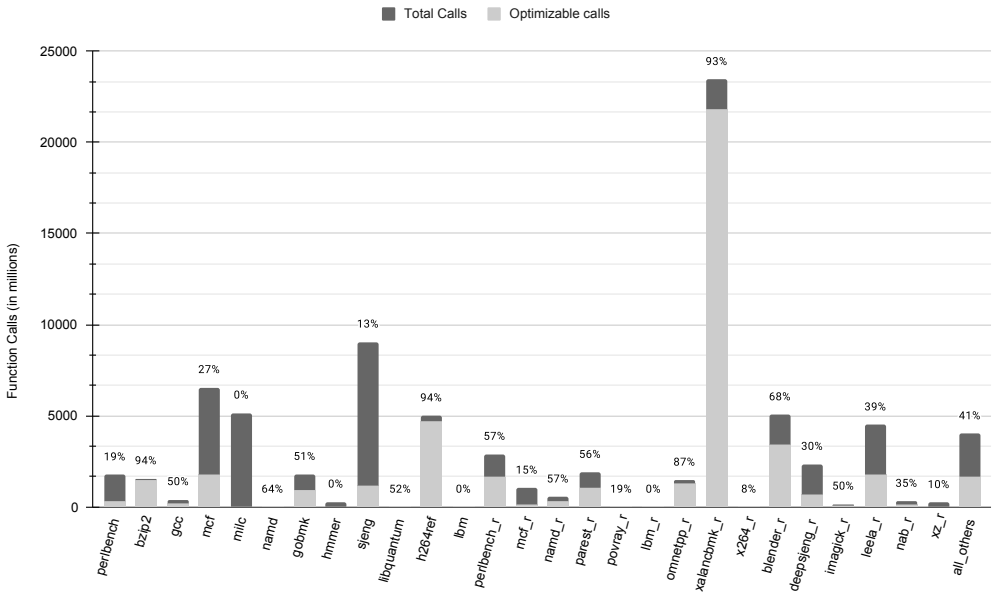

Fig. 9. Percentage of function calls with unused arguments, when tracking *promising* functions.

The aforementioned result was not surprising. Given the specificity of the pattern we target, we expected occurrences of it to be rare. However, if we restrict this analysis to promising functions, then we obtain a higher rate of optimizable functions. Figure 9 shows the result of this experiment. While the number of total calls drops significantly, there is still a non-negligible number of calls involving only promising functions. Additionally, a considerable amount of programs, including `xalancbmk_r`, which contains the most calls, have a ratio of over 50% of optimizable calls. Therefore, when analyzing contexts where the technique *could* be applied, we find that optimization opportunities are in fact significantly prevalent.

4.2 RQ2 - Running Time

This section measures the impact of lazification on the running time of programs, using, to this end, `clang -O3` as the baseline.

Methodology: We applied *Lazification* to all the programs in the LLVM test-suite, augmented with the C/C++ programs from the SPEC CPU2017 that have ports in the test-suite. We have evaluated three distinct setups for the optimization:

Naïve: we lazify *every* call site where a promising function is invoked, without any dynamic execution data.

PGO-0.4: we employ profiling as described in Section 3.5, optimizing only call sites whose parameter usage ratio is below 40%.

PGO-0.1: this is the same setup as PGO-0.4, but optimizing 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 10, 11 and 12 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). Speedup charts are presented in logarithmic scale. Programs from the LLVM test-suite are shown as square points. Programs from SPEC CPU2017 are shown as triangles, with labels indicating name and running-time variation.

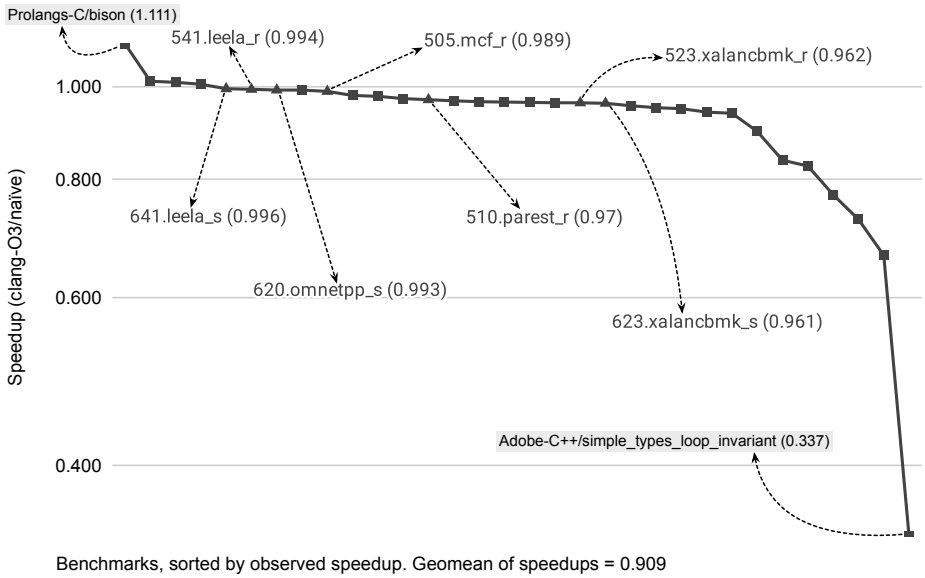

Fig. 10. Running time variation due to the naïve lazification, i.e., without the support of profiling data.

The Naïve Setup. Figure 10 shows the speedup for programs optimized using the naïve setup. In total, the optimization was applied to 111 programs. However, only 32 of them showed a statistically significant difference to the baseline; hence, we only show results for these. Speedups ranging from 1.11x for `bison` to 1.007x for `lua` were observed in four programs. No speedup could be measured in programs from SPEC CPU2017. Yet, the most dramatic slowdowns occurred for smaller benchmarks, as can be seen in the tail end of Figure 10. The biggest slowdown 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, as it does not use any form of profiling information. As seen in Section 4.1, most programs tend to have a low rate of function calls with unused arguments, even considering only promising functions. Therefore, unguided lazification tends to optimize many calls that cause regressions rather than benefits.

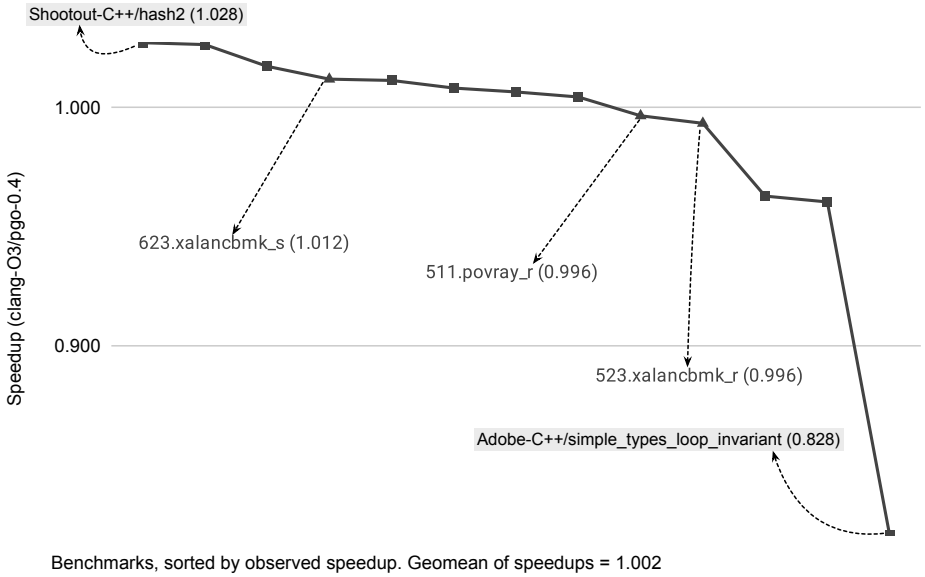

Fig. 11. Running time variation due to profiling-guided lazification, with a usage ratio of 0.4.

The PGO 4.0 Setup. Figure 11 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 13 showed mean runtimes which differed statistically from the baseline. Eight programs show speedups, while five had slowdowns. Notably, 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.

The PGO 1.0 Setup. Figure 12 shows the speedup results when optimizing with the PGO-0.1 setup. For this experiment, only six programs showed a statistically significant difference in runtime versus 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.

4.3 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 4.2.

Methodology: We measured the size, in bytes, of the `.text` section of the optimized binaries used in Section 4.2, 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 13 shows the result of this experiment. The vertical axis

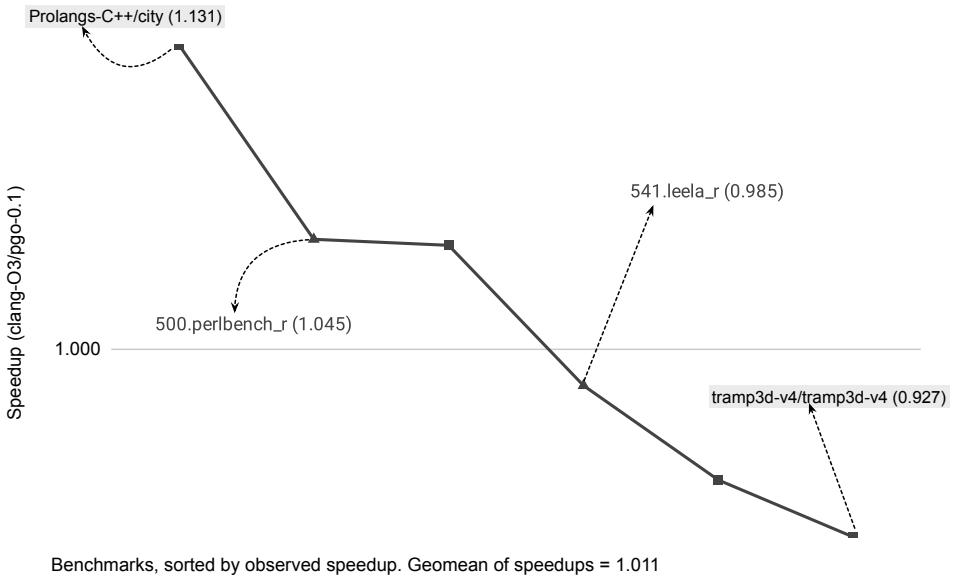

Fig. 12. Running time variation due to profiling-guided lazification, with a usage ratio of 0.1.

shows code bloat in logarithmic scale. Programs are sorted by the original binary's size, in bytes, plotted also in logarithmic scale in the horizontal axis.

Discussion. Our implementation of lazification leads to code expansion in the three different setups that we consider. The most aggressive setup, **naïve** naturally experiments the largest increase: on average, when considering the 111 benchmarks where our optimization finds service, we observe an expansion of 18.9% on top of `clang -O3`: we jump from 28.19 to 30.85 million LLVM instructions. If profiling data is used to restrict the scope of the optimization, the code expansion decreases, albeit remaining noticeable. The average expansion in the **pgo-4.0** setup is 11.6%, whereas in the **pgo-1.0** setup it is of 9.2%.

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. Nevertheless, the compiler-related literature abounds with examples of code motion that extrapolate the boundaries of functions. In this section, we go over four such techniques. Figure 14 shall provide a visual guide to such techniques.

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 [Chow et al. 1997; Kennedy et al. 1999; Knoop et al. 1992]. Lazification bears a few similarities with these approaches. In particular, like Knoop et al. [1992], 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

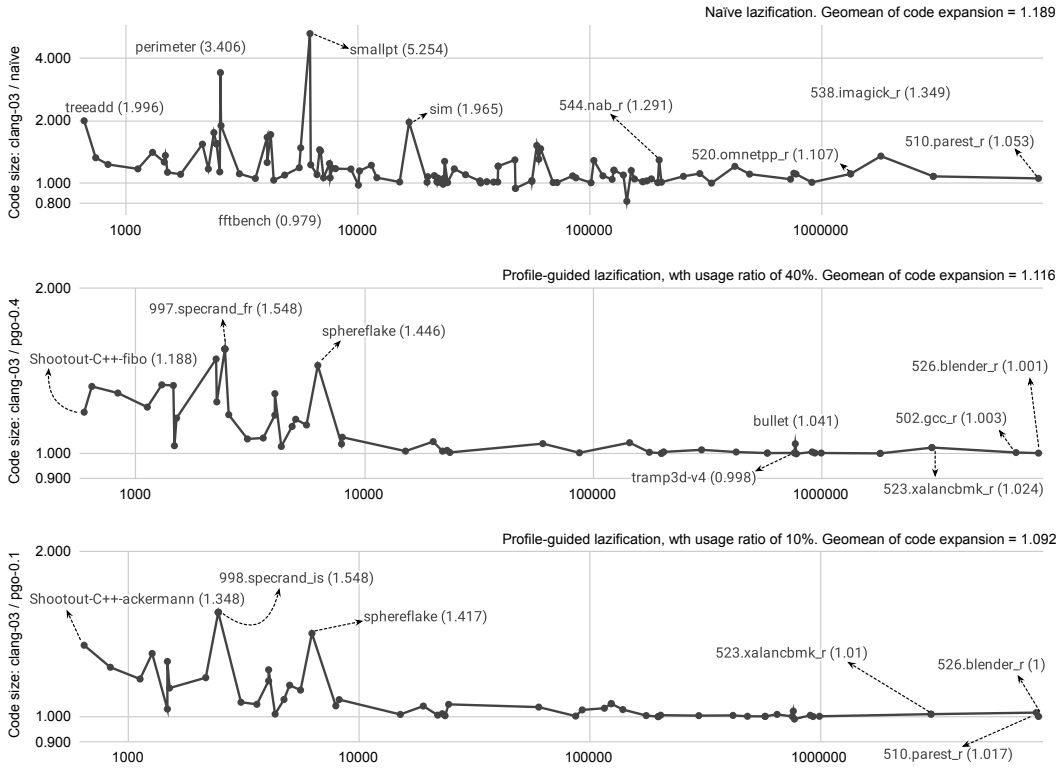

Fig. 13. Binary-size overhead of lazification, considering the three setups discussed in Section 4.2. X-axes show code size, in bytes, in log scale, ordered from smallest to largest binaries.

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.

Interprocedural Partial Redundancy Elimination. The notion of partial redundancy elimination has been extended to an interprocedural analysis by Agrawal et al. [1995]. Further developments of Agrawal et al.'s ideas have been used in the generation of communication routines during the automatic parallelization of programs [Jang et al. 2012; Rabbah and Palem 2003; Ren and Agrawal 2011]. The ideas of Agrawal et al. are very similar to those earlier proposed by Knoop et al. [1992]; 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 interprocedural optimization. In many ways, Agrawal et al.'s approach seems to doing 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.

(a) Partial redundancy elimination

```
01  a = exp();
02  if (b) {
03    c = a;
04  } else {
05    …
06  }
```

```
01  if (b) {
02    a = exp();
03    c = a;
04  } else {
05    …
06  }
```

(b) Interprocedural Partial redundancy elimination

```
01  f() {
02    while(a) {
03      g(a);
04      update(a);
05    }
06  }
07
08  g() {
09    exp();
10    use(a);
11  }
```

```
01  f() {
02    exp();
03    while(a) {
04      g(a);
05      update(a);
06    }
07  }
08
09  g() {
10    use(a);
11  }
```

(c) Stricness analysis and optimization

```
01  g(x, y) {
02    if (x == 0) {
03      return y;
04    } else {
05      return g(x-1, y)
06    }
07  }
08
09  g(1000, lazy_exp())
```

```
01  g(x, y) {
02    if (x == 0) {
03      return y;
04    } else {
05      return g(x-1, y)
06    }
07  }
08
09  g(1000, eager_exp())
```

(d) API-level lazification

```
01  a = eager_exp();
02  b = eager_exp();
03  c = eager_exp(a, b);
04  update(a);
05  d = eager_exp(c);
06  print(d);
```

```
01  a = lazy_exp();
02  b = lazy_exp();
03  c = lazy_exp(a, b);
04  eval(c);
05  update(a);
06  d = lazy_exp();
07  eval(d);
08  print(d);
```

Fig. 14. Examples of transformations produced by different code-motion techniques. For each example, the figure shows the original program on the left side, and the optimized program on the right side.

Strictness Analysis. In the late eighties, different research groups invented techniques to transform lazy evaluation of actual parameters into eager evaluation [Burn et al. 1986; Clack and Peyton Jones 1985; Kuo and Mishra 1987; Wadler 1988]. These transformations rely on a data-flow analysis presented by Mycroft [1981] in his Ph.D thesis, which became later known as *strictness analysis*. In the words of Burn et al. [1986]: "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.

API-level Lazification. Recently, Zhang and Shen [2021] 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` [Palkar et al. 2018] or `Delite` [Sujeeth et al. 2014]. 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. Therefore, transformations such as slicing or gating are of no service in their case.

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 [1984]. A number of classic surveys constitute today the canonical literature on the subject [Binkley and Gallagher 1996; Tip 1994; Xu et al. 2005]. More recently, Silva [2012] 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* [Sridharan et al. 2007] 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 5.1 illustrates how this notion of program slice differs from the concept that we use in this paper.

Example 5.1. The classic definitions of program slicing [Binkley and Gallagher 1996; Tip 1994; Xu et al. 2005], including the more modern treatment found in the work of Silva [2012], would not consider the use of variable **x** in Figure 4 (a) as a slice criterion. Rather, they would consider as a slice criterion the instruction at Line 12, e.g., `r = bar(x)`. In this case, the slice contains the conditional at Line 11, plus the switch at Line 09 of Figure 4 (a). None of these lines are part of the slice that we create, as Figure 4 (d) shows.

The difference between our work and previous techniques, which Example 5.1 illustrates, persists even in recent work that performs program slicing in the context of the LLVM compiler, using SSA-form programs [Zhang 2021]. Thus, we believe that the declarative algorithm in Definition 3.12, plus the use of gating predicates, as stated in Definition 3.7, are contributions of this paper. We emphasize that this combination of techniques works for the construction of idempotent slices.

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.

We have tested our ideas in an imperative setting: a collection of programs written in C or C++, and compiled with `clang`. In hindsight, one of the core challenges that we have faced was the difficulty to identify and avoid code with side effects. This difficulty has forced us to settle for a rather conservative analysis to mark "lazifiable" expressions. We believe that our lazification of function arguments could find more service in the context of a functional programming language that uses strict evaluation by default, such as Elixir, Idris, F# or ML. Nevertheless, the verification of this hypothesis is a challenge that we leave to the functional programming enthusiasts.

REFERENCES

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

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

Andrew Ayers, Richard Schooler, and Robert Gottlieb. 1997. Aggressive Inlining. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 134–145. https://doi.org/10.1145/258915.258928

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.

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

Edwin C. Brady. 2013a. 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

Edwin C. Brady. 2013b. Idris: General Purpose Programming with Dependent Types. In *PLPV*. Association for Computing Machinery, New York, NY, USA, 1–2. https://doi.org/10.1145/2428116.2428118

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*. Association for Computing Machinery, New York, NY, USA, 25–32. https://doi.org/10.1145/512529.512534

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

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

P. P. Chang and W.-W. Hwu. 1989. Inline Function Expansion for Compiling C Programs. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 246–257. https://doi.org/10.1145/73141.74840

Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. 1992. Profile-Guided Automatic Inline Expansion for C Programs. *Softw. Pract. Exper.* 22, 5 (may 1992), 349–369.

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*. Association for Computing Machinery, New York, NY, USA, 273–286. https://doi.org/10.1145/258915.258940

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.

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*. Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/75277.75280

Rokiatou Diarra. 2018. *Towards Automatic Restrictification of CUDA Kernel Arguments*. Association for Computing Machinery, New York, NY, USA, 928–931. https://doi.org/10.1145/3238147.3241533

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

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

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

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

ISO. 2011. *ISO/IEC 14882:2011 Information technology — Programming languages — C++* (third ed.). pub-ISO, Geneva, Switzerland. 1374 pages.

ISO. 2017. *ISO/IEC 14882:2017 Information technology — Programming languages — C++* (fifth ed.). pub-ISO, Geneva, Switzerland. 1605 pages.

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

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

Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1992. Lazy Code Motion. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 224–234. https://doi.org/10.1145/143095.143136

T.-M. Kuo and P. Mishra. 1987. On Strictness and Its Analysis. In *POPL*. Association for Computing Machinery, New York, NY, USA, 144–155. https://doi.org/10.1145/41625.41638

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

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

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

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*. Association for Computing Machinery, New York, NY, USA, 257–271. https://doi.org/10.1145/93542.93578

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

Tommaso Petrucciani, Giuseppe Castagna, Davide Ancona, and Elena Zucca. 2018. Semantic subtyping for non-strict languages. *CoRR* abs/1810.05555, Article 1810.05555 (2018), 40 pages. http://arxiv.org/abs/1810.05555

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

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

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

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

Victor Hugo Sperle Campos, Péricles Rafael Alves, Henrique Nazaré Santos, and Fernando Magno Quintão Pereira. 2016. Restrictification of Function Arguments. In *Compiler Construction*. Association for Computing Machinery, New York, NY, USA, 163–173. https://doi.org/10.1145/2892208.2892225

Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin Slicing. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 112–122. https://doi.org/10.1145/1250734.1250748

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

Guy Steele. 1990. *Common LISP: the language*. Elsevier, Amsterdam, The Netherlands.

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

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

Peng Tu and David Padua. 1995. Efficient Building and Placing of Gating Functions. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 47–55. https://doi.org/10.1145/207110.207115

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

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.

P. Wadler. 1988. Strictness Analysis Aids Time Analysis. In *POPL*. Association for Computing Machinery, New York, NY, USA, 119–132. https://doi.org/10.1145/73560.73571

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

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

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

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